

# 2013 IEEE International Conference on Software Maintenance

## ICSM 2013

### Table of Contents

Foreword.....	xii
Organising Committee.....	xiv
Technical Program Committee.....	xvi
Sub-Reviewers.....	xx
Keynotes.....	xxii

---

#### Testing

A Fuzzy Expert System for Cost-Effective Regression Testing Strategies .....	1
<i>Amanda Schwartz and Hyunsook Do</i>	
Identifying Process Improvement Targets in Test Processes: A Case Study .....	11
<i>Tanja Toroi, Anu Raninen, and Lauri Väättäinen</i>	
On Rapid Releases and Software Testing .....	20
<i>Mika V. Mäntylä, Foutse Khomh, Bram Adams, Emelie Engström, and Kai Petersen</i>	

#### Code Cloning

How Multiple Developers Affect the Evolution of Code Clones .....	30
<i>Jan Harder</i>	
Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules .....	40
<i>Wenyi Qian, Xin Peng, Zhenchang Xing, Stan Jarzabek, and Wenyun Zhao</i>	
An Empirical Study of Clone Removals .....	50
<i>Saman Bazrafshan and Rainer Koschke</i>	

#### APIs

Content Categorization of API Discussions .....	60
<i>Daqing Hou and Lingfeng Mo</i>	
An Empirical Study of API Stability and Adoption in the Android Ecosystem .....	70
<i>Tyler McDonnell, Baishakhi Ray, and Miryung Kim</i>	
How We Design Interfaces, and How to Assess It .....	80
<i>Hani Abdeen, Houari Sahraoui, and Osama Shata</i>	

## Runtime Analysis

An Accurate Stack Memory Abstraction and Symbolic Analysis Framework for Executables .....	90
<i>Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos Keromytis</i>	
An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs .....	100
<i>Rui Xin, Zhengwei Qi, Shiqiu Huang, Chengcheng Xiang, Yudi Zheng, Yin Wang, and Haibing Guan</i>	
Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues .....	110
<i>Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora</i>	

## Reverse Engineering

Exploring the Limits of Domain Model Recovery .....	120
<i>Paul Klint, Davy Landman, and Jurgen Vinju</i>	
Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams .....	130
<i>Yvan Labiche, Bojana Kolbah, and Hossein Mehrfard</i>	
An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams .....	140
<i>Mohd Hafeez Osman, Michel R.V. Chaudron, and Peter van der Putten</i>	

## Refactoring

Output-Oriented Refactoring in PHP-Based Dynamic Web Applications .....	150
<i>Hoan Anh Nguyen, Hung Viet Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen</i>	
On the Automation of Dependency-Breaking Refactorings in Java .....	160
<i>Syed Muhammad Ali Shah, Jens Dietrich, and Catherine McCartin</i>	
Reducing the Energy Consumption of Mobile Applications Behind the Scenes .....	170
<i>Young-Woo Kwon and Eli Tilevich</i>	

## Fault and Defect Management

Efficient Automated Program Repair through Fault-Recorded Testing Prioritization .....	180
<i>Yuhua Qi, Xiaoguang Mao, and Yan Lei</i>	
Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names .....	190
<i>Giuseppe Scanniello and Michele Risi</i>	
DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis .....	200
<i>Yuan Tian, David Lo, and Chengnian Sun</i>	

## Software Comprehension

An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks .....	210
<i>Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella</i>	
Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support .....	220
<i>Leo Pruijt, Christian Köppe, and Sjaak Brinkkemper</i>	
LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines .....	230
<i>Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta</i>	

## Software Authorship

Mining Software Profile across Multiple Repositories for Hierarchical Categorization .....	240
<i>Tao Wang, Huaimin Wang, Gang Yin, Charles X. Ling, Xiang Li, and Peng Zou</i>	
Mining Software Repositories for Accurate Authorship .....	250
<i>Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat</i>	

## Smells and Anti-patterns

Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains .....	260
<i>Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka</i>	
Predicting Bugs Using Antipatterns .....	270
<i>Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan</i>	

## Dependencies

The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache .....	280
<i>Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella</i>	
Stakeholders' Information Needs for Artifacts and Their Dependencies in a Real World Context .....	290
<i>Sebastian C. Müller and Thomas Fritz</i>	

## Feature Location

Improving Feature Location by Enhancing Source Code with Stereotypes .....	300
<i>Nouh Alhindawi, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic</i>	
Will Fault Localization Work for These Failures? An Automated Approach to Predict Effectiveness of Fault Localization Tools .....	310
<i>Tien-Duy B. Le and David Lo</i>	

## Traceability

Enhancing Software Traceability by Automatically Expanding Corpora with Relevant Documentation .....	320
<i>Tathagata Dasgupta, Mark Grechanik, Evan Moritz, Bogdan Dit, and Denys Poshyvanyk</i>	
Supporting and Accelerating Reproducible Research in Software Maintenance Using TraceLab Component Library .....	330
<i>Bogdan Dit, Evan Moritz, Mario Linares-Vásquez, and Denys Poshyvanyk</i>	

## Context

Social Activities Rival Patch Submission for Prediction of Developer Initiation in OSS Projects .....	340
<i>Mohammad Gharehyazie, Daryl Posnett, and Vladimir Filkov</i>	
How Does Context Affect the Distribution of Software Maintainability Metrics? .....	350
<i>Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan</i>	

## ERA

Refactoring Clones: An Optimization Problem .....	360
<i>Giri Panamootil Krishnan and Nikolaos Tsantalis</i>	
Multi-abstraction Concern Localization .....	364
<i>Tien-Duy B. Le, Shaowei Wang, and David Lo</i>	
Towards a Weighted Voting System for Q&A Sites .....	368
<i>Daniele Romano and Martin Pinzger</i>	
Latent Co-development Analysis Based Semantic Search for Large Code Repositories .....	372
<i>Rahul Venkataramani, Allahbakhsh Asadullah, Vasudev Bhat, and Basavaraju Muddu</i>	
Differentiating Roles of Program Elements in Action-Oriented Concerns .....	376
<i>Emily Hill, David Shepherd, Lori Pollock, and K. Vijay-Shanker</i>	
Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization .....	380
<i>Tien-Duy B. Le, Ferdian Thung, and David Lo</i>	
An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings .....	384
<i>Quinten David Soetens, Javier Perez, and Serge Demeyer</i>	
Automatically Extracting Instances of Code Change Patterns with AST Analysis .....	388
<i>Matias Martinez, Laurence Duchien, and Martin Monperrus</i>	
Identification of Refused Bequest Code Smells .....	392
<i>Elvis Ligu, Alexander Chatzigeorgiou, Theodore Chaikalis, and Nikolaos Ygeionomakis</i>	
Code Smell Detection: Towards a Machine Learning-Based Approach .....	396
<i>Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä</i>	
Variations on Using Propagation Cost to Measure Architecture Modifiability Properties .....	400
<i>Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Julien Delange, Marco González, and Philippe Kruchten</i>	
Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing .....	404
<i>Roberto E. Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba</i>	

Which Feature Location Technique is Better? .....	408
<i>Emily Hill, Alberto Bacchelli, Dave Binkley, Bogdan Dit, Dawn Lawrie, and Rocco Oliveto</i>	
Automatic Means of Identifying Evolutionary Events in Software Development .....	412
<i>Siim Karus</i>	
Towards Understanding Large-Scale Adaptive Changes from Version Histories .....	416
<i>Omar Meqdadi, Nouh Alhindawi, Michael L. Collard, and Jonathan I. Maletic</i>	
Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness? .....	420
<i>Tosin Daniel Oyetoyan, Daniela Soares Cruzes, and Reidar Conradi</i>	
Towards a Taxonomy of Programming-Related Difficulties during Maintenance .....	424
<i>Aiko Yamashita and Leon Moonen</i>	
A Pilot Experiment to Quantify the Effect of Documentation Accuracy on Maintenance Tasks .....	428
<i>Maurizio Leotta, Filippo Ricca, Giuliano Antoniol, Vahid Garousi, Junji Zhi, and Guenther Ruhe</i>	
Task-Driven Software Summarization .....	432
<i>Dave Binkley, Dawn Lawrie, Emily Hill, Janet Burge, Ian Harris, Regina Hebig, Oliver Keszocze, Karl Reed, and John Slankas</i>	
Determining “Grim Reaper” Policies to Prevent Languishing Bugs .....	436
<i>Patrick Francis and Laurie Williams</i>	
Which Practices Are Suitable for an Academic Software Project? .....	440
<i>Václav Rajlich and Jing Hua</i>	
WSDARWIN: A Decision-Support Tool for Web-Service Evolution .....	444
<i>Marios Fokaefs and Eleni Stroulia</i>	
A Study on Developers’ Perceptions about Exception Handling Bugs .....	448
<i>Felipe Ebert and Fernando Castor</i>	
On the Relationship between the Vocabulary of Bug Reports and Source Code .....	452
<i>Laura Moreno, Wathsala Bandara, Sonia Haiduc, and Andrian Marcus</i>	
Database-Aware Fault Localization for Dynamic Web Applications .....	456
<i>Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen</i>	
On the Personality Traits of StackOverflow Users .....	460
<i>Blerina Bazelli, Abram Hindle, and Eleni Stroulia</i>	
Towards Identification of Software Improvements and Specification Updates by Comparing Monitored and Specified End-User Behavior .....	464
<i>Tobias Roehm, Bernd Bruegge, Tom-Michael Hesse, and Barbara Paech</i>	
An Empirical Illustration to Validate a FLOSS Development Model Using S-Shaped Curves .....	468
<i>Ana Erika Camargo Cruz, Hajimu Iida, and Norbert Preining</i>	
Understanding Schema Evolution as a Basis for Database Reengineering .....	472
<i>Maxime Gobert, Jérôme Maes, Anthony Cleve, and Jens Weber</i>	

## Tools

SAMOA—A Visual Software Analytics Platform for Mobile Applications .....	476
<i>Roberto Minelli and Michele Lanza</i>	
Towards a Scalable Cloud Platform for Search-Based Probabilistic Testing .....	480
<i>Louis M. Rose, Simon Poulding, Robert Feldt, and Richard F. Paige</i>	
LHDiff: Tracking Source Code Lines to Support Software Maintenance Activities .....	484
<i>Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta</i>	
gCad: A Near-Miss Clone Genealogy Extractor to Support Clone Evolution Analysis .....	488
<i>Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider</i>	
eCITY: A Tool to Track Software Structural Changes Using an Evolving City .....	492
<i>Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer</i>	
ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications .....	496
<i>Juan Castrejón, Genoveva Vargas-Solar, Christine Collet, and Rafael Lozano</i>	
A Visualization Tool for Reverse-Engineering of Complex Component Applications .....	500
<i>Lukas Holy, Jaroslav Snajberk, Premek Brada, and Kamil Jezek</i>	
Interactive Exploration of Collaborative Software-Development Data .....	504
<i>Eleni Stroulia, Isaac Matichuk, Fabio Rocha, and Ken Bauer</i>	
SourceMiner Evolution: A Tool for Supporting Feature Evolution Comprehension .....	508
<i>Renato L. Novais, Camila Nunes, Alessandro Garcia, and Manoel Mendonça</i>	
CONQUER: A Tool for NL-Based Query Refinement and Contextualizing Code Search Results .....	512
<i>Manuel Roldan-Vega, Greg Mallet, Emily Hill, and Jerry Alan Fails</i>	
srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration .....	516
<i>Michael L. Collard, Michael John Decker, and Jonathan I. Maletic</i>	
TRINITY: An IDE for the Matrix .....	520
<i>Jeroen van den Bos and Tijds van der Storm</i>	

## Industry

E-Xplore: Enterprise API Explorer .....	524
<i>Allahbaksh M. Asadullah, M. Basavaraju, and Nikita Jain</i>	
Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features .....	528
<i>Nataliia Semenenko, Marlon Dumas, and Tönis Saar</i>	
Automated Classification of Static Code Analysis Alerts: A Case Study .....	532
<i>Ulas Yüksel and Hasan Sözer</i>	
Mining Telecom System Logs to Facilitate Debugging Tasks .....	536
<i>Alf Larsson and Abdelwahab Hamou-Lhadj</i>	
Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study .....	540
<i>Dusica Marijan, Arnaud Gotlieb, and Sagar Sen</i>	
Improving Statistical Approach for Memory Leak Detection Using Machine Learning .....	544
<i>Vladimir Šor, Plumb Oü, Tarvo Treier, and Satish Narayana Srirama</i>	

Large-Scale Automated Refactoring Using ClangMR .....	548
<i>Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan</i>	
Assuming Software Maintenance of a Large, Embedded Legacy System from the Original Developer .....	552
<i>William L. Miller, Lawrence B. Compton, and Bruce L. Woodmansee</i>	
The Adventure of Developing a Software Application on a Pre-release Platform: Features and Learned Lessons .....	556
<i>Clairton Siebra, Angelica Mascaro, Fabio Q.B. Silva, and Andre L.M. Santos</i>	
 <b>Doctoral Symposium</b>	
Analysis of Multi-dimensional Code Couplings .....	560
<i>Fabian Beck</i>	
How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study .....	566
<i>Aiko Yamashita</i>	
Refactoring Planning for Design Smell Correction: Summary, Opportunities and Lessons Learned .....	572
<i>Javier Pérez</i>	
Revealing the Effect of Coding Practices on Software Maintainability .....	578
<i>Péter Hegedus</i>	
Automated S/W Reengineering for Fault-Tolerant and Energy-Efficient Distributed Execution .....	582
<i>Young-Woo Kwon</i>	
Reverse Engineering Web Sales Configurators .....	586
<i>Ebrahim Khalil Abbasi</i>	
 <b>Author Index</b> .....	 590

# A Fuzzy Expert System for Cost-Effective Regression Testing Strategies

Amanda Schwartz, Hyunsook Do  
 North Dakota State U.  
 {amanda.j.schwartz, hyunsook.do}@ndsu.edu

**Abstract**—Different testing environments and software change characteristics can affect the choice of regression testing techniques. In our prior work, we developed adaptive regression testing (ART) strategies to investigate this problem. While the ART strategies showed promising results, we also found that the multiple criteria decision making processes required for the ART strategies are time-consuming, often inaccurate and inconsistent, and limited in their scalability. To address these issues, in this research, we develop and empirically study a fuzzy expert system (FESART) to aid decision makers in choosing the most cost-effective technique for a particular software version. The results of our study show that FESART is consistently more cost-effective than the previously proposed ART strategies. One of the biggest contributors to FESART being more cost-effective is the reduced time required to apply the strategy. This contribution has significant impact because a strategy that is less time-consuming will be easier for researchers and practitioners to adopt, and will provide even greater cost-savings for regression testing sessions.

**Index Terms**—Regression testing, test case prioritization, adaptive regression testing strategy, AHP, fuzzy AHP, empirical studies

## I. INTRODUCTION

Software maintenance is a large part of the software development life-cycle. Maintaining a software system includes many different tasks, such as fixing defects, adding new features, or modifying the software to accommodate different environments. After the software system has been modified, it needs to be tested to ensure that the changes did not have any adverse effects on the previously validated code. Regression testing is the process of checking modified software systems to ensure continued quality. Regression testing is often performed by re-running existing tests from previous versions along with new tests which test new features.

However, as software systems grow, the size of the test suite can become too large, making it too time-consuming and costly to run all the tests. For example, in [1], one company mentioned has a software product with a regression test suite containing over 30,000 test cases that requires over 1,000 machine hours to execute. In all situations, including that one, regression testing is still necessary to ensure the continued quality of the software system. However, requiring 1,000 hours to run all test cases is not a feasible option, so reducing the cost and time required for regression testing sessions has considerable importance.

Many regression testing techniques and maintenance approaches have been proposed to reduce the costs of regression testing, such as test case prioritization, test case selection, and test case minimization. Also, empirical studies have been

performed to evaluate the cost-effectiveness of the different regression testing techniques. Some of these studies have shown that various environmental and testing factors affect the cost-effectiveness of the techniques [1], [2], [3]. Therefore, the technique which is most cost-effective for one version may not be the most cost-effective for every version of a software system. We can say that there is no single regression testing technique that is the most cost-effective for every version of a software system.

Because there is no single technique which is most cost-effective for every version of a software system, there is potential for large cost-savings by choosing the most cost-effective technique for each software version considering various factors that affect the overall costs and benefits. However, very little research has been done on the problem of helping practitioners choose appropriate techniques [4], [5] under particular testing environments. To address this issue, in our prior studies [6], [7], we investigated *adaptive regression testing* (ART) strategies that try to choose the most cost-effective regression testing techniques for each regression testing session considering various evaluation factors. In the first study [6], we utilized the analytical hierarchy process (AHP) [8] to choose the best test case prioritization techniques across the system lifetime. In the second study [7], we conducted additional research using fuzzy AHP to address the problem of imprecision by decision makers in the pairwise comparisons that we observed in the first study. The results of both studies indicate that the techniques chosen by ART strategies are consistently more cost-effective than those used by approaches that do not consider system lifetime and testing processes.

Although the prior studies showed promising results, there are still several limitations which remain with the proposed ART strategies. First, comparisons made by the decision maker during the pairwise comparison process are often inconsistent [9], [10]. Judgements made for one comparison often contradict judgements made for another comparison. Second, the pairwise comparisons are very time-consuming for the decision maker [11], [12]. Third, the use of pairwise comparisons is not scalable. Because of the work required by pairwise comparisons, there is a limit to the number of criteria and alternatives that can be considered [13]. To address these problems, other decision making methods need to be considered.

One method which has frequently been used to solve problems which normally require human experts is a fuzzy

expert system (e.g., fuzzy expert systems have been used in the medical field to diagnose heart disease [14] and back pain [15]). Fuzzy expert systems provide a mechanism to simulate the judgement and reasoning of experts in the particular field. Fuzzy expert systems have two major components which simulate expert judgement: a knowledge base and an inference engine. The knowledge base contains knowledge about the particular domain which is used by human experts to solve the problem, and the inference engine contains a set of rules which utilizes the knowledge to determine appropriate output.

We believe that fuzzy expert systems are able to address the limitations of the previously proposed ART strategies for the following reasons. First, a fuzzy expert system does not require pairwise comparisons, so the issue of inconsistencies with the comparisons is eliminated. Second, without pairwise comparisons, there is less input needed from the decision maker, making it less time-consuming for the decision maker. A method which requires pairwise comparisons with  $n$  alternatives requires  $\frac{n^2-n}{2}$  comparisons for each criteria, where a fuzzy expert system would only require  $n$  number of inputs for each criteria. In addition to a fuzzy expert system being less time-consuming because it requires fewer input per criteria and alternative, a fuzzy expert system is more scalable because the input required per criteria and alternative does not grow as quickly as a method with pairwise comparisons.

In this research, we develop and empirically study a fuzzy expert system for ART. In the next section, we describe the background information and related work relevant to prioritization techniques and decision making strategies. Section III describes fuzzy expert systems, including how we developed a fuzzy expert system for ART. Section IV presents our experiment design and discusses threats to validity. Section V presents the results of the study and data analysis. Section VI discusses our results, and Section VII presents our conclusions and future work.

## II. BACKGROUND AND RELATED WORK

To date, many regression testing techniques have been proposed and empirically evaluated, but here, we limit our discussion to test case prioritization techniques which are most closely related to our work. Further, we provide related work relevant to decision making processes by focusing on AHP and fuzzy expert systems.

### A. Test Case Prioritization

Test case prioritization techniques reorder test cases according to some goal (e.g., increasing the rate of fault detection) so that maximum benefit can be achieved even if testing is halted early. To date, many other prioritization techniques have been proposed, and a recent survey by Yoo and Harman [16] provides an overview of many different techniques.

With many different techniques available, recent research has begun to include empirical studies to evaluate the cost-benefit trade-offs among techniques by considering various factors and testing contexts [1], [2], [3]. These studies show that various techniques have strong potential for reducing the cost of regression testing, but the studies also reveal

wide variances in performance. The varying performance is attributed to the different factors involving the program under test, the test suites used to test them, the types of program modifications, and the testing processes. More recent studies [6], [7] introduced adaptive regression testing (ART) strategies to identify the most cost-effective regression testing techniques for each regression testing session.

### B. Multiple Criteria Decision Making (MCDM) Methods

Choosing a prioritization technique involves many different factors which have trade-offs. These trade-offs are considered to be conflicting criteria. A problem which has multiple conflicting criteria is known as a multiple criteria decision making (MCDM) problem.

Analytic Hierarchy Process (AHP) is one of the widely used MCDM methods. It has been used in many different areas. For instance, Aull-Hyde and Davis [17] discuss how AHP is used as an important tool for decision making in the U.S. military, and Subramanian and Ramanathan [18] provide a review of how AHP is used in operations management. Further, AHP has recently been used in software engineering areas, such as aiding early effort estimations [19] and prioritizing software requirements [20]. Yoo and Harman [21] cluster test cases and use the AHP method for prioritization of the clustered test cases. Although AHP is a widely used method for decision making problems, it has been noted throughout the literature that there are several limitations to this method, such as its subjectiveness of decision maker's judgements [22], [23], inconsistency in pairwise comparisons [10], time-consuming comparison process [11], [24], and scalability problem (a limit of  $7 \pm 2$  alternatives is suggested in [13]).

Fuzzy AHP has been used by many researchers to address the imprecision in judgments made by the decision maker [25], [22], and was used as an ART strategy in [7]. However, fuzzy AHP still requires pairwise comparisons and, therefore, has the drawbacks of inconsistent comparisons, being very time-consuming, and not being very scalable.

Fuzzy expert systems have been used in many different domains to aid in complex decision making problems. For example, fuzzy expert systems have been developed in the medical field to diagnose heart disease [14] and back pain [15], and in economics for choosing stock in the stock exchange [26]. Fuzzy expert systems have also been developed in the area of software engineering. They have been used frequently for software cost estimation [27], [28]. There has been very little use in the area of software testing, however. Xu et. al developed a fuzzy expert system to build a new test selection technique [29]. Our work develops a fuzzy expert system that helps choose the most cost-effective regression testing technique for regression testing sessions. In the next section, we discuss fuzzy expert systems in more detail as well as how a fuzzy expert system can be developed for ART.

## III. FUZZY EXPERT SYSTEMS

In this section, we describe fuzzy expert systems, and how a fuzzy expert system can be utilized for creating a new Adaptive Regression Testing (ART) strategy.

### A. Fuzzy Expert Systems

A fuzzy expert system is an expert system comprised of fuzzy membership functions and rules. It contains three main parts: fuzzification, fuzzy inference, and defuzzification. A fuzzy expert system is represented in Figure 1. This figure shows how crisp input is given by the decision maker to the fuzzification process, which determines a fuzzy input set. That fuzzy input set is used in the inference process. The fuzzy inference process uses fuzzy rules built from a knowledge base to determine the fuzzy output set. The fuzzy output set is then defuzzified in the defuzzification process to determine crisp output. Crisp output is then used by the decision maker in the decision making process. Each of these parts are explained in more detail in this section.

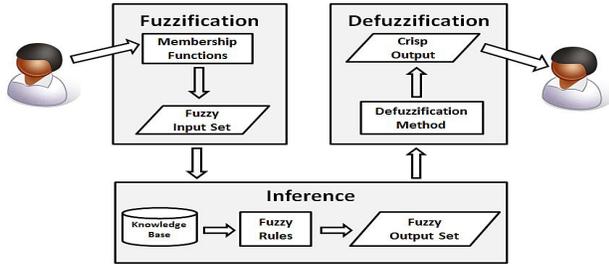


Fig. 1. Fuzzy Expert System.

### Fuzzy Set Theory

To understand the fuzzification process, some knowledge of fuzzy set theory is necessary. Fuzzy set theory, which was introduced by Zadeh in 1965 [30], defines fuzzy sets as an extension of conventional sets. In conventional sets, elements are considered to either be a part of a set or not be a part of a set. The membership,  $\mu_A(x)$ , of an element,  $x$ , of a classical set,  $A$ , is defined by the equation below:

$$\mu_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases} \quad (1)$$

Fuzzy sets allow partial membership. The degree of membership is calculated using a membership function which generates the degree of membership on the interval  $[0, 1]$ .

Fuzzy sets can be formally defined by:

$$A = (x, \mu_A(x)) | x \in X, \mu_A(x) : X \rightarrow [0, 1] \quad (2)$$

where  $A$  is the fuzzy set,  $\mu_A$  is the membership function, and  $X$  is the universe of discourse.

Fuzzy sets can be used effectively to represent linguistic values in a fuzzy expert system. For example, if you are going to describe the service and food at a restaurant, you may describe the service as *poor*, *average*, or *excellent* and the food as *poor*, *good*, or *delicious*. Because fuzzy sets allow partial membership, you would be able to specify a degree of membership for each fuzzy set, so the food could be classified as *somewhat poor* and *somewhat good*. By being able to

allow partial membership, fuzzy set theory can handle the imprecision of input from the decision makers.

### Fuzzification

The fuzzification process takes input from the decision maker and determines its degree of membership to the fuzzy sets using membership functions defined in the fuzzy expert system. Fuzzification is necessary to convert the input data into fuzzy sets for the inference engine to process. Three of the most commonly used membership functions are triangular, trapezoidal, and gaussian. The triangular membership function is described using three values  $(a, b, c)$  where  $b$  is the modal value,  $a$  is the minimum boundary, and  $c$  is the maximum boundary. The trapezoidal membership function is described using four values  $(a, b, c, d)$   $a$  is the minimum value,  $b$  is the minimum support value,  $c$  is the maximum support value, and  $d$  is the maximum value. The gaussian membership function transforms the values into a normal distribution with the midpoint defining the ideal definition for the set. The midpoint is assigned a degree of membership of 1.

### Fuzzy Inference

The fuzzy inference system takes the fuzzified input from the fuzzification process and determines fuzzy output. The fuzzy inference process maps all inputs,  $x = [x_1, x_2, \dots, x_n]$  to an output,  $f(x)$ . The mapping is done using fuzzy rules. The antecedent of the fuzzy rule defines the fuzzy region of the input space, and the consequent defines the fuzzy region of the output space. The fuzzy inference process is modeled in Figure 2. In this figure, the fuzzy inference process is shown in the area outlined by the dotted line. This particular inference system has three rules that are used to map the input,  $x$ , to an appropriate output set.  $A1$ ,  $A2$ , and  $A3$  are linguistic variables that categorize the input. Based on the categorized input, the rule determines the output ( $B1$ ,  $B2$ , or  $B3$ ). For example, using Rule 1, if  $x$  is categorized as linguistic variable  $A1$ , then the output set is  $B1$ .

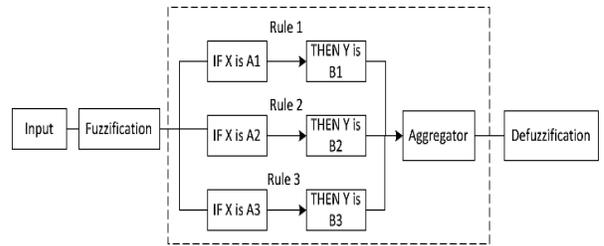


Fig. 2. Fuzzy Inference Process.

There are two popular inference systems: the Mamdani inference system [31] and the Takagi-Sugeno inference system [32]. In this research, we will be using the Mamdani inference system which is a more commonly used system, so we will limit our discussion to that inference system.

The first step in a Mamdani fuzzy inference system is to match the input to the fuzzy rules which have some degree of truth in the antecedent forming the fuzzy conclusion set. Then, the fuzzy rules in the fuzzy conclusion set are evaluated. The

next step of the fuzzy inference system is the aggregation of the rule output. All the *then*-parts of the rules are combined into a final output set. The final output set is a fuzzy set which requires defuzzification for the final output.

### Fuzzy Rules

A fuzzy rule is a conditional statement that uses linguistic variables. Fuzzy rules are used to determine output from fuzzy input. The knowledge needed to construct fuzzy rules in a fuzzy expert system comes from a combination of several different sources. The most widely used sources are human knowledge and expertise, historical data analysis of a system, and engineering knowledge from existing literature. Fuzzy rules express knowledge about the relationship between input and output variables. A generic fuzzy rule assumes the following form:

If  $x$  is  $A$  then  $y$  is  $B$

where  $A$  and  $B$  are linguistic values defined by fuzzy sets. The first part of the rule, the *if*-part, is called the antecedent, and the *then*-part is called the consequent. Any rule that has some truth in the antecedent will be included in the fuzzy conclusion set. In the fuzzy conclusion set, if the antecedent is true to some degree of membership, then the consequent is also true to that same degree of membership. Some rules may contain more than one input in the antecedent, and the input variables may be combined using fuzzy set operators such as *AND* or *OR*. A generic fuzzy rule with two inputs, one using *AND* and one using *OR* is:

If  $x$  is  $A$  *AND*  $y$  is  $B$  then  $z$  is  $C$

If  $x$  is  $B$  *OR*  $y$  is  $B$  then  $z$  is  $C$

where  $A, B,$  and  $C$  are linguistic values defined in the fuzzy set,  $x$  and  $y$  are the input variables, and  $z$  is the output variable. One of the most common ways for evaluating fuzzy rules with fuzzy operators is the Zadeh technique [30], which is also referred to as the min-max technique. The Zadeh technique for the fuzzy intersection takes the minimum degree of membership for the membership values of the antecedent. The technique is defined by:

$$\mu_{A \cap B}(x) = \min[\mu_A(x), \mu_B(x)] \quad (3)$$

The Zadeh technique for fuzzy union takes the maximum degree of membership for the membership values of the antecedent. The technique is defined by:

$$\mu_{A \cup B}(x) = \max[\mu_A(x), \mu_B(x)] \quad (4)$$

### Defuzzification

Defuzzification is the way the fuzzy output from the inference process is converted to a crisp value. Many different defuzzification techniques have been proposed, but the center of gravity is the most widely accepted and regarded as being accurate [33], [34]. The definition for the center of gravity is:

$$y^* = \frac{\int \mu_B(y)ydy}{\int \mu_B(y)dy} \quad (5)$$

where  $y^*$  is the defuzzified output,  $\mu_B(y)$  is the aggregated membership function, and  $y$  is the output variable.

### B. A Fuzzy Expert System for ART (FESART)

In this section, we outline the fuzzy expert system we developed for ART, FESART. We describe each of the main parts of a fuzzy expert system: fuzzification, fuzzy inference using fuzzy rules, and defuzzification.

#### Fuzzification

The fuzzification process takes input from the decision maker and determines its degree of membership to the fuzzy sets using the membership functions defined in the fuzzy expert system. The input provided by the decision maker contains information which would aid in the decision making process. For ART, we consider the following criteria:

- Cost of applying the test case prioritization technique: the time required to run a test case prioritization algorithm
- Cost of software artifact analysis: the costs of instrumenting programs and collecting test execution traces
- Cost of delayed fault detection: the waiting time for each fault to be exposed while executing test cases under a test case prioritization technique
- Cost of missed fault: the time required to correct missed faults

The decision maker will evaluate each criterion on a scale from 1 to 9, with 9 being a high cost. This input will then be fuzzified according to its degree of membership to the membership functions provided in the fuzzy expert system. The fuzzy expert system contains three triangular membership functions for each criterion being considered. Triangular membership functions are defined by three values ( $a, b, c$ ) where  $b$  is the modal value,  $a$  is the minimum boundary, and  $c$  is the maximum boundary. These membership functions are shown in Table I. The resulting fuzzy input set from the fuzzification process is used as input for the fuzzy inference process.

TABLE I  
MEMBERSHIP FUNCTION FOR INPUT VARIABLES

Linguistic Value	Triangular Fuzzy Numbers( $a, b, c$ )
<i>Low</i>	(-3, 1, 5)
<i>Average</i>	(1, 5, 9)
<i>High</i>	(5, 9, 13)

#### Fuzzy Inference

The fuzzy inference process takes the fuzzified input from the fuzzification process and determines the fuzzy output set. The fuzzy output set for FESART contains eight triangular membership functions. The output is rated on a scale from 1 to 9, with the membership functions being evenly distributed across these values. The membership functions are shown in Table II. The output set was built to categorize the overall cost for the regression testing technique and are categorized from low to high.  $L1, L2,$  and  $L3$  are considered low costs, with  $L1$  being the lowest. Then,  $A1$  and  $A2$  are categorized as average cost, with  $A1$  being lower than  $A2$ .  $H1, H2,$  and  $H3$  are all high costs, with  $H3$  being the highest cost.

The fuzzy output set is determined by using fuzzy rules. More detail about how the fuzzy rules for ART were developed is provided in the next section.

TABLE II  
MEMBERSHIP FUNCTION FOR OUTPUT VARIABLE

Linguistic Value	Triangular Fuzzy Numbers ( $a, b, c$ )
$L1$	(-14, 1, 2.14)
$L2$	(1, 2.14, 3.29)
$L3$	(2.14, 3.29, 4.43)
$A1$	(3.29, 4.43, 5.57)
$A2$	(4.43, 5.57, 6.71)
$H1$	(5.57, 6.71, 7.86)
$H2$	(6.71, 7.86, 9)
$H3$	(7.86, 9, 10.14)

### Fuzzy Rules

In a fuzzy expert system, the fuzzy rules bring expert knowledge into the system to aid in the decision making process. The knowledge needed to construct fuzzy rules in a fuzzy expert system comes from a combination of several different sources. The most widely used sources are human knowledge and expertise, historical data analysis of a system, and engineering knowledge from existing literature. To develop rules for a fuzzy expert system in ART, knowledge about the factors that influence cost-benefits for regression testing techniques is needed. To gain this knowledge, each of the previously mentioned methods were used. For example, from the literature [35], [36], it can be said the costs of delayed fault detection are greater than the costs related to setting up and running the test cases, so the fuzzy rules involving these two items would give more importance to the cost of delayed fault detection.

Each criterion was considered and evaluated through information gained from the methods listed above. Using this knowledge, the criteria were ordered by their impact on cost-benefit trade-offs. The order was determined to be the cost of missed faults ( $CF$ ), cost of delayed fault detection ( $CD$ ), cost of applying the prioritization techniques ( $CR$ ), and costs of software artifact analysis ( $CA$ ), with the cost of missed faults having the strongest impact and the cost of software artifact analysis having the least impact.

For each of the criterion possible combinations of membership functions were considered. There are four input variables, and three membership functions for each one, so there are 81 unique combinations. Each combination was evaluated and assigned an appropriate output set. Then, the rules were studied to see if any of them could be combined or eliminated. We were able to reduce the rule set to 67. The following example demonstrates how we were able to reduce the rule set. In the original rule set, the following three rules existed:

- IF  $CF$  is  $H$  and  $CD$  is  $H$  and  $CR$  is  $H$  and  $CA$  is  $H$  then Cost is  $H3$ .
- IF  $CF$  is  $H$  and  $CD$  is  $H$  and  $CR$  is  $H$  and  $CA$  is  $A$  then Cost is  $H3$ .
- IF  $CF$  is  $H$  and  $CD$  is  $H$  and  $CR$  is  $H$  and  $CA$  is  $L$  then Cost is  $H3$ .

For each possible value for  $CA$ , the output set was still  $H3$ . The value of  $CA$  did not have any impact on these particular rules, so we can reduce the three rules into the following rule:

IF  $CF$  is  $H$  and  $CD$  is  $H$  and  $CR$  is  $H$  then Cost is  $H3$

A subset of the fuzzy rules is shown in Table III. Because  $CF$  and  $CD$  were determined to have the most impact on the cost-benefit calculations and are classified as a high cost in these rules, the final cost is categorized in the different high

cost output sets ( $H1$ ,  $H2$ , and  $H3$ ). When  $CR$  and  $CA$  have a higher cost, then the final cost is determined on the higher end of the high output sets ( $H2$  or  $H3$ ). When  $CR$  and  $CA$  have a low cost, the final cost is still considered high because  $CF$  and  $CD$  are high and have more impact, but the cost on the lower end of the high cost ( $H1$ ).

Once the fuzzy set is determined through the use of the fuzzy rules, the defuzzification process is used to determine a crisp value. The defuzzification process is described in the next section.

TABLE III  
FUZZY RULES FOR ART

2. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $A$ and $CA$ is $H$ then Cost is $H3$
5. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $A$ and $CA$ is $A$ then Cost is $H2$
3. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $A$ and $CA$ is $L$ then Cost is $H2$
6. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $L$ and $CA$ is $H$ then Cost is $H2$
4. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $L$ and $CA$ is $A$ then Cost is $H1$
7. IF $CF$ is $H$ and $CD$ is $H$ and $CR$ is $L$ and $CA$ is $L$ then Cost is $H1$

### Defuzzification

Defuzzification is the way that the fuzzy output from the inference process is converted to a crisp value. Many different defuzzification techniques have been proposed. We chose to use the center of gravity method because it is the most widely accepted and is regarded as being accurate [33], [34]. The output provided from the defuzzification process gives a crisp value to use in the decision making process. In the case of ART, the goal is to identify the most cost-effective regression testing technique for a specific software version. The number provided by the defuzzification process for one technique can be used to compare with numbers from the defuzzification process for other techniques. For ART, the lower the number, the more cost-effective the regression testing strategy is. The technique with the lowest number is the technique which is expected to be the most cost-effective for that particular software version.

## IV. EMPIRICAL STUDY

In this study, we address the following research question:

RQ: Is a fuzzy expert system that considers testing environments and contexts more cost-effective than the previously proposed ART strategies at choosing the most cost-effective regression testing techniques across a system lifetime?

To investigate this research question, a controlled experiment was conducted. The experiment setup was similar to that of our earlier studies [6], [7], with the test case prioritization technique application mapping strategies considered being: AHP, fuzzy AHP, and FESART. The same object programs and versions from the previous study [6] were used. The following subsections present, for this experiment, our objects of analysis, independent variables, dependent variables and measures, experimental setup and design, and threats to validity.

### A. Objects of Analysis

The objects of analysis for this study included five Java programs from the SIR infrastructure [37]. Table IV lists the

objects of analysis, as well as, for each object of analysis, data on its associated “Versions” (the number of versions of the object program), “Classes” (the number of class files in the latest version of that program), “Size (KLOCs)” (the number of lines of code in the latest version of the program), and “Test Cases” (the number of test cases available for the latest version of the program). To study the research question, we needed fault data, so we utilized mutation faults provided with the programs [38]. The rightmost column, “Mutation Faults,” is the total number of mutation faults for the program (summed across all versions).

TABLE IV  
EXPERIMENT OBJECTS AND ASSOCIATED DATA

Objects	Versions	Classes	Size (KLOCs)	Test Cases	Mutation Faults
<i>ant</i>	9	914	61.7	877	412
<i>jmeter</i>	6	434	42.2	78	386
<i>xml-sec</i>	4	145	15.9	83	246
<i>nanoxml</i>	6	64	3.1	216	204
<i>galileo</i>	16	68	14.5	912	2494

## B. Variables and Measures

1) *Independent Variable*: Our experiment consists of one independent variable and one dependent variable. The independent variable is the *test case prioritization technique application mapping strategy* which assigns, to a specific sequence of versions,  $S_i, S_{i+1}, \dots, S_j$ , for system  $S$ , specific test case prioritization techniques. There are three strategies used in this paper. Each strategy chooses one of four prioritization techniques (total block coverage, additional block coverage, random order, and original order). Total block coverage sorts test cases by the order of the number of blocks they cover. Additional block coverage selects a test case that yields the greatest block coverage, adjusts the coverage information for the remaining test cases to indicate their coverage for the blocks not yet covered, and then repeated this process until all blocks are covered by at least one test case. Random order is the average of a number of runs (in our experiment 30 runs) with random ordering of test cases. Original order executes the test cases in the order given in the test script provided with the object programs. The three strategies used are as follows:

- AHP: Uses the ART strategy utilizing the AHP method across all versions. This technique is used as the baseline strategy.
- Fuzzy AHP: Uses the ART strategy utilizing the fuzzy AHP method across all versions.
- FESART: A new ART strategy that utilizes a fuzzy expert system to select the best technique across all versions.

2) *Dependent Variable and Measures*: The dependent variable in the study is the *relative cost-benefit value*. To calculate this value, we used the EVOMO economic model [35]. EVOMO<sup>1</sup> involves two equations: one that captures the costs related to the salaries of the engineers who perform regression testing (to translate time spent into monetary values) and one that captures the revenue gains or losses related to changes

<sup>1</sup>Due to space limitations, here we just summarize each equation briefly. See [39], [35] for detailed descriptions.

in the system release time (to translate time-to-release into monetary values). Significantly, the model accounts for costs and benefits across entire system lifetimes, rather than snapshots (i.e. single releases) of those systems, through equations that calculate the costs and benefits *across entire sequences of system releases*. The major cost components that EVOMO captures are as follows: costs for applying regression testing techniques, costs associated with missed faults, costs for artifact analysis, costs of delayed fault detection feedback, and costs associated with obsolete tests.

In addition to the costs already considered in the EVOMO model, we modified the EVOMO model to consider one additional cost: the cost of applying the ART strategy ( $C_{ART}$ ).  $C_{ART}$  is a cost related to human effort, so it is applied in the equations in the same way as other costs related to human effort (by capturing the cost related to the salary of the engineer who performed the activity).

The cost and benefit calculations for the EVOMO model are measured in dollars. To determine the *relative cost-benefit* of the ART strategy,  $S$ , with respect to baseline strategy,  $base$  (the strategy utilizing the AHP method), we use the following equation:

$$(\text{Benefit}_S - \text{Cost}_S) - (\text{Benefit}_{base} - \text{Cost}_{base}) \quad (6)$$

When this equation is applied, positive values indicate that  $S$  is beneficial compared to the  $base$ , and negative values indicate otherwise.

## C. Experiment Setup and Procedure

In order to measure costs such as delayed fault detection, the object programs needed to contain some faults. We used mutation faults and mutant groups created by the *ByteME* (Bytecode Mutation Engine) tool from the SIR Repository [38]. Each mutant group contained, at most, 10 mutants that were randomly selected per version.

To evaluate the fuzzy expert system, two decision makers, each having seven years of industry experience in software development, rated the input criteria described in the previous section for each version of every object program. The decision makers rated each criterion for each prioritization technique in terms of their cost on a scale from 1 to 9, with 9 being considered a very high cost. This input was used in the fuzzy expert system. Then, based on the output provided by the fuzzy expert system, the decision maker determined which technique should be used for every version of each program. Techniques with lower numbers represent a lower cost for using that technique on that particular software version.

Regression testing sessions are often faced with strict deadlines, budgets, or both. Therefore, software companies often need to cut their testing activities short. How much the company has to cut the testing by can change due to the particular software version or a company’s circumstances (e.g., different amount or complexity for a feature update, technical personnel loss, etc.). For this reason, we consider varying the time constraints for each version when we apply regression testing strategies in this experiment. We randomly assigned

the level of time constraints (25%, 50%, or 75%) for each version. These time constraint levels represent situations where time constraints shorten the testing process by 25%, 50%, and 75%.

To implement time constraint levels, we shortened the test execution process for each version by the assigned time constraint level. Further, we ran four sets of random assignments across all versions for each program as shown in Figure 3. For instance, for run 1, we randomly assigned time constraints for each version: 50% for V1, 25% for V2, 75% for V3, and 50% for V4. We repeated this random assignment four times and defined each random assignment across versions as “Run n” (n = 1, 2, 3, and 4). Finally, each decision maker rated the criteria considering these time constraints, and these ratings were used in the fuzzy expert system. Cost-benefit calculations were collected for all strategies and used to determine which strategy was most cost-effective for that particular software version.

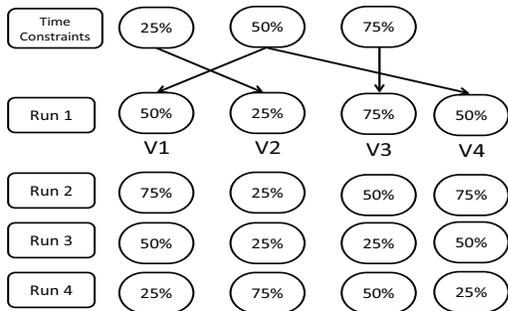


Fig. 3. Random Assignment of Time Constraint Level

#### D. Threats to Validity

This section discusses the construct, internal, and external threats to the validity of our study.

**Construct Validity:** The construct validity could be threatened by the number of criteria considered in this experiment. We considered four criteria, but additional criteria could be considered which could change the results. Also, we developed 67 rules for FESART. A fuzzy expert system with fewer or more rules could be developed and potentially change the results.

**Internal Validity:** The ratings from the decision maker were entered into the fuzzy expert system built in MATLAB. Each of the produced outputs from the expert system were double-checked, but the possibility of small marginal human errors still exists due to the ratings being hand entered into MATLAB.

**External Validity:** The external validity of this experiment could be limited in a couple ways. First, we chose to use three triangular membership functions for the input set and eight triangular membership functions for the output set. Many different numbers of membership functions could be considered, as well as different types (e.g. gaussian, trapezoidal, etc.). We cannot generalize our results because the type and number of membership functions we used are not representative of those

functions. Also, we used two decision makers in this study. The backgrounds and experience levels for the decision makers could differ from those of professional programmers, so we cannot generalize our findings. We tried to reduce this risk by selecting decision makers who have several years of industry experience.

#### V. DATA AND ANALYSIS

In this section, we present the results of our study. Each version for every program is assigned a random time constraint (25%, 50%, or 75%). This procedure is performed four times giving four runs of random time constraint levels for every version for all programs. The cost-benefit results for the four runs are shown in Table V. The AHP-based ART strategy is used as the baseline strategy in the relative cost-benefit calculation, so it is not displayed in the table.

The cost-benefit values for three programs are displayed in a subtable of Table V. The data in the table show the cost-benefit values, in dollars, with respect to the AHP-based ART strategy (baseline) defined in Section IV-B2. Positive cost-benefit values indicate greater cost-benefits than the baseline strategy, and negative values indicate fewer cost-benefits than the baseline strategy. Two decision makers were used in this study and the previous study [7] evaluating the AHP and fuzzy AHP strategies. Results for the first decision maker for the ART strategy utilizing fuzzy AHP are labeled Fuzzy AHP-1, and Fuzzy AHP-2 is used for the second decision maker. Similarly, the results for the first decision maker for FESART are labeled FESART-1, and FESART-2 is used for the second decision maker.

When examining the total cost-benefit values for each version of the program, we found that FESART was more cost-effective than the other ART strategies for all four runs of all five programs. One of the biggest reasons FESART was consistently more cost-effective than the other two strategies was because the cost of applying the ART strategy was much lower. If the strategies picked the same prioritization technique, FESART would be more cost-effective because the cost of applying the strategy was lower than that of the AHP and fuzzy AHP methods. For example, for version 2 of run 1 for *jmeter*, all strategies chose the additional block coverage technique as the most cost-effective technique. However, the costs of applying the AHP and fuzzy AHP strategies were higher, so FESART produced a better result. The AHP strategy took, on average between the two decision makers, 34 minutes, and the fuzzy AHP, averaged between the two decision makers, was 36 minutes. FESART took, on average, half the time of the other two strategies at 13 minutes. When looking at the total cost-benefit values between AHP and fuzzy AHP, fuzzy AHP was frequently more cost-effective than AHP.

To summarize our results and show them visually, we present them in bar graphs by averaging the total cost-benefit values for the four runs in Figure 4. The figure shows the average totals for FESART being largely more cost-effective than the other strategies. In particular, in the case of *galileo*, the differences between FESART and others are more outstanding than other programs. Also, the totals show

TABLE V  
EXPERIMENT RESULTS: RELATIVE COST-BENEFIT IN DOLLARS

<i>ant</i>																
	Run 1				Run 2				Run 3				Run 4			
	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2
v1	52	-5	88	37	-7	-3	30	173	-6	-5	81	36	52	-5	88	37
v2	-4	-2	29	39	-4	-2	29	39	-6	-2	31	39	-4	-2	34	40
v3	104	108	140	149	-47	-2	-13	38	104	108	140	149	-5	-47	29	38
v4	-3	-15	31	25	11	-2	46	39	26	28	60	67	-3	-15	31	25
v5	-3	-5	30	39	-3	-5	30	38	-3	-5	30	39	-3	-5	29	38
v6	-3	0	29	36	-3	0	29	36	-3	0	30	37	-3	0	30	38
v7	-3	-3	30	35	86	-3	71	36	-3	-3	29	34	-3	-3	30	35
v8	-3	2	29	36	-3	2	29	36	173	176	206	211	97	100	130	136
<b>Total</b>	<b>137</b>	<b>80</b>	<b>406</b>	<b>396</b>	<b>30</b>	<b>-15</b>	<b>251</b>	<b>435</b>	<b>282</b>	<b>297</b>	<b>607</b>	<b>612</b>	<b>128</b>	<b>23</b>	<b>401</b>	<b>387</b>
<i>jmeter</i>																
	Run 1				Run 2				Run 3				Run 4			
	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2
v1	29	-3	65	32	-7	42	76	78	29	-3	65	32	-7	-3	31	34
v2	-3	-2	34	36	-3	-2	33	36	-3	-2	34	36	-3	-2	34	36
v3	6	-2	44	34	6	-2	44	34	-5	-2	32	32	-5	-2	32	32
v4	-3	-2	36	130	-3	-2	35	39	-3	-2	36	40	-3	-2	36	40
v5	58	59	93	95	5	7	32	43	5	7	32	43	58	60	93	95
<b>Total</b>	<b>87</b>	<b>50</b>	<b>272</b>	<b>327</b>	<b>-2</b>	<b>43</b>	<b>220</b>	<b>230</b>	<b>23</b>	<b>-2</b>	<b>199</b>	<b>183</b>	<b>40</b>	<b>51</b>	<b>226</b>	<b>237</b>
<i>xml-security</i>																
	Run 1				Run 2				Run 3				Run 4			
	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2
v1	-5	-3	33	36	-5	-3	31	34	-5	-3	33	35	-5	-3	31	34
v2	-3	-2	36	40	-3	-2	36	35	-3	-2	37	41	-3	-2	36	40
v3	-2	0	34	36	-2	0	35	37	-2	0	34	36	-2	0	35	37
<b>Total</b>	<b>-10</b>	<b>-5</b>	<b>103</b>	<b>112</b>	<b>-10</b>	<b>-5</b>	<b>102</b>	<b>106</b>	<b>-10</b>	<b>-5</b>	<b>104</b>	<b>112</b>	<b>-10</b>	<b>-5</b>	<b>102</b>	<b>111</b>
<i>nanoxml</i>																
	Run 1				Run 2				Run 3				Run 4			
	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2
v1	-7	4	42	33	-7	-39	33	34	-6	3	40	40	-7	4	42	33
v2	-3	-2	35	35	-3	-2	34	35	-3	-2	34	35	-3	-2	35	35
v3	-5	-2	32	34	-5	-2	32	34	-5	-2	33	35	-5	-2	33	35
v4	47	49	87	91	47	49	87	91	47	49	87	91	47	49	87	91
v5	-3	-3	33	34	-3	-3	34	35	-3	-3	33	34	-3	-3	34	35
<b>Total</b>	<b>29</b>	<b>46</b>	<b>229</b>	<b>227</b>	<b>29</b>	<b>3</b>	<b>220</b>	<b>229</b>	<b>30</b>	<b>45</b>	<b>227</b>	<b>235</b>	<b>29</b>	<b>46</b>	<b>230</b>	<b>229</b>
<i>galileo</i>																
	Run 1				Run 2				Run 3				Run 4			
	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2	Fuzzy AHP-1	Fuzzy AHP-2	FES-ART-1	FES-ART-2
v1	-7	-5	30	37	-6	-5	-31	36	-7	-5	30	38	-6	-5	-31	36
v2	-4	-2	32	39	-4	-2	33	39	-4	-2	33	39	-4	-2	-35	40
v3	-5	-2	30	38	-5	-2	29	38	-5	-2	30	39	-5	-2	29	38
v4	-3	69	103	110	-3	69	103	110	-3	-2	31	38	-3	-2	-12	38
v5	-3	-3	29	-9	-3	-3	30	38	-3	-3	30	38	-3	-3	-17	39
v6	-3	0	30	38	-3	0	30	38	-3	0	47	54	-3	0	19	37
v7	-3	-3	30	35	-3	-3	29	34	-3	-3	30	36	-3	-3	29	34
v8	-3	-2	29	36	-3	-2	30	37	-3	-2	30	37	-3	-2	30	38
v9	-4	-2	33	39	-4	-2	32	39	-4	-2	34	-55	-4	-2	32	39
v10	-5	-2	30	39	-5	-2	30	39	-5	-2	29	38	-5	-2	30	38
v11	-50	-49	31	38	-120	115	31	38	-120	115	31	38	-120	115	32	155
v12	3	3	36	44	38	-3	71	38	1	1	30	43	38	-3	71	38
v13	135	139	169	176	135	139	169	176	174	177	206	213	174	177	207	214
v14	-3	-1	193	200	-3	90	70	127	-3	-89	116	36	-3	90	70	127
v15	-3	-2	30	38	-3	-2	30	38	-3	-2	29	36	-3	-2	30	37
<b>Total</b>	<b>42</b>	<b>138</b>	<b>835</b>	<b>898</b>	<b>8</b>	<b>387</b>	<b>686</b>	<b>865</b>	<b>9</b>	<b>179</b>	<b>736</b>	<b>668</b>	<b>47</b>	<b>354</b>	<b>484</b>	<b>948</b>

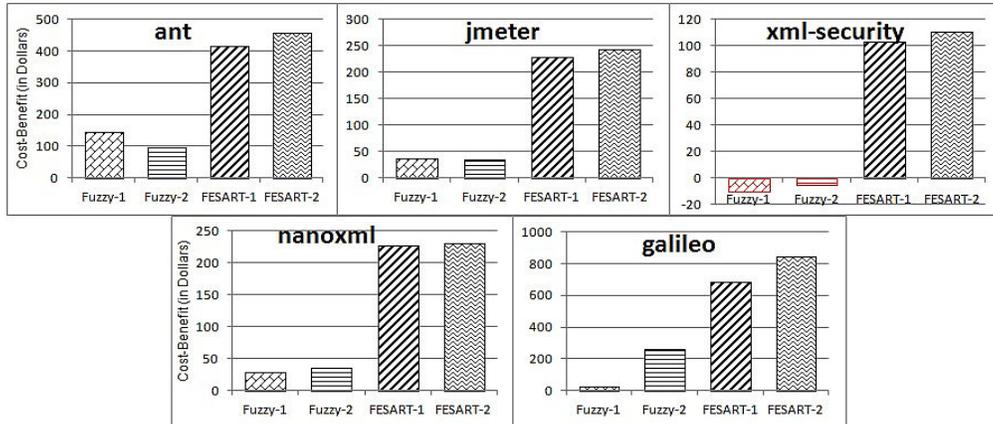


Fig. 4. Average Cost-Benefit Values

fuzzy AHP being more cost-effective than AHP for four of the five programs. For *xml-security*, the average results are negative, which means the strategy is less cost-effective than the baseline. The results for fuzzy AHP are negative for *xml-security* because the techniques chosen were often the same as the techniques chosen by the AHP strategy, and the cost of applying the fuzzy AHP strategy was slightly higher, making it less cost-effective for those cases.

The total cost-benefit values provide a general trend about the data, but one version's cost-benefit value can skew the results. Thus, we further examine the results for individual versions. Examining the results of each version provides more insight about the ART strategies. First, examining it this way shows that FESART is consistently more cost-effective than the other two strategies across all versions for all programs except for a few cases (e.g., version 7 of run 2 for the first decision maker for *ant*). Second, comparing AHP and fuzzy AHP for individual versions, we find that the overall trend is different from what we observed with the total value comparison. Although the fuzzy AHP is frequently more cost-effective than AHP in regards to the total cost-benefit values for programs, there are many versions which are less cost-effective than AHP because the cost of applying the fuzzy AHP strategy is slightly higher than the AHP strategy. We used a tool for the AHP calculations, but no such tool was available for fuzzy AHP. Instead, we wrote code in MATLAB to do the calculations and then manually entered the pairwise comparisons into MATLAB to calculate the results. This process is slightly more time consuming than the tool used for AHP, so when the two strategies choose the same technique, the fuzzy AHP strategy is slightly less cost-effective.

## VI. DISCUSSION AND IMPLICATIONS

We developed FESART to address the limitations of the previously proposed ART strategies utilizing the AHP and fuzzy AHP methods. In this section, we will discuss how FESART effectively addresses these limitations as well as the implications of our results for researchers and practitioners.

*FESART Strategy Results:* Developing a strategy that does not require pairwise comparisons eliminates some of the problems with the previous ART strategies. First, the issue of inconsistent comparisons is eliminated. By not requiring the decision maker to rank the alternatives compared to other alternatives, the risk of inconsistency in the rankings is eliminated. Second, FESART is less time consuming than a strategy requiring pairwise comparisons. The number of rankings needed in FESART is reduced from the number of rankings required by pairwise comparisons, making it less time-consuming for the decision maker. Third, the decreased number of rankings required by the decision maker helps address the issue of scalability. Fewer rankings makes FESART more scalable than the other strategies.

By addressing these limitations, our results indicate that FESART is more cost-effective than the previously proposed ART strategies. One of the biggest contributors to FESART being more cost-effective is the reduction in the amount of

time it takes to apply the strategy. Because the time required by the FESART strategy was less than the other two strategies, if the strategies chose the same technique, FESART was more cost-effective. In addition, in some situations, FESART chose a more cost-effective technique than the other strategies, making the total cost-savings even greater. One possible explanation for FESART choosing a more cost-effective technique than the other strategies is that some of the expert knowledge is placed in the rule base in the fuzzy expert system, so the amount of knowledge required by the decision maker is not as high as it is with the previous strategies.

*Further Understanding About the Implications of the Results:* The findings of our study provide practical implications for practitioners and researchers in software engineering. Our results show that FESART, that considers cost criteria related to testing environments and contexts, improves the cost-effectiveness of that regression testing session.

Savings of hundreds of dollars presented in this study may be unimportant. In practice, however, regression testing could take days or even weeks, so if results such as those presented in this study scale up, savings of the dollar amount may be substantial. For instance, in this study, we used small/medium sized programs, but typical industrial applications have millions of lines of code (e.g., a popular accounting software, Quickbooks, has over 80,000 files and ten million lines of code). Thus, if they were to apply the FESART strategy, the savings would be far greater than those presented in our study.

Further, the costs associated with the defects escaped into the released system could impact the results greatly. This study considered ordinary defects (not severe defects). A survey by Shull et al. [40] suggests that the effort to find and fix severe defects is far more expensive than non-severe defects. Thus, if we take into account different types of defects, our approach could have an even greater impact on cost savings related to early fault detection.

## VII. CONCLUSIONS AND FUTURE WORK

In this paper, we investigated a new adaptive regression testing (ART) strategy by building a fuzzy expert system for ART (FESART). FESART addresses the limitations of other proposed ART strategies. We conducted an empirical study to examine the cost-benefits for three ART strategies (*AHP*, *fuzzy AHP*, and *FESART*). Our empirical study included five Java programs with multiple versions. The results showed that FESART was consistently more cost-effective than the other ART strategies. These results were true when looking at the data by the total cost-benefit values for all versions and the number of versions that were most cost-effective.

In this study, we addressed many limitations of the previous ART strategies. There are still some limitations, mentioned in Section IV-D, which could be addressed. For example, our study considered three triangular membership functions to determine the fuzzy input set. Therefore, for future work, we intend to use other types and number of membership functions (e.g. gaussian or trapezoidal) for FESART. Further, with a different number of membership functions, we plan to

update the rule set to investigate how these changes affect the outcome.

Through the results found in this paper and any additional work addressing the limitations mentioned above, we hope that useful insight is provided to help researchers and practitioners consider testing and environment factors to choose a regression testing technique for a particular software version.

#### Acknowledgements

This work was supported in part by NSF CAREER Award CCF-1149389 to North Dakota State University.

#### REFERENCES

- [1] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 593–617, Sep. 2010.
- [2] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002.
- [3] A. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques," in *Proceedings of the International Conference on Software Maintenance*, Oct. 2002, pp. 204–213.
- [4] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, 2004.
- [5] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker, "Empirical studies of a prediction model for regression test selection," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 248–263, Mar. 2001.
- [6] M. Arafeen and H. Do, "Adaptive regression testing strategy: An empirical study," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2011, pp. 130–139.
- [7] A. Schwartz and H. Do, "A Fuzzy AHP Approach to Improve Adaptive Regression Testing Strategies," North Dakota State University, Tech. Rep. NDSU-CS-TR-13-001, 2013.
- [8] T. L. Saaty, *The Analytic Hierarchy Process*. McGraw-Hill, 1980.
- [9] J. Benítez, X. Delgado-Galván, J. Gutiérrez, and J. Izquierdo, "Balancing consistency and expert judgment in ahp," *Mathematical and Computer Modelling*, vol. 54, no. 7, pp. 1785–1790, 2011.
- [10] L. Lin and C. Wang, "On consistency and ranking of alternatives in uncertain ahp," *Natural Science*, vol. 4, no. 5, pp. 340–348, 2012.
- [11] S. Hatton, "Early prioritisation of goals book series," *Advances in Conceptual Modeling Foundations and Applications*, vol. 4802, no. 235–244, pp. 517–526, 2007.
- [12] O. López-Ortega and M.-A. Rosales, "An agent-oriented decision support system combining fuzzy clustering and the ahp," *Expert Systems with Applications*, vol. 38, no. 7, pp. 8275–8284, 2011.
- [13] T. L. Saaty and M. S. Ozdemir, "Why the magic number seven plus or minus two," *Mathematical and Computer Modelling*, vol. 38, no. 3, pp. 233–244, 2003.
- [14] A. Adeli and M. Neshat, "A fuzzy expert system for heart disease diagnosis," in *Proceedings of International Multi Conference of Engineers and Computer Scientists, Hong Kong*, vol. 1, 2010.
- [15] M. A. Kadhim, M. A. Alam, and H. Kaur, "Design and Implementation of Fuzzy Expert System for Back pain Diagnosis," *International Journal of Innovative Technology & Creative Engineering (IJITCE)*, no. 9, pp. 16–22, 2011.
- [16] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *Software Testing, Verification, and Reliability*, pp. 67–120, Mar. 2010.
- [17] R. Aull-Hyde and K. A. Davis, "Military applications of the analytic hierarchy process," *International Journal of Multicriteria Decision Making*, vol. 2, no. 3, pp. 267–281, 2012.
- [18] N. Subramanian and R. Ramanathan, "A review of applications of analytic hierarchy process in operations management," *International Journal of Production Economics*, vol. 138, no. 2, 2012.
- [19] Y. Zhang, X. Zhang, X. Zhao, and T. Zhang, "Early effort estimation by ahp: A case study of project metrics in small organizations," in *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 1. IEEE, 2012, pp. 452–456.
- [20] A. Perini, F. Ricca, and A. Susi, "Tool-supported requirements prioritization: Comparing the AHP and CBRank methods," *Information and Software Technology*, vol. 51, pp. 1021–1032, Jun. 2009.
- [21] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 201–212.
- [22] C. Shen, M. Cheng, C. Chen, F. Tsai, and Y. Cheng, "A fuzzy AHP-based fault diagnosis for semiconductor lithography process," *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 2, pp. 805–815, Feb. 2011.
- [23] C.-C. Sun, "A performance evaluation model by integrating fuzzy ahp and fuzzy topsis methods," *Expert systems with applications*, vol. 37, no. 12, pp. 7745–7754, 2010.
- [24] V. Ahl, "An experimental comparison of five prioritization methods," *Master's Thesis, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden*, 2005.
- [25] S. Lee, "Using fuzzy AHP to develop intellectual capital evaluation model for assessing their performance contribution in a university," *Expert Systems with Applications*, vol. 37, no. 7, pp. 4941–4947, 2010.
- [26] M. Fasanghari and G. A. Montazer, "Design and implementation of fuzzy expert system for tehran stock exchange portfolio recommendation," *Expert Systems with Applications*, vol. 37, no. 9, pp. 6138–6147, 2010.
- [27] S. Kad and V. Chopra, "Fuzzy logic based framework for software development effort estimation," *Research Cell: An International Journal of Engineering Sciences*, vol. 1, pp. 330–342, 2011.
- [28] M. Kazemifard, A. Zaeri, N. Ghasem-Aghaee, M. Nematbakhsh, and F. Mardukhi, "Fuzzy emotional cocomo ii software cost estimation (fecscce) using multi-agent systems," *Applied Soft Computing*, vol. 11, no. 2, pp. 2260–2270, 2011.
- [29] Z. Xu, K. Gao, and T. M. Khoshgoftaar, "Application of fuzzy expert system in test case selection for system regression test," in *Proceedings of International Conference on Information Reuse and Integration*. IEEE, 2005, pp. 120–125.
- [30] L. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.
- [31] E. H. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International journal of man-machine studies*, vol. 7, no. 1, pp. 1–13, 1975.
- [32] T. Takagi and M. Sugeno, "Fuzzy identification of systems and its applications to modeling and control," *IEEE Transactions on Systems, Man and Cybernetics*, no. 1, pp. 116–132, 1985.
- [33] P. C. Shill, K. K. Pal, M. F. Amin, and K. Murase, "Genetic algorithm based fully automated and adaptive fuzzy logic controller," in *IEEE International Conference on Fuzzy Systems*. IEEE, 2011, pp. 1572–1579.
- [34] P. R. Srivastava, S. Kumar, A. Singh, and G. Raghurama, "Software testing effort: An assessment through fuzzy criteria approach," *Journal of Uncertain Systems*, vol. 5, no. 3, pp. 183–201, 2011.
- [35] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2008, pp. 51–62.
- [36] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 329–338.
- [37] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *International Journal on Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.
- [38] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, Sep. 2006.
- [39] —, "An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2006, pp. 141–151.
- [40] Shull, F. et al., "What we have learned about fighting defects," in *Int'l. Softw. Metrics Symp.*, 2002, pp. 249–258.

# Identifying Process Improvement Targets in Test Processes: A Case Study

Tanja Toroi, Anu Raninen and Lauri Väätäinen

School of Computing  
University of Eastern Finland  
Kuopio, Finland  
tanja.toroi@uef.fi

**Abstract** In this paper, we present the results of case studies conducted in three software companies in order to identify the improvement targets in their test processes. The test processes were modeled using the LAPPI (A Light-weight Technique to Practical Process Modeling and Improvement Target Identification) technique. Identifying improvement targets quickly and in a cost-effective way is important to maximize the value of SPI (Software Process Improvement) initiatives. The most significant problems identified in this study were the low level of unit testing, unknown repercussions of code changes, and missing exit criteria for testing. These findings support the results of earlier studies. Further, a less reported problem was identified. It appears that software companies find it difficult to integrate test automation with their manual test processes. Making the recurring test related problems known among the software industry helps to justify the need for test process improvement and standardization of test processes.

**Keywords-** test process improvement, process modeling, case study

## I. INTRODUCTION

High quality software is a key factor if companies want to remain competitive in software markets [1]. One of the most important factors in software quality assurance is efficient software testing [2]. Thus, continuous improvement of the software test process is important. Widely used software process improvement approaches CMMI [3] and ISO/IEC 15504 [4] define software test processes on a general level. CMMI defines two process areas related to testing; Verification and Validation. ISO/IEC 15504 standard defines Software testing and System testing life cycle processes. However, especially smaller software companies consider CMMI and ISO/IEC 15504 too heavy-weight for them to adopt [5-6].

In addition to the general software process improvement approaches, there are several test process improvement models available, for example TPI Next (Business Driven Test Process Improvement) and TMMi (Test Maturity Model Integration) [7-8]. However, our experience working with several software companies has shown that they consider the test improvement models as too exhaustive and difficult to apply in practice. Consequently, the models are not widely used in software test process improvement [9].

In this paper, we present how test process improvement can be initiated in a light-weight way. We conducted case studies [10] in three software companies. The aim of the

studies was to identify test process improvement targets. The test processes were modeled using the LAPPI (A Light-weight Technique to Practical Process Modeling and Improvement Target Identification) technique [11]. LAPPI is a lightweight technique that can be applied to make a process and its problem points visible using only a modest amount of resources.

The results of the study reveal that the most significant test related problem areas are the low level of unit testing, unknown repercussions of code changes, and missing exit criteria for testing. These problems were identified in all three case organizations and are similar to those presented in previous studies [9, 12-16].

In addition, we found a less reported problem; software companies seem to have difficulties with integrating test automation with the manual test process. There was no clear place for the automated testing phase and its contents were vague. Based on visualizing the problem points of the test processes using LAPPI, we gave improvement suggestions in order to help the companies avoid the identified problems in the future. In addition, making the recurring test related problems known among the software industry helps to justify the need for test process improvement and standardization of test processes.

The overall structure of this paper is: Section II presents the related work on the subject. In section III, the research setting, case organizations and the LAPPI technique are presented. The problems found and proposed improvement suggestions are analyzed in section IV and discussed in section V. Section VI provides the conclusion.

## II. RELATED WORK

Previous research with test practices shows a number of problems with testing [9, 12-16]. In this section, we present the newest studies discussing the test process problems.

The results of a survey presented in [9] show that test process improvement is considered to be very important in order to ensure good quality software products. However, the maturity of the test processes in software companies is still surprisingly low [9]. The most troublesome issues hindering software testing are lack of resources, especially time and testing knowledge, lack of documentation, such as proper test case specifications, proper testing methods, insufficient test environments, and rush to deliver incomplete applications to the customers [9].

In their study Ahonen et al. [12] have noticed that test planning, test cases, and unit testing are often vague and neglected. The results of the study suggest that the best improvements may not be achieved by employing very complicated technology for software process support but by investing in proper management of people.

In [13] Taipale et al. show that the actual total effort spent on testing (28.9%) was noticeably smaller than often mentioned in the literature (50-60%). The respondents of the study emphasized that the lack of testing know-how and insufficient testing tools hinder testing development. Taipale et al. suggest that efficient communication and interaction between development and testing should be enhanced and early involvement of testing and planning of testing should be invested in.

The results of a survey by Ng et al. [14] reveal that the lack of testing expertise is the dominant factor preventing organizations from using software testing methodologies. In addition, over one third of organizations have ad hoc testing methods. Further, automated tools, metrics and standards are regarded as time wasting.

Torkar and Mankefors [15] found out that verification and validation are the first processes to be neglected in when time runs out. Their results also reveal that developers need a better knowledge on the importance on unit testing. Further, the developers need a tool that can indicate how good their test cases really are, i.e. tool for test coverage.

The results of the survey by Runeson [16] suggest that it is hard to create a relevant test environment, and to find the people with the right knowledge and skills. In addition, he found out that there are problems with test documentation.

It can be seen as surprising that the problems found in the above-mentioned studies are similar although the companies are very different between themselves e.g. their size and business domain vary a lot, and they work on both, national and international markets. The studies are compared in table I.

TABLE I.

Study	Case orgs.	Size	Main problems found
This paper	3	SC, SME, Large	The low level of unit testing, unknown repercussions of code changes, no exit criteria for testing, difficulty to integrate automated testing with manual test process
[9]	14	SC, SME, Large	Lack of resources and proper test case specifications, lack of knowledge on testing, insufficient test environments, rush to deliver incomplete applications to the customers
[12]	1	Large	Test planning, test cases, and unit testing are often vague and neglected
[13]	30	SME,	Total effort spent on testing

	org. units	Large	smaller than often mentioned, lack of testing know-how, insufficient testing tools
[14]	65	SC, SME, Large	Lack of testing expertise, ad hoc testing methods, automated tools, metrics and standards are regarded as time wasting
[15]	3	SC, SME, Large	V&V are the first processes to be neglected in when time runs out, developers need a better knowledge on the importance on unit testing, test coverage
[16]	12	SC, SME, Large	Relevant test environment, how to find people with the right knowledge and skills, test documentation

SC = small company, less than 50 employees; SME = Small and medium sized enterprise, less than 250 employees; Large = over 250 employees [17]

### III. RESEARCH SETTING

In this section we present the research problem, the research method, the case organizations, and how the LAPPI process modeling technique was applied in this study. The research method used was case study [10]. The case organizations, whose test processes were the target of the process modeling efforts present different company sizes; one small company, one SME and one large company. The LAPPI process modeling technique [11] is a technique where process modeling is complemented with role and information flow modeling in order to better understand the process at hand.

#### A. Research Problem and Method

Software testing is important in order to reduce software defects and to ensure software quality [18]. The cost of software defects can vary from nothing at all to large amounts of money and even loss of life [1]. Testing identifies defects removal of which increases the software quality by increasing the software's potential reliability [19]. However, software companies often have trouble with implementing testing [20] and based on the results of this and previous studies [9, 12-16] it appears that test processes are immature.

Software process improvement is one way of making test processes more efficient and mature. For example, software process improvement reference models CMMI [3] and ISO/IEC 15504 [4] offer guidance in what companies should take into account while improving the maturity of their test processes. In addition, there are test improvement approaches developed, for example TMMi [8] and TPI Next [7]. These approaches, as well as the more general software process improvement approaches, are proven to be efficient [21-24].

However, identifying the software process problem areas can be time-consuming and problematic. For example, small companies tend to have problems with process assessments

because of their often heavy resource requirements to produce meaningful results [25]. However, the first results in SPI (Software Process Improvement) should be made visible in a short period of time so that the employees remain motivated [26, 27]. When applying TMMi [8] or TPI Next [7] it can take a long time to visualize problems and thoroughly understand them in a company.

It appears that there is a lack of a lightweight and cost-effective way to identify test process problems areas. Hence, the research problems of the study are:

- How to identify software test process problem areas in a cost effective way?
- What are the major problem areas in software testing?

Case study [10] was used as a research method in order to answer the research problems. The case studies were conducted in three case organizations. SPI approach via process modeling was chosen to be applied. The LAPPI technique [11] was used to model the test process and identify its problem points. The LAPPI technique is presented in [11]. In section C, we describe how the technique was applied in our three case studies.

### B. The Case Organizations

There are three case organizations in the study; one small, one SME-sized and one large software company. Company A has 9 employees in development and maintenance and 4-6 people in testing. Company B has 24 employees in development and maintenance and 2 employees in system testing. Company C has about 30 employees in development and maintenance and one person in system testing. Company A uses an open source, web-based defect tracking tool, Mantis for reporting the defects as well as managing their overall software development. Company B uses Jira and company C HP Quality Center to support their test activities. The characteristics of the companies are presented in table II.

TABLE II. THE CASE ORGANIZATIONS

	Company A	Company B	Company C
<b>Market</b>	Farming	Metal industry	Telecom
<b>Size</b>	Small	Large	Medium
<b>Employees in develop. / system testing</b>	9 / 4-6	24 / 2	30 / 1
<b>Country</b>	Finland	Multi-national	Finland
<b>Defect tracking system</b>	Mantis	Jira	HP Quality Center
<b>Age of the company</b>	24	43	13

### C. Applying the LAPPI Process Modeling Technique

LAPPI (A Light-weight Technique to Practical Process Modeling and Improvement Target Identification) is a lightweight technique to practical process modeling and improvement target identification [11]. The LAPPI technique enables discovering the problem points with minimum disturbances to the actual software development.

Mendling et al. [28] have criticized that many process modeling techniques are too abstract to be applied by novices and non-experts in practice. The users of LAPPI do not have to have any knowledge of exact modeling notation and the former experience of modeling is not required [11]. Thus, the LAPPI technique is very useful especially in companies where separate modeling specialists do not exist. When the processes and their problem points are made visible it is relatively easy to give improvement suggestions and proceed with the improvement project.

Two modeling sessions are the key steps of the LAPPI technique. In modeling session I the participating roles of the process and information flows between them are modeled. In modeling session II the process itself is made visible. Identifying the problematic information flows among the process help to identify the overall problem points of the process [29]. A wall-chart technique is used in both of the sessions.

As a result of applying the LAPPI technique, the case organizations received an electronic process overview which includes the process descriptions, the problem points of the process and improvement suggestions.

Modeling the test processes of the case organizations was carried out in collaboration between the researchers and the employees of the organizations. In addition to two researchers, 2-6 representatives of all the case organizations participated in the modeling sessions. The participants represented different roles in the case organizations (e.g. tester, advisor, development manager).

Table III presents the hours that modeling required in case organizations. Applying LAPPI showed out to be a practical and efficient solution for identifying problems in the test processes.

As it can be seen from the table III, it took 133 (74+59) man-hours in company A, 83 (53+30) man-hours in company B, and 89 (32+57) man-hours in company C to provide the companies with a process overview and practical improvement suggestions. All this required merely 74 man-hours of company A's time, 53 man-hours of company B's time, and 32 man-hours of company C's time. The modeling took more man-hours in company A because more participants participated in the modeling sessions.

The LAPPI [11] technique is practical and requires only a modest amount of resources to visualize process problems. There is no reason why TMMi [8] or TPI [7] or any other improvement approach could not be applied to support process improvement after identifying the problem areas using LAPPI.

TABLE III. RESOURCES OF APPLYING LAPPI IN CASE ORGANIZATIONS

	A	RA	B	RB	C	RC
<b>Time spent (man-hours)</b>	74	59	53	30	32	57
<b>Total (man-hours)</b>	<b>133</b>		<b>83</b>		<b>89</b>	

A, B, C = Time spent in company A, B, or C in man-hours  
 RA, RB, RC = Time spent in company A, B, or C by researchers in man-hours

Total = Total time spent in company in man-hours

IV. RESULTS

In this section we present the results of the process modeling conducted applying the LAPPI technique. Section A presents problems discovered. In section B possible solutions to those problems are suggested.

A. Problems in the Test Processes of the Case Organizations

In this section we present the problems discovered during the modeling sessions I and II. Problems and the improvement suggestions related to them are summarized in table IV, where “A”, “B” and “C” show in which company the problem was identified. “I” and “P” show in which modeling session the problem was identified (I = Information flow modeling, P = Process modeling).

TABLE IV. SUMMARY OF THE PROBLEMS AND THEIR SOLUTIONS

Problem	Improvement suggestions (see section B)	A	B	C
The level of unit testing by programmers varies a lot. (I)	- Define minimum level of unit testing. - Specify test cases. - Provide training in testing.	x	x	x
All repercussions of code changes are not known. (I)	-Streamline the usage of defect tracking system. - Specify test cases.	x	x	x
No exit criteria defined for testing. (P)	- Define exit criteria for testing.	x	x	x
Lack of proper test case specification. (I)	- Consider software testing already in the requirement specification phase. - Specify test cases.	x	x	
Test automation not integrated with manual test process. (P)	- Integrate test automation with the manual test process.		x	x
Documentation processes are vague. (I)	- Improve documentation.	x		x
Customers’ defect reports are often poorly documented. (I)	- Streamline the usage of defect tracking system. - Improve the defect descriptions.	x		x
Resource	- Provide more test		x	x

problems in system testing (P)	resources for system testing on busy times.			
Areas of responsibility are not clear. (I)	- Define the roles and responsibilities.		x	x
Estimation of defect priorities in system testing (P)	- Programmers re-estimate defect priorities.			x
Short cuts are taken in the process. (P)	-Improve communication and enforce following the process.	x		x
Slow pace of the test process. (I)	- Improve communication and enforce following the process. - Provide more test resources. - Provide training in testing.	x		
The amount of defects detected by the customers is unknown. (P)	- Streamline the usage of defect tracking system.	x		
Interface testing is problematic. (P)	To be worked out. The solution depends on a third party.	x		x

- The level of unit testing by programmers varies a lot** in all the companies. There is no defined level of unit testing. Some programmers implement unit testing while others do not test their code at all. The testers can receive code with very basic defects that should have been detected or, in worst case, program code that can’t even be compiled. This became apparent based on problems with information flows between programmers and testers.
- All repercussions of code changes are not known.** The tester can not necessarily know what the programmer has modified and what needs to be tested due to the modifications. This problem was identified in all the companies and it became apparent based on problems with information flows between programmers and testers. Further, it can sometimes be the case that the tester knows what has been modified but he/she doesn’t know which part of software/the component the modification will or might influence indirectly.
- No exit criteria defined for testing.** The development schedule dictates the end of testing phase and no exit criteria are defined. It can happen that testing is exited even if it is not yet finished. This problem was identified in process modeling in all the companies.
- The lack of proper test case specifications** was recognized as one of the biggest problems in companies A and B. Test cases are not described properly and there is no proper process designed for doing this. As a consequence, the testers do not always know what should

- be tested. This became apparent based on problems with information flows between designers, programmers, and testers. In addition, updating and maintaining test data and auxiliary test material has been problematic.
5. **Test automation is not integrated with the manual test process** in companies B and C. In company B it is unknown who defines the scripts and who is responsible for running them. In addition, it is unclear when the automated scripts should be run. In company C system testing is partly automated but unit testing does not utilize test automation possibilities. These problems became apparent in process modeling.
  6. **Vague documentation processes** in companies A and C. For example, programmers often don't have enough time to document the changes they have made to program code. This became apparent based on problems with information flows between programmers and testers. As a consequence, people in other roles have to partially guess what has been changed and transmit the information to the tester and eventually the customer. As an example of a problematic document type, release notes are often inadequate in company C. Further, **defect reports from customer are often documented in an unsatisfying level**. The testers find it often difficult to repeat the defect according to the description. This became apparent based on problems with information flows between customers, project managers, and testers.
  7. **Problems in system testing**. Company C has **problems with providing necessary resources for system testing**. There is only one full time tester who implements system testing. In addition, **the areas of responsibilities are not clear** in companies B and C. In system testing, tester does not know who solves vague cases and to whom they must be assigned. Further, in company B, the defects are reported to testing in quite a diffuse way. Almost everyone can report them to everybody else and defects are not necessarily entered to the defect tracking system. Problems in system testing became apparent in process modeling when tester's work in the test process was modeled.
  8. **Estimation of defect priorities** is sometimes difficult for the person implementing system testing. Underestimating the seriousness of a defect has caused problems in company C. When a priority is set low by the system tester the programmers might think that the defect is not worth fixing before the next software release. As a result, the customer receives a software version including this defect and makes his/her own (more accurate) estimation of the level of seriousness of the defect at hand and insists on an immediate fix. Consequently the company has had to make resource consuming extra software releases. The problem of defect priority estimation became apparent in process modeling when tester's work phases in the test process were modeled.
  9. **Short cuts are taken in the process** in companies A and C. Sometimes the defined processes are not followed. This became apparent in process modeling. This can cause problems in version management and cause the old defects to re-emerge. For example, defects are sometimes fixed only to the software version installed to the customer environment. Hence, the fix is absent from the development version and due to this the defect may reappear in the customer's environment when the next version of software is installed there.
  10. **Slow pace of the test process** in company A. When the programmer has fixed a defect and altered the status of the issue in defect tracking system to *solved* the tester can start testing the modified code in question. The programmer continues with his/her other tasks. Sometimes it can take a long time to initiate testing. If new defects are found from the modified software part in question, the programmer cannot necessarily remember what he/she has changed earlier. In addition, new modifications might have influenced or totally removed the old problems. In this case finding the erroneous code can take an excessively long time. Further, sometimes it can take a too long time to get feedback from the external validators and beta testers. This became apparent based on problems with information flows between programmers and testers.
  11. **The amount of defects detected by the customer is unknown** in company A. This problem shouldn't exist since the defect tracking system of company A includes a data field where the person reporting a defect should choose whether the defect was detected inside the company (by testers) or by the customer. However, default value of this field is 'the company'. On a busy day, when a customer calls and reports a defect, the defect detector is often forgotten as the default value. Thus, this important measure of software quality cannot be trusted. This became apparent in process modeling when advisors (help desk) work in the test process was modeled.
  12. **Interface testing is problematic**. This problem was identified in process modeling in companies A and C. For example, the companies have interfaces to bank's systems. Every bank has its own way to respond to the interface request. However, some of the banks have not provided the test bed service for testing the interface. Thus, the response cannot be tested beforehand because the other end of the interface does not exist.
- B. Analysis and Improvement Suggestions*
- The majority of the problems in the test processes can be solved by rationalizing the software test processes and the usage of defect databases. In addition, good project management principles and proper instructions and documentation are needed.

In this section we provide improvement suggestions in order to avoid the before mentioned problems in the future. The improvement suggestions can be utilized by all software companies struggling with similar problems. Most of the improvement suggestions provided support the lessons learned in software testing, reported in [30]. Further, most of the improvement suggestions can also be found from standards and process models, see e.g. [3-4, 7-8, 31]. The problems and improvement suggestions are linked in table IV.

1. **Define minimum level of unit testing.** The minimum level of unit testing implemented by the programmers before committing modified code to the software versioning system should be defined. In addition, it should be defined how much system testing is enough.
2. **Pay attention to test case specification.** A standard format of test cases makes test cases reusable and self-sufficient. Test cases should be specified in a commonly agreed way. In addition, all the testers often need the same test data and test materials. Thus, a shared test data should be created and a person in charge nominated.
3. **Provide training in software testing.** With specific test method skills and competencies the tester makes the test process more predictable and manageable. Programmers should be trained in unit testing skills. Testers need training in system testing and usability testing. In our case, the external testers would need guidance on how to use the defect tracking system of company A efficiently.
4. **Streamline the usage of defect tracking system.** Defects should be analyzed to learn from past mistakes and avoid them in the future. At least the amount of defects detected by the customer(s) should be known and monitored. In company A, the value of the defect detector in the defect tracking system must be obligatory to assign and empty as default. At minimum, there should be values “the customer”, “the company”, and “external tester” to choose from.
5. **Define exit criteria for testing.** Define a set of conditions (e.g. all the planned requirements must be met, all the high priority bugs should be closed, all the test cases should be executed) that should be met in order to close testing phase. Exit criteria define when testing has been conducted well enough, i.e. when test coverage is sufficient.
6. **Consider software testing already in the requirement specification phase.** Considering testing from the beginning of the project helps to ensure and even improve the product quality. Test requirements must be taken into account already when new software features are specified. Thus, preliminary test cases should be created while creating new requirements. Further, a tester should be involved in project planning and design phases.
7. **Integrate test automation with the manual test process.** Ensure that test automation is integrated with

the manual test process. Responsibilities must be defined and person in charge nominated. In addition, schedule of the automated tests must be defined together with the manual testers.

8. **Improve and monitor documentation.** Ensure that the test documentation guidelines are unambiguous and adequate. Further, ensure that the guidelines are followed. In addition, the descriptions of a defect must be paid attention to. The description must be written on an exact enough level that the tester can repeat the defect.
9. **Provide more test resources.** Allocate extra time and resources in testing on busy times (e.g. two weeks before release). In addition, more time should be allocated for documentation purposes.
10. **Define the roles and responsibilities related to testing.** It should be ensured that someone owns the test process. In addition, it must be checked that everyone knows who is responsible for a certain task.
11. **Conduct re-estimation of defect priorities by programmers.** When the defects have been assigned to the programmer for the correction, the programmer should check that the defect priority has been correctly estimated in system testing. In addition, the priorities could be discussed in project meetings.
12. **Improve communication and enable and enforce following the processes.** Communication of information to all stakeholders makes it possible to make the appropriate decisions related to testing. In our case organizations, internal communication should be improved. For example, there should be a way of informing all the stakeholders about version updates, e.g. email. Further, it should be ensured that all the stakeholders are trained to follow the defined test process. Following of the process should also be enforced and supervised in order to stay in check. Also, the test process must be streamlined and kept up-to-date via regular reviews.

## V. DISCUSSION

It would appear that software testing is at a turning point. For example, at the moment new software testing standards are being developed on two fronts. There is the relatively new software testing standard 29119 [31], aim of which is to provide one definitive standard for software testing that defines vocabulary, processes, documentation, and techniques for software testing that can be used within any software development life cycle. In addition, there is the emerging process assessment model for software testing; ISO/IEC CD 33063 [32] that uses ISO/IEC 29119-2 as a basis for test process assessment.

However, it appears that, despite the broad standardization effort, the implementation of test processes in software companies is still fairly immature. Based on the benchmark results [33], 84% of the test organizations

assessed are still at TMMi maturity level 1, and a mere 10% are at TMMi maturity level 2. Hence, at this point it is important to identify the basic problems of the test processes to be able to truly improve the maturity of testing. In addition, making the recurring test related problems known among the software industry helps to justify the need for test process improvement and standardization of test processes.

In this study we have identified several test process related problems in three software companies. The most significant problems were the low level of unit testing, unknown repercussions of code changes, and missing exit criteria for testing. Similar problems have been reported in earlier studies [9] and [12-16].

In addition to problems that are similar to those presented in earlier studies, we also identified less reported problems. We noticed that, automated testing appears as a separate phase apart from the actual software test process. This was the case in two of our case organizations. For example, the scheduling of the automated tests, responsibilities and the maintenance of the test automation were not defined as part of the test process. It was not carefully thought who runs the automated scripts, when they are run, and how the scripts are maintained. In addition, there were no clear exit criteria defined for automated testing. It was not decided what kind of result would be acceptable. This notion, in its part, shows the relative immaturity of test processes.

Together with the study in this paper, the referenced studies, [9] and [12-16], describe the test related problems of over one hundred case organizations. It can be seen as surprising that the problems are similar although the companies are very different between themselves.

In this paper, we suggest process modeling as means to get the test process improvement projects going. We have applied the LAPPI process modeling technique [11] to identify the process problems. A light-weight way of making the problems of test processes visible might help to underline the importance of test process improvement. Seeing the concrete problems might convince the companies to exploit the test related standards and test improvement approaches to support the process improvement.

As reported also in earlier studies (see [11] for a list), we found that the LAPPI technique is a simple and cost-effective tool to identify process problems with small amount of resources. At the minimum, it took only 83 man-hours to provide a case organization with a process overview and identified process problems. All this required merely 53 man-hours of the case organization's time. Additional 30 hours were spent by the researchers in developing the process overview and other documentation related to the LAPPI technique, see [11] for details. In addition, improvement suggestions were easy to provide based on the detailed modeling results.

In addition to listing the problems identified in our case organizations, we provide improvement suggestions for the companies. These improvement suggestions are general enough to be utilized in other software companies in order to improve their testing processes. The problem areas identified in this paper help especially companies who have no previous experience in test process improvement. The

companies can start test process improvement by reflecting whether these common problems might be their problem spots also.

We are currently implementing the improvements in our case organizations. It would appear that via implementing the proposed improvement suggestions many important improvements can be achieved. For example, agreed level of unit testing makes testers work more meaningful. Testers can concentrate on finding more challenging defects when knowing that most of the simple defects have been removed at the unit testing phase. Another example of the implementation of the improvements is that the case organizations where automated testing integration problems were detected are now defining their test processes on a more detailed level and integrating test automation with their manual test processes. In addition, exit criteria for testing will be defined.

In order to be able to identify the impact of the improvements, measurement is needed. However, measuring the success of process improvement efforts is quite problematic [34]. At the moment, in the case organizations the quality of the test process and its improvement can be measured through the amount of defects and defect density. These measures alone are not enough. This has now been noted and in future, more attention will be paid to measurement. For example, more accurate data will be gathered in order to be able to utilize more metrics. The metrics should be aligned with the company's goals and with the things that are important to the company [35]. For example, the amount of specified test cases, the amount of executed test cases, defects found by customers, and defects slipped through earlier test phases [36] would be useful metrics in our case organizations.

## VI. CONCLUSION

Software companies find it often problematic to initiate process improvement projects. It appears that there is a lack of a lightweight way for identifying test process problem areas. Hence, the objective of the study was to find out how to identify software test process problem areas in a cost effective way and find out what the most troublesome issues affecting software testing are.

To answer the research questions we conducted case studies in three case organizations. We applied the LAPPI (A Light-weight Technique to Practical Process Modeling and Improvement Target Identification) process modeling technique to make the test processes visible and to identify their problem points. We found that the most significant problems were the low level of unit testing, unknown repercussions of code changes, missing exit criteria for testing, and difficulty to integrate test automation with the manual test process. After visualizing the processes we provided improvement suggestions to the problems found. The improvement suggestions are general enough for other software companies struggling with the similar problems to make use of. The identified problems appear to have been the problem points of testing for a long period of time. The recurring test related problems show the need for test process improvement and standardization of test processes.

## Acknowledgements

This research was funded by the Finnish Funding Agency for Technology and Innovation (Tekes) with grant 70030/10 for METRI (Metrics Based Failure Prevention in Software Engineering) project.

## References

1. S. Slaughter, D. Harter, and M. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, 8, pp. 67-73, 1998.
2. J. Gao, H.-S. Tsao, and Y. Wu, *Testing and quality assurance for component-based software*, Artech House Inc, Norwood, MA, 2003.
3. CMMI for Development, Version 1.3: CMMI-DEV, V1.3, *Improving processes for developing better products and services*, 2011
4. ISO/IEC 15504, 2003. *Information Technology, Process Assessment, Parts 1-5*, 2003.
5. M. Staples, M. Niazi, R. Jeffery, A. Abrahams, P. Byatt, and R. Murphy, "An exploratory study of why organizations do not adopt CMMI," *Journal of Systems and Software*, vol. 80, 6, pp. 883-895 2007.
6. ISO/IEC 29110 Software engineering – Lifecycle profiles for very small entities (VSEs), 2011.
7. Sogeti, TPI Next, *business driven test process improvement*, UTN Publishers, The Netherlands, 2009.
8. TMMi Foundation. *Test Maturity Model Integration (TMMi)* <http://www.tmmifoundation.org/html/tmmiref.html>
9. T. Toroi, "Improving software test processes," *Proc. New Trends in Software Methodologies, Tools and Techniques*, IOS Press, Amsterdam, 2006.
10. R.K. Yin, *Case study research: Design and methods*, Sage publications, INC, 2009.
11. A. Raninen, J.J. Ahonen, H.-M. Sihvonen, P. Savolainen and S. Beecham, "LAPPI: A light-weight technique to practical process modeling and improvement target identification," *J. Softw. Evol. and Proc.* 2012, DOI: 10.1002/smr.1571
12. J.J. Ahonen, and T. Junttila, "A case study on quality-affecting problems in software engineering projects," *Proc. IEEE International Conference on Software – Science, Technology & Engineering*, IEEE 2003.
13. O. Taipale, K. Smolander, and H. Kälviäinen, "A survey on software testing," *Proc. the 6th International SPICE Conference on Process Assessment and Improvement*, pp. 69-85, SPICE User Group, 2006.
14. S.P. Ng, T. Mumane, K. Reed, D. Grant, and T.Y. Chen, "A preliminary survey on software testing practices in Australia," *Proc. IEEE 2004 Australian Software Engineering Conference*, pp. 116-125, 2004.
15. R. Torkar, and S. Mankefors, "A survey on testing and reuse," *Proc. IEEE International Conference on Software – Science, Technology & Engineering*, pp. 164-174, 2003.
16. P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, 4, pp. 22-29, 2006.
17. The new SME definition, User guide and model declaration, EU, [http://ec.europa.eu/enterprise/policies/sme/files/sme\\_definition/sme\\_user\\_guide\\_en.pdf](http://ec.europa.eu/enterprise/policies/sme/files/sme_definition/sme_user_guide_en.pdf) (2005)
18. R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005.
19. GlobalWorldTech, *Why software testing is necessary?* <http://qasoftwaretestingindia.blogspot.com/search/label/Software%20Testing>, 2009.
20. J.A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, 1, pp. 70-79, 2000.
21. T. Koomen, "Worldwide survey on test process improvement," *Technical Report*, Sogeti Netherland, 2002.
22. J. Zyl, "Software testing in a small company," *Test Focus*, vol. 12, 1, pp. 4-8, 2011.
23. D. Goldenson, and D. Gibson, "Demonstrating the impact and benefits of CMMI: an update and preliminary results," *Technical report*, 2011.
24. K. El Emam, and I. Garro, "Estimating the extent of standards use: the case of ISO/IEC 15504," *Journal of Systems and Software*, vol 53, 2, pp. 137-143, 2000.
25. F. McCaffery, G. Coleman, "Lightweight SPI assessments: what is the real cost?" *Software Process: Improvement and Practice* vol 14, 5, pp. 271-278, 2009.
26. S. Zahran, "Software process improvement: practical guidelines for business success," Addison-Wesley: Harlow, England, 1998.
27. J. Statz, D. Oxley, P. O'Toole, "Identifying and managing risks for software process improvement," *Crosstalk, The Journal of Defense Software Engineering*, pp. 1-6, 1997.
28. J. Mendling, H.A Reijers, and W.M.P van der Aalst, "Seven process modeling guidelines (7PMG)," *Information and Software Technology*, vol. 52, pp. 127-136, 2010.
29. K. Stapel, K. Schneider, D. Lübke, T. Flohr, "Improving an Industrial Reference Process by Information Flow Analysis: A Case Study," *PROFES 2007*, pp. 147-159, 2007.
30. C. Kaner, J. Bach, B. Pettichord, "Lessons learned in software testing," John Wiley & Sons, New York, 2002.
31. ISO/IEC 29119 Software Testing standard <http://www.softwaretestingstandard.org/>, 2012.
32. ISO/IEC WD 33063.3, *Information technology - Process assessment - Process assessment model for software testing*, 2012.
33. E. van Veenendaal, J.J. Cannegieter, "Testing Maturity – Where Are We Today? Results of the first TMMi benchmark," <http://www.sysqa.nl/wp-content/uploads/2012/09/Artikel-Testing-maturity-Where-are-we-today.pdf> (2012)
34. M. Unterkalmsteiner, T. Gorschek, A.M. Islam, C.K. Cheng, R.B. Permadi, R. Feldt, "Evaluation and measurement of software process improvement - A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 99, 2011.
35. L. Westfall, "12 Steps to Useful Software Metrics," The Westfall Team, <http://floors-outlet.com/specs/spec-t-1-20111117171200.pdf>, 2005.
36. V.V. Mohan, "The Intensity of Fault Slip through in Software Process," *International Journal of Scientific & Engineering Research* vol. 3, 3, 2012.



# On Rapid Releases and Software Testing

Mika V. Mäntylä<sup>1</sup>, Foutse Khomh<sup>2</sup>, Bram Adams<sup>2</sup>, Emelie Engström<sup>3</sup>, Kai Petersen<sup>4</sup>

<sup>1</sup> *Dep. of Computer Science and Engineering, Aalto University, Finland*

<sup>2</sup> *SWAT-MCIS, École Polytechnique de Montréal, Québec, Canada*

<sup>3</sup> *Dep. of Computer Science, Lund University, Sweden*

<sup>4</sup> *School of Computing, Blekinge Institute of Technology, Sweden*

*mika.mantyla@aalto.fi, {foutse.khomh, bram.adams}@polymtl.ca, emelie.engstrom@cs.lth.se, kai.petersen@bth.se*

**Abstract**—Large open and closed source organizations like Google, Facebook and Mozilla are migrating their products towards rapid releases. While this allows faster time-to-market and user feedback, it also implies less time for testing and bug fixing. Since initial research results indeed show that rapid releases fix proportionally less reported bugs than traditional releases, this paper investigates the changes in software testing effort after moving to rapid releases. We analyze the results of 312,502 execution runs of the 1,547 mostly manual system-level test cases of Mozilla Firefox from 2006 to 2012 (5 major traditional and 9 major rapid releases), and triangulated our findings with a Mozilla QA engineer. In rapid releases, testing has a narrower scope that enables deeper investigation of the features and regressions with the highest risk, while traditional releases run the whole test suite. Furthermore, rapid releases make it more difficult to build a large testing community, forcing Mozilla to increase contractor resources in order to sustain testing for rapid releases.

**Keywords**—Software testing; release model; builds; bugs; open-source; agile releases; Mozilla.

## I. INTRODUCTION

Due to heavy competition, web-based organizations, both at the server side (e.g., Facebook and Google) and the client side (e.g., Google Chrome and Mozilla Firefox), have been forced to change their development processes towards rapid release models. Instead of working for months on a major new release, companies limit their cycle time (i.e., time between two subsequent releases) to a couple of weeks, days or (in some cases) hours to bring their latest features to customers faster [1]. For example, starting from version 5.0, Firefox has been releasing a new major version every 6 weeks [2].

Although rapid release cycles provide faster user feedback and are easier to plan (due to the smaller scope) [3], they also have important repercussions on software quality. For one, enterprises currently lack time to stabilize their platforms [4] and customer support costs are increasing because of the frequent upgrades [5]. More worrying are the conflicting findings that rapid release models (RRs) are either *slower* [6] or *faster* [7] at fixing bugs than traditional release models (TRs). Even in the latter case, the study still found that proportionally less bugs were being fixed, and that the bugs that were not fixed led to crashes earlier on during execution.

Since testing plays a major role in quality assurance, this paper investigates how RR models impact software testing

effort. For example, Porter et al. noted that, since there is less time available, testers have less time to test all possible configurations of a released product, which can have a negative effect on software quality [8]. On the other hand, other studies have reported the positive effects of RRs on software testing and quality in the context of agile development, where testing has become more focused [9], [10]. To our knowledge, the impact of RRs on software testing measures, beyond defect data, have not yet been investigated.

Hence, to analyze the impact of RR models on the testing process, we analyze the system testing process in the Mozilla Firefox project, a popular web browser, and the changes it went through while moving from a TR model of one release a year to an RR model where new releases come every 6 weeks. We analyzed the system-level test case execution data from releases 2.0 to 13.0 (06/2006–06/2012), which includes five major TR versions (2.0, 3.0, 3.5, 3.6, and 4.0) with 147 smaller releases (20 alphas, 29 betas, 12 release candidates, and 86 minor), and nine RR versions (5.0 until 13.0) with 89 smaller releases (17 alphas, 56 betas, and 16 minor).

Based on this data and feedback from a Mozilla QA engineer, we studied the following four research questions:

*RQ1) Do RRs affect the amount of testing performed?*

RRs perform more test executions per day, but these tests focus on a smaller subset of the test case corpus.

*RQ2) Do RRs affect the number of testers working on a project?*

RRs have less testers, but they have a higher workload.

*RQ3) Do RRs affect the frequency of testing activity?*

RRs test fewer, but larger builds.

*RQ4) Do RRs affect the number of configurations being tested?*

RRs test fewer platforms in total, but test each supported platform more thoroughly.

A better understanding of the impact of the release cycle on testing effort will help software organizations to plan ahead and to safely migrate to an RR model, while enabling them to safeguard the quality of their software product.

The rest of the paper is organized as follows. Section II provides some background on Mozilla Firefox, while Section III describes the design of our study. Section IV presents the results of the study, followed by a discussion of these results

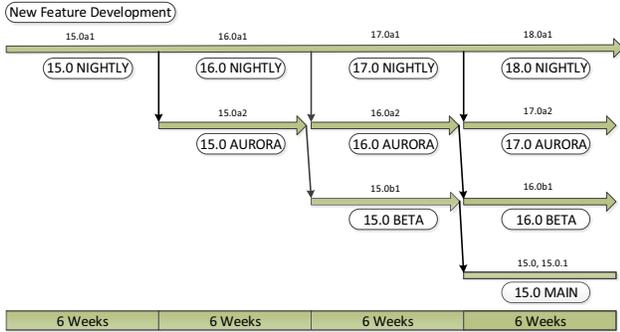


Figure 1: Rapid release process of Mozilla Firefox.

(Section V). Section VI relates our study with previous work. Finally, Section VII concludes the paper and outlines some avenues for future work.

## II. BACKGROUND

### A. Firefox Development Process

Firefox is an open source web browser developed by the Mozilla Corporation. As of April 2013, Firefox has approximately 22% of web browser usage share worldwide [11], with almost half a billion users. Firefox 1.0 was released in November 2004 and the latest version, Firefox 21, was released on May 7, 2013, containing more than 8 MLOC (especially C++, C and JavaScript).

Firefox followed a traditional release model until version 4.0 (March 2011), after which it moved to a rapid release cycle from version 5.0 on, in order to compete with Google Chrome’s rapid release model [4], [12], which was eroding Firefox’s user base. Every TR version of Firefox was followed by a long series of minor versions, each containing bug fixes or minor updates over the previous version. However, in the RR model, every Firefox version now flows through four release channels (Figure 1): nightly, aurora (alpha), beta and main.

In RRs, the versions basically move from one channel to the next every 6 weeks [13]. The nightly channel integrates new features from the developers’ source code repositories as soon as the features are ready. The aurora channel inherits new features from nightly at regular intervals (*i.e.*, every 6 weeks). The features that need more work are disabled and left for the next import cycle into aurora. The beta channel receives only new alpha features from aurora that are scheduled by management for the next Firefox release. Finally, mature beta features make it into main, *i.e.*, the next official release.

### B. Firefox Quality Assurance

Firefox heavily relies on contributors and end users to participate in the quality assurance (QA) effort. The estimated number of contributors and end users on the channels are respectively 100,000 for nightly, 1 million for alpha (aurora), 10 million for beta and 100+ millions for a major Firefox version [14]. Except for the automated Mozmill infrastructure for

in-house regression testing, the testing done by the community is mostly manual.

To co-ordinate this community-based testing, Firefox has a system-level regression testing infrastructure called Litmus [15]. As explained by a Mozilla QA engineer, “*We use it primarily to test past regressions ... and as an entry point for community involvement in release testing*”. It consists of a database with well-documented, functional test cases and stored execution results that are used to make sure that all functionality still works. Each test case corresponds to a user-visible feature, for example “Standard installation”, “Back and Forward buttons”, and “Open a new window”. The test case for “Back and Forward buttons” states: “Steps to perform: 1) Visit two successive sites. 2) Click Back button twice. 3) Click Forward button twice. Expected Results: page loading should move back and forward through history as expected”.

The interface of Litmus is a web-based GUI that allows contributors to follow the status of currently running or archived test executions, and to submit the results of manual tests. Furthermore, users can consult the error messages generated during failing test runs. Litmus is used mainly for beta, release candidate, main, and minor versions, but much less frequently for alpha releases (only 27% of them used Litmus). The pass percentage for test executions in Litmus is around 98%.

## III. STUDY DESIGN

In order to address the four research questions presented in the introduction, we mined the test execution data stored in Firefox’ Litmus repository, then performed various analyses on this data. The remainder of this section elaborates on our data collection and analysis.

### A. Data Collection

We performed web crawling of the Litmus system to get the test cases and the execution data of the test cases for Firefox versions 2.0 to 13.0. Overall, we identified 1,547 unique test cases (roughly 10% of them are automated) for a total of 312,502 test case executions across 6 years of testing (06/2006–06/2012), performed by 6,058 individuals on 2,009 software builds, 22 operating system versions and 78 locales. During this time frame, the Firefox project made 249 releases, of which 213 releases (142 TR and 71 RR) reported their testing activity into the Litmus system. We only consider data until June 2012, since immediately afterwards Litmus was replaced by the Moztrap system in order to enable adding new test cases in a collaborative way and scaling up in terms of usability and functionality [16]. The transition to Moztrap happened instantaneously from one version to the next, which means that our data is not biased by this transition.

In Litmus, all test cases provide the following information: major version number of the release (2.0–13.0), unique identifier, summary, regression bug identifier (if any), the test steps to perform, the expected results, the test group and subgroup the test case belongs to (if any), and links to the corresponding test execution results. Each test execution contains the following information: status (“pass”/“fail”/“test

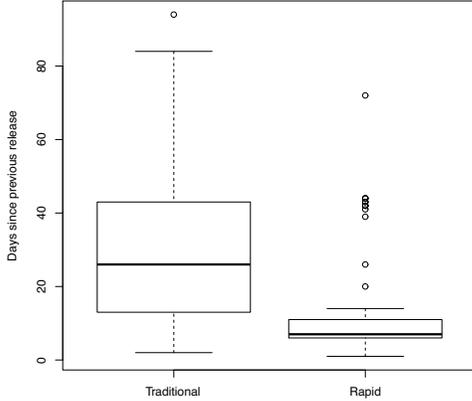


Figure 2: Distribution of release cycle length (in days) for TRs and RRs.

unclear or broken”), test case identifier, time-stamp, platform (e.g., Windows), operating system (e.g., Windows XP), build identifier, locale, user agent, referenced bugs (if any), comments (if any) and the test logs (if any). However, there was no explicit information of the exact release (alpha, beta, release-candidate, major or minor) each test execution was related to. Hence, we mapped the test execution time-stamps to the release dates, which were available online [17].

For each of the analyzed versions, we also extracted the code revision history from the Mercurial repository [18] and parsed it to extract information about the frequency and number of commits. Finally, to triangulate our findings we performed an email interview with a Mozilla QA engineer who has been working on QA for the Firefox project for the past five years. Although the interview results are based on the views of one Mozilla employee (and hence might be incomplete or contain small inaccuracies), they were consistent with our analysis results and provide insights into some of our empirical findings.

### B. Data Analysis

We analyze the collected data set using a set of metrics defined specifically for each question. Details of the metrics are provided later on in the sections discussing the research questions. We use the R statistical analysis tool to perform all the calculations. The Shapiro-Wilk test showed that our data is not normally distributed. Therefore, we use non-parametric statistical analysis throughout the paper. To compare between two groups, we use the Wilcoxon rank-sum test (WRST) and to study effect sizes we use Cliff’s delta (provided by the R package orrdom [19]), which varies between -1 and 1.

As the length of software release projects can vary greatly, we normalized all metrics for project duration (measured in days) to make statistical comparison between TR and RR releases possible. In the RR model, the median time between releases is 7 days while in the TR model it is 26 days, if we consider all types of releases (i.e., alpha, beta, release candidate, major and minor) together. This difference is statistically

Table I: Comparison between metrics for the TR and RR. Effect size uses Cliff’s Delta.

	TR (median)	RR (median)	WRST (p-value)	Effect size
Release length	26.00	7.00	<b>0.000</b>	-0.586
Test exec. per day (RQ1)	50.86	127.3	<b>0.000</b>	0.359
Testcases per day (RQ1)	10.65	14.00	0.855	0.015
Testers per day (RQ2)	1.667	1.000	<b>0.000</b>	-0.297
Builds per day (RQ3)	0.427	0.250	<b>0.004</b>	-0.242
Locales per day (RQ4)	0.302	0.143	<b>0.000</b>	-0.356
OSs per day (RQ4)	0.270	1.286	<b>0.000</b>	0.695

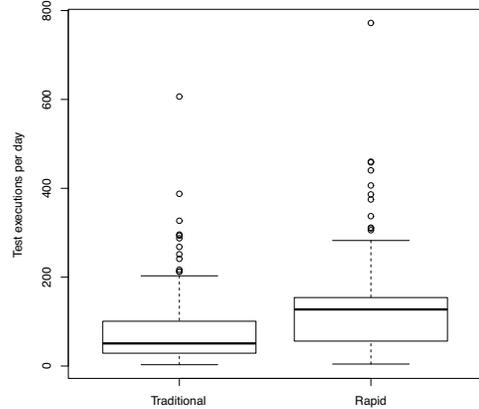


Figure 3: Distribution of number of test executions per day for TRs and RRs.

significant with a large effect size, see Table I. Note that this normalization does not apply to Figure 4, Figure 5, Figure 7 and Figure 9, which show cumulative growth.

## IV. CASE STUDY RESULTS

This section discusses for each research question, its motivation, the approach we used, our findings and the feedback by the interviewed QA engineer.

### RQ1) Do RRs affect the amount of testing performed?

*Motivation:* In our previous work [7], we found that RR models fix bugs faster than TR models, but fix proportionally less bugs. Since the short release cycle time of RR models seems to allow less time for developers to test the system, QA teams could decide to reduce the amount of testing for their RR versions in order to cope with their tight schedule. In this research question, we verify this by investigating the amount of testing effort performed for each TR and RR version of Firefox.

*Null Hypotheses:* We test the following two null hypotheses to compare the amount and the functional coverage of tests executed for TR and RR models:

$H_{01}^1$ : There is no significant difference between the number of tests executed per day for RR releases and TR releases.

$H_{02}^1$ : There is no significant difference between the number of unique test cases executed per day for RR releases and TR releases.

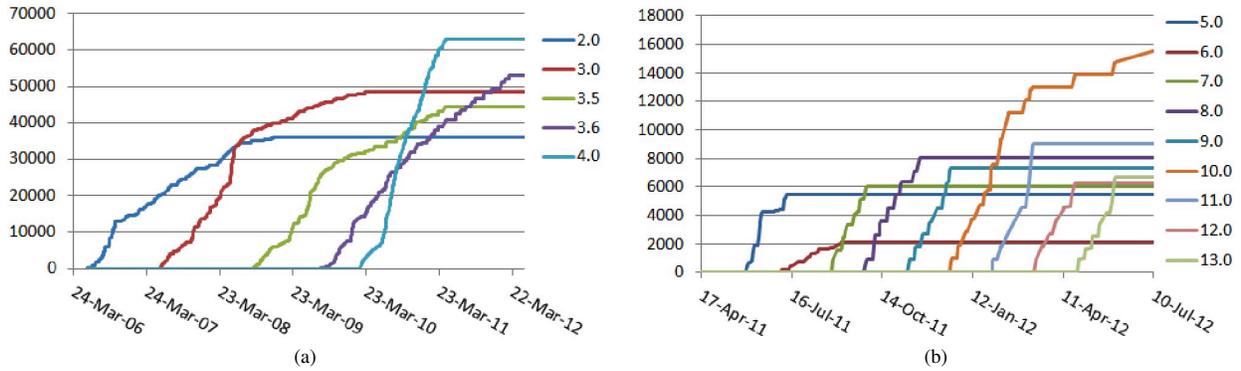


Figure 4: Cumulative number of test executions over time (not normalized) for (a) TR and (b) RR releases.

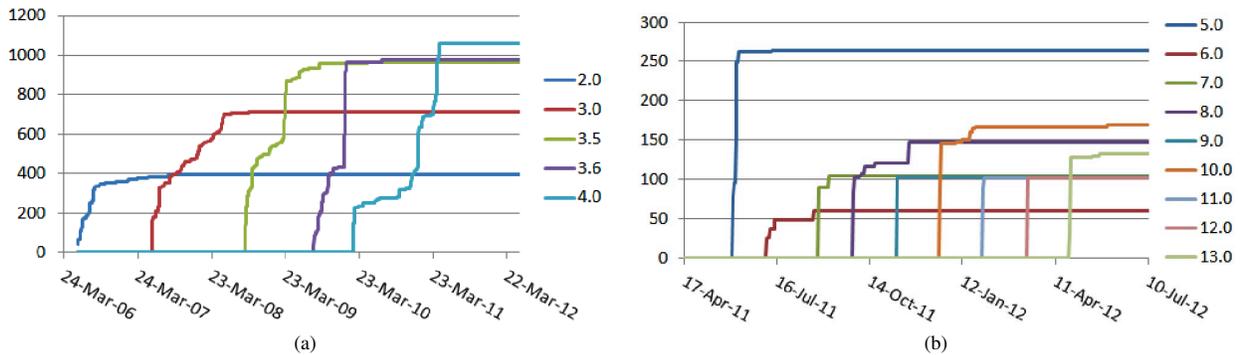


Figure 5: Cumulative number of unique test cases executed over time (not normalized) for (a) TR and (b) RR releases.

We use the Wilcoxon rank-sum test [20] to test  $H_{01}^1$  and  $H_{02}^1$  using a confidence level of 1% (i.e.,  $p$ -value  $< 0.01$ ).

**Metrics:** For each alpha, beta, release-candidate, major and minor version of Firefox in our data set, we compute the following two metrics that capture the amount of tests executed and the functional coverage of the tests. Functional coverage is the degree to which the different features in a software version are tested by a test suite. The more unique (i.e., functionally different) test cases are being executed, the higher the functional coverage.

- #Test exec. per day: the number of tests executed per day.
- #Test cases per day: the number of unique test cases executed per day.

**Findings: The RR model executes almost twice as many tests per day (median) compared to TR models.** Figure 3 and Table I show a statistically significant difference between the two distributions, with a medium effect size of 0.359 (Cliff’s delta), i.e., RR models run more tests in a shorter time frame. Therefore, we reject  $H_{01}^1$ . This difference is also clear from Figure 4, which shows the cumulative (absolute) number of test executions for each major TR and RR releases over time. Since alpha releases typically are not tested in Litmus, the data for each RR starts from the first beta release (this holds for all cumulative plots in this paper). The number of test executions obtains a much higher absolute value (between

35,000 and 50,000, except for the 4.x release series) than the RR releases (usually smaller than 9,000, except for release 10.0), but accumulates over a much longer time.

Firefox 10.0 is an exception for RRs, since it is the first “Extended Support Release” (ESR) [21], i.e., it is meant to last for 54 weeks instead of 6 (lifetime of 9 “normal” RRs). An ESR helps corporate clients [22] to certify and standardize on one particular browser version for a longer period, while still receiving security updates (backported from more recent non-ESR releases). We can see that testing for 10.0 evolved linearly until its release, after which testing is resumed only shortly before the release of a new version. In between, no testing occurs for 10.0.

Especially for the TR releases, testing continues even though development on a newer version has already started. This is due to the many minor releases that follow a major TR release. For RR releases, testing of the next release starts soon after the testing for the previous release stops. Release 10.0 again is an exception, since testing needs to continue for 9 releases. All releases (TR and RR) see accelerated test execution right before a release, which is visible as an almost vertical trend in Figure 4.

**Similar functional test coverage per day, but lower coverage overall.** Data exploration revealed that the test cases executed for each major release vary based on the

features implemented in the release. The median similarity of functional test coverage between subsequent major releases was 56% for both TRs and RRs. We counted the number of unique test cases executed for a particular release, then divided this by the length of the release cycle to compare the number of unique test cases per day executed by each release. We found that, on average, the number of unique test cases executed per day is slightly higher in RR. However, this difference is not statistically significant and the effect size is almost non-existent (0.015). Therefore, we cannot reject  $H_{02}^1$ . This means that, although a particular test case gets more executions in absolute numbers for TR releases, it will be executed more frequently in a shorter time frame for RR releases.

However, since there is less time to run tests, RR testers limit the scope (and hence coverage) of their tests to only the most important ones. Figure 5 shows for each TR and RR release the evolution over time of the cumulative number of unique test cases being executed. The number of different tests executed increases monotonically across the major TR releases, i.e., Firefox 4.x was tested on more cases (almost 1,100) than Firefox 3.5.x and earlier releases. However, the RR releases seem to be tested on progressively less different test cases, going from 270 unique test cases for Firefox 5.0 down to 100 for Firefox 12. Most RR releases reach 90% of their maximum number of unique test cases within a week, which indicates that the reduced scope of testing is determined very early during a new release cycle.

**Feedback QA engineer** The QA engineer could not confirm the difference in the number of test executions, but strongly supported our finding that testing is more focused: *“To survive under the time constraints of a rapid release we’ve had to cut the fat and focus on those test areas which are prone to failure, less on ensuring legacy support”*. In particular, the focused test set consists of *“a fixed set of tests for areas prone to regression (Flash plugin testing for example)”* and *“a dynamically changing set of tests to cover recent regressions we chemspilled for and high risk features”*. A “chemspill” is a negative event like a vulnerability that requires a quick update. Overall, the QA engineer believed that the narrow scope of RR tests is highly beneficial: *“The greatest strength is that the scope of what needs to be tested is narrow so we can focus all of our energy on deep diving into a few areas”*.

*The amount of test executions per day is significantly larger in RR, but these tests focus on a smaller subset of the test case corpus instead of on the full corpus.*

**RQ2) Do RRs affect the number of testers working on a project?**

**Motivation:** With short release cycles, development teams have less time to implement new features and test the features before they are released to users. In **RQ1**, we observed on the one hand a reduction in functional coverage, while on the other hand the remaining test cases are executed more frequently in the shorter time between two releases. Given these observations, does the same testing team as before handle

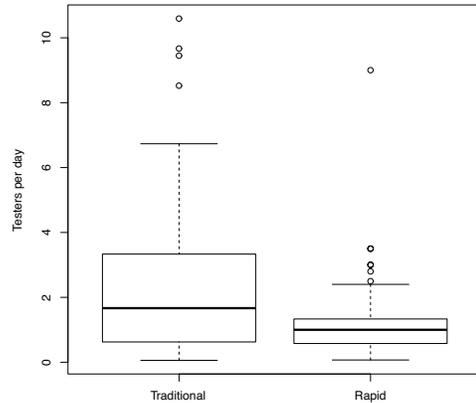


Figure 6: Distribution of number of testers per day for TRs and RRs.

testing, with each tester having to perform less work, or did the test team shrink, either because there is less work to do, or because the rapid succession of releases makes it harder to retain testers?

**Null Hypotheses:** We test the following null hypothesis to compare the number of testers for TR and RR releases:

$H_{01}^2$ : There is no significant difference between the number of testers for RR releases and TR releases.

Similar to **RQ1**, we use the Wilcoxon rank-sum test [20] to test  $H_{01}^2$  using a 1% confidence level.

**Metrics:** For each alpha, beta, release-candidate, major and minor version of Firefox in our data set, we compute the following metric:

- #Testers per day: the number of testers per day.

**Findings: Fewer testers conduct testing for RR releases.**

Figure 6 shows the distribution of the number of individuals per day testing the traditional and rapid releases. We can see that TR releases have a median of 1.67 testers per day compared to 1.0 testers per day for RR releases. The Wilcoxon rank-sum test yields a statistically significant result, i.e., we can reject  $H_{01}^2$ . This result in conjunction with the results of the previous section means that the average workload per individual is much higher under an RR model, since more tests need to be executed per day by less people (i.e., median of 35 vs. 120 test executions per tester per day).

Figure 7a and Figure 7b by themselves do not show a clear trend, with some releases having significantly more testers than others. However, the contrast between TR and RR releases again is very stark when measured from the Litmus system. The most heavily tested RR releases (ESR release 10.0) reached 34 testers by July 2012, which is a factor 56 lower than the 1,900 different testers for version 4.x. Overall, TR releases had a total of 6,010 unique testers, while for RR releases there were only 105 unique software testers registered in the Litmus system.

One possible hypothesis is that the drop in number of testers can be explained by an increase in test automation. For example, across the analyzed Firefox history, we found

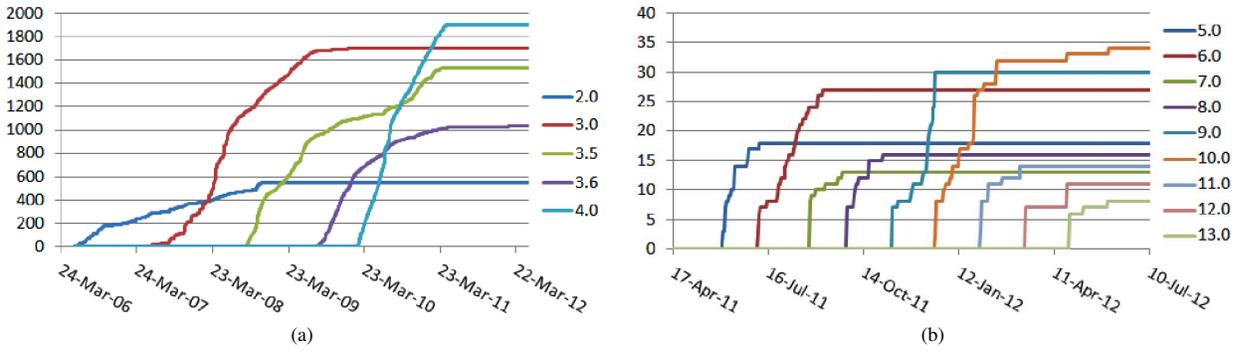


Figure 7: Cumulative number of unique testers running Litmus tests for (a) TR and (b) RR releases.

that 158 out of the 1,547 test cases have been executed by the “#mozmill” username (corresponding to the name of the automated regression testing system). However, 16 of those test cases have been executed only once by “#mozmill”, suggesting a failed automation attempt. Furthermore, all 158 test cases had also been executed with other usernames, suggesting that sometimes the test is run manually (the automated tests contained detailed instruction for manual execution), perhaps due to the test breaking down. However, if changes in the share of test automation would have dramatically impacted testing, this should have led to a significant positive correlation between the evolution of the project and the number of test executions (or cases) per day. Section V-A shows that this is not the case.

**Feedback QA engineer** The interview confirmed our statistical findings about the decreasing number of testers: “The weakest point [in RR] is that it’s harder to develop a large community which more accurately represents the scale and variability of the general population. Frequently this means that we don’t hear about issues until after release, in effect turning our release early adopters into beta testers”. To counter this, Mozilla has augmented their core testing team with contractors: “1) the core team has remained largely unchanged since adopting rapid release 2) the contract team has nearly doubled . . . We can scale up our team much faster through contractors than through hiring. The time afforded to us to make the switch to rapid releases left little room for failure which is why we took that approach.”. The number of testers has also been impacted by some competing Mozilla projects. Regarding test automation, the QA engineer noted that “many of our Litmus tests have partial coverage across our various automation frameworks”, but that after the switch to Moztrap all automated tests were left out. Furthermore, he confirmed that “I think it’s impossible to say how much [test automation] coverage we have for sure [in Litmus]”.

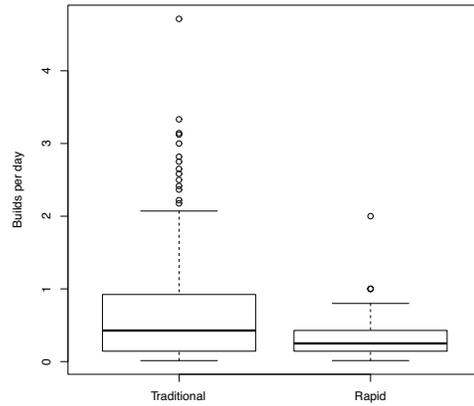


Figure 8: Distribution of the number of builds tested per day for TRs and RRs.

The migration to the RR model has reduced the community participation in testing when adjusting for project duration. However, to keep up with the rapid releases the number of specialized testing resources has increased.

**RQ3) Do RRs affect the frequency of testing activity?**

**Motivation:** Given that more tests are executed per day for RR releases, this could be explained because comparatively more intermediate builds that need testing are produced in a shorter time frame. In other words, developer productivity could have increased compared to TR releases, requiring more tests to be run. Alternatively, maybe the number of builds did not increase significantly, but the amount of change between builds has increased, requiring more testing to be performed on each build. This research question investigates these hypotheses.

**Null Hypotheses:** We test the following null hypotheses:  
 $H_{01}^3$ : There is no significant difference between the number of tested builds per day for RR releases and TR releases.  
 $H_{02}^3$ : There is no significant difference between the number of commits per day for RR releases and TR releases.

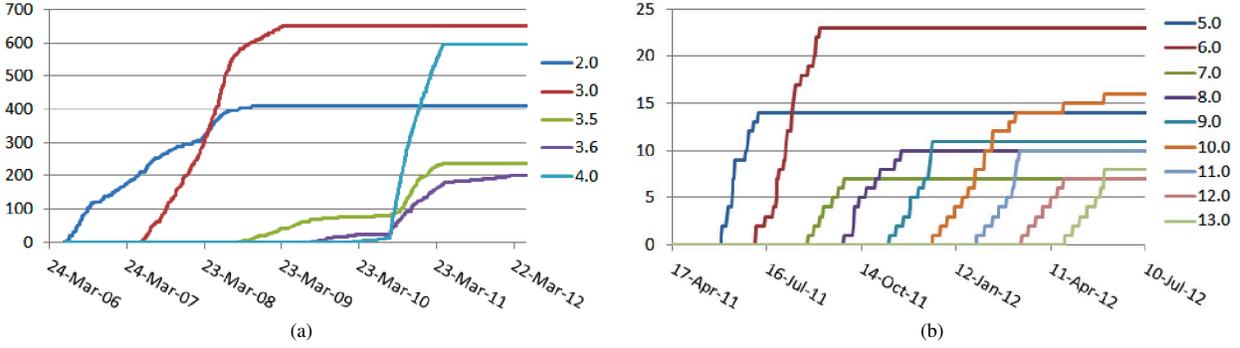


Figure 9: Cumulative number of unique builds for which tests have been run for (a) TR and (b) RR releases.

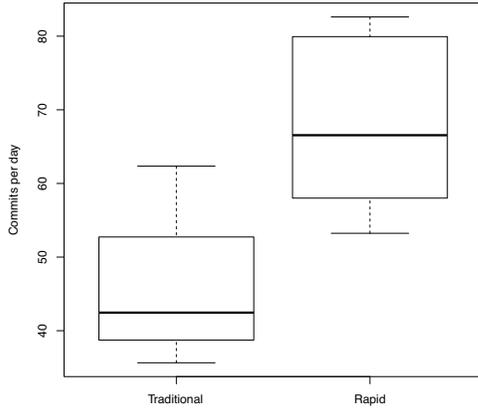


Figure 10: Distribution of the number of commits per day for TRs and RRs.

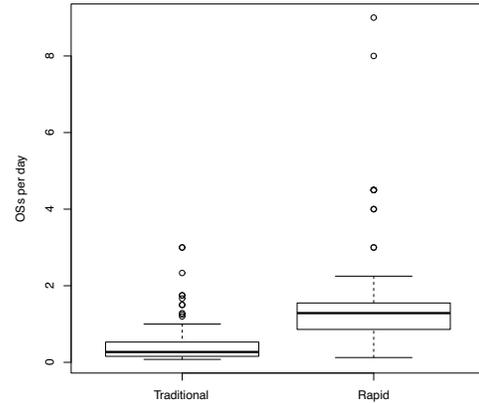


Figure 11: Distribution of number of operating systems tested per day for TRs and RRs.

We again use the Wilcoxon rank-sum test [20] to test these null hypotheses using a 1% confidence level.

*Metrics:* We calculated the following metrics:

- #Tested builds per day: the number of tested builds per day
- #Commits per day: the number of commits to the Mercurial repository per day

We calculated the #Tested builds per day for each alpha, beta, release-candidate, major and minor version of Firefox in our data set, while we calculated the #Commits per day only for the major release, since commits are hard to link to specific releases.

**Findings: Less rapid release builds are being tested per day.** Figure 8 shows that the number of tested RR builds per day is statistically significantly lower than the number of tested TR builds per day (0.25 vs. 0.427). We reject  $H_{01}^3$ . In other words, the higher relative frequency of test executions cannot be explained by more builds being tested, but it seems that each build is tested more thoroughly (albeit with a smaller coverage, see **RQ2**).

**RR builds contain more code commits than TR builds.** To understand why less RR builds are being tested, we

analyzed whether these builds contain more commits relative to TR builds. Figure 10 compares the distribution of the number of commits per day for all major TR and RR releases. RR releases result in statistically significantly more commits per day than the TR releases. Hence, we can reject  $H_{02}^3$ . Since our data exploration showed a downward trend across time in the number of commits, more commits are integrated into version control in a shorter time frame.

**Feedback QA engineer** The QA engineer had not noticed any difference between the number of builds tested between TR and RR releases. However, he agreed that RRs contained more changes, but he attributed this observation more to the project’s evolution than to the RR model: “As time has gone on we have increased the number of changes that land per day”.

*RR releases focus testing on fewer, but larger builds when adjusted for release duration.*

**RQ4) Do RRs affect the number of configurations being tested?**

*Motivation:* The final dimension that we study are the different configurations that are tested, such as different operating systems or support for more locales (i.e., language settings and

internationalization [23]). More thorough testing of different configurations could explain the larger number of tests per build (**RQ3**), as well as the higher workload of individual testers (**RQ2**).

*Null Hypotheses:* We test the following null hypotheses:

$H_{01}^4$ : There is no significant difference between the number of tested locales per day for RR releases and TR releases.

$H_{02}^4$ : There is no significant difference between the number of tested operating systems per day for RR releases and TR releases.

We again use the Wilcoxon rank-sum test [20] to test these null hypotheses using a 1% confidence level.

*Metrics:* We calculated the following metrics:

- #Tested locales per day: the number of tested locales per day
- #Tested operating systems per day: the number of operating systems tested per day

We calculated these metrics for all alpha, beta, release-candidate, major and minor releases of Firefox in our data set.

*Findings: RR tests are conducted on only one locale manually.* When comparing the distribution of the number of tested locales per day, we found that the number of RR locales tested is only half the number of TR locales tested (0.302 vs. 0.143). It should be noted that the locale “English US” dominates the number of test executions in all TR and RR releases. The average share of test executions of the English US locale for TR models is 91%, compared to 99% for RR models.

**A slightly lower number of platforms is being tested, but more thoroughly.** Figure 11 shows that the number of operating systems tested per day has increased by almost 400% (median of 0.27 vs. 1.286) when moving to RR releases. However, the total number of tested operating systems has dropped slightly, with most of the RR releases testing 9 operating systems compared to 12 to 17 for TR releases. This can be partly attributed to the longer time frame of the TRs, e.g., if a major release is tested over a two years period versus 6 weeks (see Figure 4) it is far more likely that new operating system versions enter the market during the longer time period.

Furthermore, when looking at the detailed execution data per operating system, we found for each RR release that all tested operating systems get roughly the same amount of test executions. For TR releases, there were large fluctuations in the number of executions between the tested operating systems.

**Feedback QA engineer** The interview revealed that locale test coverage has actually increased, but has been entirely converted to automated tests, disappearing out of the scope of the Litmus system. Furthermore, the total number of operating systems tested has decreased because “*we now distribute across Betas. For example, we might test Windows 7, OSX 10.8, and Ubuntu in one Beta then Windows XP, Mac OSX 10.7, and Ubuntu in another Beta*”.

Table II: Kendalls’s tau correlation between release model, length and date. Significance levels \*=0.05 \*\*=0.01, \*\*\*=0.001.

	Release length	Project Evolution
Release model	-0.397***	0.634***
Release length	N/A	-0.294***

Table III: Partial correlation of three variables (release model, length and date) to test effort, while controlling two out of the three variables. Significance levels \*=0.05 \*\*=0.01, \*\*\*=0.001.

RQ	Partial Kendall correlation coefficients		
	Release model	Release length	Project Evolution
#Test executions per day (RQ1)	0.026	-0.414***	0.048
#Test cases per day (RQ1)	-0.231***	-0.593***	0.017
#Testers per day (RQ2)	-0.224***	-0.331***	-0.069
#Builds per day (RQ3)	-0.105*	-0.258***	-0.176***
#Locales per day (RQ4)	-0.281***	-0.443***	-0.115*
#OSs per day (RQ4)	0.149**	-0.822***	0.130**

*RR releases test less locales manually. Each supported operating system is tested more thoroughly, but spread across beta releases.*

## V. DISCUSSION

### A. Confounding factors

During our empirical study, we realized that there are two important confounding factors that may affect the results: release length and the project’s natural evolution. First, one could easily think that the differences observed between TR and RR are not due to the release model, but due to the release cycle length, since even some of the TR releases have a shorter release cycle (see Figure 2). Second, the evolution of the project refers to the natural changes and events occurring over time, which are not necessarily related to release length or release model. For example, the reduction in the number of testers over time could be due to re-organization across competing projects or to a loss in community interest. To complicate matters more, both of these confounding factors are impacted significantly by the choice of release model, as the correlations between these variables show (Table II). In these calculations, release length (days between releases) and project evolution (the time-stamp of each release date) are simply modelled as continuous numeric variables, while release model is nominal, set either to zero for TRs or one for RRs.

Hence, we investigated the effect of release model, release length, and project evolution on the metrics calculated for the four research questions, while controlling the confounding effect that these variables have on each other. For this, we used partial correlation (R package ppcor [24]), in which the correlation of one of the RQs’ metrics between RRs and TRs is measured, while controlling two variables out of release model, release length, and project evolution. We used the non-parametric Kendall’s tau instead of linear multiple regression, since the data is not normally distributed.

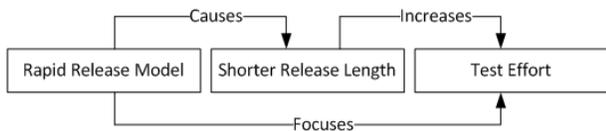


Figure 12: Model explaining the relationship between release model, release length and test effort.

Table III shows that the release model has a significant effect for five out of the six metrics when controlling for the release length and project evolution. It appears that the number of testers, test cases, builds, and locales tested per day are significantly smaller in the RR releases, while the number of operating systems per day significantly increases in the RR releases. On the other hand, the larger number of test executions per day for RR releases is not statistically significant when controlling for release length and project evolution. Instead, the effect that we observed in **RQ1** seems to be due to the consistently shorter time in between releases.

Regarding release length, the results show that test effort overall increases when release length shrinks, i.e., testing becomes more work. One hypothesis, supported by the feedback that we received, is that for the short TR releases a fixed set of regression tests needs to be run, regardless of the release duration. Another hypothesis is that the shorter TR releases are more often rapid patches used to quickly fix major bugs. In such a scenario, the changes in code are small, but as the release is going for millions of users they must be thoroughly regression-tested.

Finally, for project evolution, we find that the number of different locales and builds tested has decreased over time, while the number of operating systems tested has increased over time. No statistically significant change in number of testers can be observed.

Taking all of this into account, our interpretation of the relation between release model and test effort is depicted in Figure 12. Our initial hypothesis was that the shorter release length of RR releases was responsible for increased testing effort. However, even after controlling for the effect of the RRs’ shorter release length, the RRs still have a lower number of test cases, testers, tested builds and tested locales. This means that Firefox’ development process must have changed, as supported by the received feedback, since otherwise one would actually expect a higher proportion of the above metrics. The change in process has focused the testing efforts for the RR releases compared to the TR releases. Only the increase in test executions per day can be fully attributed to the shorter release length.

### B. Limitations

Every empirical study has limitations. First, we cannot be certain that the changes that we see in the TR and RR metrics are caused by the change from TRs to RRs, which affects the construct validity of this study. After all, there could always be hidden factors that actually cause these observed differences,

such as the competing projects in RQ2. To control for this, we triangulated our findings with a Mozilla engineer, who confirmed most of our findings.

Second, although we study over 200 Firefox releases, our study only considers one open source system, which affects the external validity of the study. However, the test data used for this study is not straightforward to obtain, even for open source systems, while for closed source systems substantial data is sealed within corporate walls. Nevertheless, more studies are needed before affirmative conclusions on the effects of a release model on testing effort can be made.

Third, the Litmus database only represents a part of the Firefox testing process, which is mostly aimed at manual regression testing of risky regression test cases and as entry point for community members to join testing. Since we did not study the automated regression test infrastructure, we cannot provide a complete picture of the Firefox testing process. This affects the internal validity of this study.

## VI. RELATED WORK

To the best of our knowledge, this study is the first attempt to empirically quantify the impact of release cycle time on testing in a real world setting with a large quantitative data set. This section looks at the literature on migration from TR to RR release models, followed by work on the impact of agile processes on testing.

### A. Rapid releases and software quality

In recent years, many modern commercial software projects [25], [26] and open source projects backed by a company [12], [27] have switched towards shorter release cycles. Tool builders and researchers (e.g., [8]) have focused especially on enabling continuous delivery [28]. Amazon, for example, deploys on average every 11.6 seconds [26], achieving more than 1,000 deployments per hour.

However, the impact of rapid releases on the quality of the software product experienced by the end user has not been studied until recently. Baysal et al. [6] compared the release and bug fix strategies of Mozilla Firefox 3.x (TR) and Google Chrome (RR) based on browser usage data from web logs. Although the different profiles of both systems make direct comparison hard, the median time to fix a bug in the TR system (Firefox) seemed to be 16 days faster than in the RR system (Chrome), but this difference was not significant.

Khomh et al. [7] studied the impact of Firefox’ transition to shorter development cycles on software quality and found no significant difference in the number of post-release defects, except that proportionally less defects were fixed (normalizing for the shorter time between releases). This suggests that Mozilla’s strategies for testing RR, such as more contractor testing, more focused testing and alternating beta-release testing for operating systems, have been successful in assuring the quality. However, the larger testing community in the TR era might have been able to find some errors earlier than is the case now, which could explain Khomh et al.’s findings about faster crashes in RR versions.

## B. Process changes

Petersen and Wohlin [9] showed that early and continuous testing has a positive effect on fault-slip-through when migrating from plan-driven to agile development with faster releases. They reported on the improvements of test coverage at unit-test level, but highlighted that test cycles are often too short to conduct an extensive system test of quality attributes (e.g., performance), as these are more time intensive. We found that RR reduced the scope of testing, but allowed deeper testing within that scope. Earlier research [29] found that frequent deliveries to subsystem testing allowed earlier and more frequent feedback on each release, and increased the developers' incentives to deliver higher quality.

Li et al. [10] investigated the effect on product quality of introducing Scrum with a 30-day release cycle. They found that the quality focus had improved due to regular feedback for every sprint, better transparency, and an improved overview of remaining defects, leading to a timely (i.e., improved) removal of defects. Kettunen et al. [30] studied the differences in testing activities due to differences in process models. They found that early testing leads to more test execution time and a need for more predictable test resources. Our results support the finding that RR leads to more predictable test resource allocation.

## VII. CONCLUSION

This paper has presented a case study on the effects of moving from traditional to rapid releases on Firefox' system testing. By triangulating data from the Litmus regression testing database with feedback from an interview with a Mozilla QA engineer, we make four key findings. First, we found that due to time-constraints RR system tests have a smaller scope and that the RR model has forced the Firefox testing team "to cut the fat and focus on those test areas which are prone to failure". This narrow scope allows deeper testing in selected areas, which was seen as one of the largest strengths of RR testing. Second, we found that the number of specialized testers has grown due to an increase in the number of contractors, which were needed to sustain testing effort in the rapid release model. However, at the same time the large testing community which "represent the scale and variability of the general population" has decreased. Third, comparison to [7] shows that these rather significant changes in testing process have not significantly impacted the product quality. Fourth, based on empirical data we have proposed a theoretical model explaining the relationship between release model, release length and test effort that needs to be validated in future case studies.

## ACKNOWLEDGMENT

We would like to thank the Mozilla QA engineer who provided feedback to our findings. The responses by Mozilla employees in this paper are accounts of personal experience and opinion, and are in no means whatsoever an official statement from Mozilla.

## REFERENCES

- [1] HP, "Shorten release cycles by bringing developers to application lifecycle management," *HP Applications Handbook*, Retrieved on February 08, 2012, 2012. [Online]. Available: <http://bit.ly/x5PdX1>
- [2] InvestmentWatch, "Mozilla puts out firefox 5.0 web browser which carries over 1,000 improvements in just about 3 months of development," <http://bit.ly/aecRrL>, June 2011.
- [3] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.
- [4] S. Shankland, "Rapid-release firefox meets corporate backlash," <http://cnet.co/ktBsUU>, June 2011.
- [5] M. Kaply, "Why do companies stay on old technology?" Retrieved on January 12, 2012, 2012. [Online]. Available: <http://bit.ly/k3fruK>
- [6] O. Baysal, I. Davis, and M. W. Godfrey, "A tale of two browsers," in *Proc. of the 8th Working Conf. on Mining Software Repositories (MSR)*, 2011, pp. 238–241.
- [7] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in *MSR*, 2012, pp. 179–188.
- [8] A. Porter, C. Yilmaz, A. M. Memon, A. S. Krishna, D. C. Schmidt, and A. Gokhale, "Techniques and processes for improving the quality and performance of open-source software," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 163–176, 2006.
- [9] K. Petersen and C. Wohlin, "The effect of moving from a plan-driven to an incremental software development approach with agile practices," *Empirical Softw. Engg.*, vol. 15, no. 6, pp. 654–693, Dec. 2010.
- [10] J. Li, N. B. Moe, and T. Dybå, "Transition from a plan-driven process to scrum: a longitudinal case study on software quality," in *Proc. of the 2010 ACM-IEEE Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 13:1–13:10.
- [11] R. S. Ltd., "Web browsers (global marketshare)," <http://bit.ly/81klgi>, April 2013.
- [12] S. Shankland, "Google ethos speeds up chrome release cycle," <http://cnet.co/wIS24U>, July 2010.
- [13] D. Sicore, "New channels for firefox rapid releases," <http://bit.ly/hc1zmY>, April 2011.
- [14] R. Paul, "Mozilla outlines 16-week firefox development cycle," <http://bit.ly/fLHEfo>, March 2011.
- [15] Mozilla, "Litmus wiki," <http://mzl.la/evJmTW>, January 2013.
- [16] —, "Moztrap wiki," <http://bit.ly/XBGMfu>, January 2013.
- [17] Wikipedia, "Firefox release history," <http://bit.ly/Ngvfln>, January 2013.
- [18] Mozilla, "Mozilla source code mercurial repositories," 2013. [Online]. Available: <http://hg.mozilla.org/>
- [19] J. J. Rogmann, "orddom: Ordinal dominance statistics," <http://bit.ly/Y0K0eo>, January 2013.
- [20] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*, 2nd ed. John Wiley and Sons, inc., 1999.
- [21] Wikipedia, "Extended support release," [http://bit.ly/ZlgqoM#Extended\\_Support\\_Release](http://bit.ly/ZlgqoM#Extended_Support_Release), January 2013.
- [22] R. Paul, "Firefox extended support will mitigate rapid release challenges," <http://ars.to/M2TbFQ>, January 2012.
- [23] Wikipedia, "Locale," <http://bit.ly/2iJLwB>, January 2013.
- [24] S. Kim, "ppcor: Partial and semi-partial (part) correlation," <http://bit.ly/XkWuyn>, October 2012.
- [25] A. W. Brown, "A case study in agile-at-scale delivery," in *Proc. of the 12th Intl. Conf. on Agile Processes in Software Engineering and Extreme Programming (XP)*, vol. 77, May 2011, pp. 266–281.
- [26] J. Jenkins, "Velocity culture (the unmet challenge in ops)," Presentation at O'Reilly Velocity Conference, June 2011.
- [27] E. Gamma, "Agile, open source, distributed, and on-time – inside the eclipse development process," Keynote at the 27th Intl. Conf. on Software Engineering (ICSE), May 2005.
- [28] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.
- [29] K. Petersen and C. Wohlin, "A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case," *J. Syst. Softw.*, vol. 82, no. 9, pp. 1479–1490, Sep. 2009.
- [30] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander, "A study on agility and testing processes in software organizations," in *Proc. of the 19th Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2010, pp. 231–240.

# How Multiple Developers Affect the Evolution of Code Clones

Jan Harder  
Software Engineering Group  
University of Bremen, Germany  
harder@informatik.uni-bremen.de

**Abstract**—The use of copy and paste in programming causes redundant passages of source code. The effect such clones have on software quality and maintainability in particular has been subject to various studies in the recent past. Although negative effects could not be confirmed in general, a non-negligible number of situations where clones did cause problems has been found. Hence, there may be yet unknown influencing factors that cause these problems. One such factor may be the number of developers involved in the creation and maintenance of a clone. The interaction of multiple developers, unevenly distributed knowledge and communication deficiencies may lead to unwanted inconsistencies and bugs, when the clones are changed. This paper presents an empirical study on long-lived software systems, in which we analyze how many developers are involved in the maintenance exact clones and whether the number of developers affects the likelihood of inconsistent changes. Our results indicate that differences between single-author and multi-author clones exist. Nevertheless, we did not find multiple developers to be the cause of problematic changes to clones.

**Keywords**—code clones; code authorship; software evolution; software quality;

## I. INTRODUCTION

The source code of software systems often contains redundant passages that resulted from copy-and-paste programming. There is a widespread belief that such code clones complicate software maintenance, because changes must be applied repeatedly to all copies. Furthermore, it is often assumed that failing to change all copies consistently may cause new bugs or incomplete bug removals.

These conjectures have been subject to various studies in the recent past. The results are diverse. On the one hand, higher maintenance activity—and therefore higher costs—as well as a general tendency to bugs could not be confirmed for cloned code. On the other hand, cases where cloning caused bugs have been identified. These results suggest that clones do not pose a threat to correctness or maintenance in general. Nevertheless, some clones do. As to now, no method to separate the 'good' from the 'bad' is known.

One possible factor that may complicate the maintenance of clones is the number of developers involved in creating and changing them. While a single developer may be able to keep track of the copies she or he made, copying someone else's code without letting her or him know, could result in unintended inconsistent changes to the clones in the future. So far, possible authorship effects like these have rarely been researched. Balint and colleagues categorized changes to

clones through different authors in an explorational study [1]. They found inconsistent changes by different authors to be a reoccurring pattern and suggest to use the number of developers, who are involved in cloning, as a metric to prioritize clones in practice. Nevertheless, the frequency of such changes and whether the inconsistency was indeed unintended has not been evaluated, yet. Cai and Kim correlated the life-time of clones with different metrics and found that the number of developers has a strong positive correlation [2]. That is, clones, to which multiple developers contribute, remain in the system longer. It remains, however, unknown whether a long lifetime is positive (e.g., code maturity) or negative (e.g., difficult to remove).

Further indication that multiple developers may cause problems with clones is given by an observational field study on program comprehension by Roehm and colleagues [3]. They observed that developers tend to copy code from others instead of changing or extending it. Either because they do not fully understand the implementation or because they cannot foresee all consequences of changing the existing code fragment. When clones are created because of such uncertainty and communication is avoided, the question whether they lead to further problems arises. In a self-experiment we found it difficult to decide whether an inconsistent clone poses a problem, when the original and the copy were created by different developers [4]. It was particularly difficult to gather all required information from different people.

In this study, we analyze the relationship between code clones and code authorship in five subject systems. Our aim is to shed light on the aforementioned assumptions. To this end, we define the following research questions.

**Question 1** — *How many developers are involved in the creation and maintenance of clones?*

Before the effects of multiple developers can be evaluated, we need to examine how often clones are indeed authored by more than one developer. We define a mechanism to detect the authorship of cloned code fragments and quantify how often multiple developers are involved in the creation and maintenance of clones.

**Question 2** — *Do clones change more often when multiple developers are involved?*

We analyze whether clones that have multiple authors are more likely to change. An increased change frequency can imply higher maintenance effort. Each change entails the risk of an unintended inconsistency.

**Question 3** — *Are inconsistent changes more likely if multiple developers are involved?*

We also analyze the change consistency. That is, whether the changes to a clone are applied consistently to all copies. Inconsistent changes can cause faulty program behavior. If a cloned bug is fixed inconsistently, the system remains in a defective state.

The above questions will be pursued statistically. While statistics will shed first light on the general relationship, manual inspection of the clones and the changes made to them is essential to understand the rationale behind the cloning. Possible problems, such as unintended inconsistencies and bugs, can only be safely assessed with manual inspections.

**Question 4** — *Do multi-author clones cause unintended inconsistencies or bugs? Does the rationale behind single-author and multi-author cloning differ?*

**Contribution.** Our main contribution is the first empirical data on the effect multiple developers have on clones. We also provide first insights into the rationale of single-author and multi-author cloning. The results provide a basis for the decision whether the number of developers may be used to prioritize clone management activity. In order to detect code authorship, we also present a new method based on the metadata of `Subversion` and the source code. Authorship is analyzed on the level of tokens, which is more precise than other methods that have been proposed previously.

**Outline.** The remainder of this paper is organized as follows. In Section II we give an overview on the related research. Our authorship detection technique is explained in Section III. Section IV describes the setup of our case study, which is followed by the presentation of the results in Section V. In Section VI we discuss possible implications of our results. Possible threats to validity are covered in section VII. Section VIII concludes.

## II. RELATED WORK

Code clones have been subject to research for more than two decades. Various methods and tools to detect clones in source code exist and have been applied in research and practice. The interpretation of the—often vast amount—of results and the management of the detected clones, are an active area of research. An overview is given in the surveys by Koschke [5] and Roy and colleagues [6].

### A. Effects of Clones

A central question is whether clones do actually complicate maintenance and cause bugs. To answer these questions the

evolution of clones has been studied. The following provides a brief summary. An extensive overview is given by Pate and colleagues [7]. It was found that many clones are short-lived, while others that live longer rarely ever change [8], [9]. Comparisons of the change-frequency of cloned and non-cloned code—often referred to as clone stability—showed that, in many systems, cloned code changes even less frequently than non-cloned code [10], [11]. If clones change often they may be more difficult to maintain. Each change also bears the risk of unwanted inconsistencies. Some studies reported on cases where cloned code, indeed, changed more often [12], [13], but it could not be found that clones cause more changes in general.

Other studies investigated how often changes to clones are carried out inconsistently and how often such changes caused new bugs or led to the incomplete removal of existing ones. Inconsistent changes appear frequently [14]. In some systems they accord for more than half of all changes to clones [9], [15]. Cases in which such inconsistencies cause bugs exist and cannot be ignored [16]. A controlled experiment, in which developers were assigned bug-fixing tasks, indicated a high risk of incomplete bug-fixes when bugs are cloned [17]. Nevertheless, a general relation between clones and bugs could not be established [18], [19]. That is, some clones do cause bugs, but most do not.

### B. Clones and Developers

The first to analyze code clones against the background of developers where Kim and colleagues [20]. They defined a taxonomy of the developer intent behind cloning after they observed programmers at work. They found that developers often recall the clones they have created and that this knowledge is difficult to transfer to other programmers. Nevertheless, they did not investigate how the interaction of developers affects clones. Balint and colleagues applied an automatic approach to detect who is changing the clones in three open-source software systems [1]. Using the `blame` functionality of `CVS`, they analyzed who made the last change to each line of a clone and visualized the data in a clone evolution view. They report on reoccurring patterns of changes to clones, including cases where developers change clones, which were created by others, inconsistently. In contrast to our study, they did not quantify their observations. Neither did they investigate whether the inconsistent changes were made intentionally or whether they caused bugs.

Cai and Kim correlated the lifetime of clone genealogies with the number of developers who make changes to the containing files [2]. They found that the more developers are involved, the longer a clone survives in the system. They conclude that clones maintained by multiple developers may be more difficult to remove because a longer life-time was observed for them. Nevertheless, the reason for this relation was not investigated and the findings are solely based on correlation analysis. Our study, in contrast, analyzes who *created* the cloned code, provides statistics how often clones are authored by different developers, how often inconsistent

changes occur in these cases, and whether problems, such as bugs, occur.

### C. Code Authorship

The detection of the author of a code entity has been a concern to other studies previously. The approaches can roughly be partitioned into two categories. First, the contributions of different developers can be quantified on the file level. Cai and Kim’s work uses such an approach, but does not provide detail how the data are gathered [2]. Weyuker and colleagues investigated whether the number of developers who contributed to a file can be used to enhance bug prediction models [21]. They found that this metric does not play a decisive role in this context. In a follow-up study by Bird and colleagues the notion of authorship was refined with contribution networks. These are also based on file-level data [22]. Their results indicate that bugs are more likely to occur when developers change code they are not experienced with. An approach by Gîrba and colleagues extracts the number of changed lines from `CVS` log messages to calculate the code ownership on the file level. They introduce a time-line visualization that shows code ownership and how it changes over time [23]. These metrics are easy to compute from the metadata of a source code management system (SCM), but are not precise enough for a study on clones. Clones are local passages of source code which cannot be effectively analyzed on the file level.

The second category of approaches is the line-based authorship detection. The aforementioned study on clones by Balint and colleagues [1] uses `CVS blame` that provides the revision in which each line was modified last and the user who applied this change. Rahman and Devanbu use the same functionality of `git` to investigate who contributed to buggy fragments of code and conclude that the lack of experience in a particular file is a factor that may induce bugs [24]. Although such authorship detection methods provide data on a much more fine-grained level than file-based techniques, they are still not accurate enough for our study. Even if just one character is modified, the whole line will be regarded as changed, which leads to an overestimation of changes. Furthermore, code formatting—often carried out automatically and in the large—can heavily distort the authorship detection. More imprecision is caused by the fact that the tools used to obtain the author only provide who made the most recent change to a line. The previous history remains hidden.

In this paper we use an incremental approach that is different from both categories. It will be described in Section III.

### D. Authorship vs. Ownership.

The question, who contributed a piece of code, is frequently discussed under the term of *code ownership*. Being the owner of a thing implies possession, control, and responsibility at a certain point of time. These properties cannot be extracted from source code and its evolution alone. For instance, some code may not change over some time, but the responsibility for

it may be transferred to another developer (e.g., when the previous owner leaves the project). SCM metadata only tells who added, modified, or deleted code. Hence, we speak of *code authorship*, since authoring does only imply the creational aspect. This term is more precise for what approaches such as the aforementioned and ours can provide. It also avoids confusion and misinterpretation of the results.

## III. AUTHORSHIP DETECTION

As described in Section II, existing approaches for authorship detection are too inaccurate for our purpose. Consequently, we define a new approach that overcomes the shortcomings of the existing ones. The general idea is to incrementally track the author of each token in the source code over all revisions recorded in the code repository. Tokens are the smallest meaningful unit in the source code. Detecting authors on this level has the advantage that layout changes do not affect the measurements, because tokens abstract from whitespace and line-positions. By obtaining the authorship information from all available revisions, we further ensure that no information is lost. The authors of code fragments and clones are aggregated from the detected token-authors. Changes to the code are extracted from a `Subversion (SVN)` repository. `SVN` was chosen although newer systems such as `git` provide better metadata. We aimed to analyze long-lived systems that provide complete records of their history. `Subversion` has been widely used for more than a decade and many projects used it during their whole life-time. In the following our approach is explained in detail.

### A. Token Authorship

The author of a token is the user who originally added it to the code. Whenever a developer adds a token it will be assigned to him. If existing tokens are changed, we decompose this action into a deletion and an addition of tokens. The newly added tokens will be assigned to the person who made the change. The change analysis differentiates between two kinds of changes: (1) changes inside a file, that is, changes to the file’s token stream and (2) changes on the file-system level, such as copying, moving, or deleting files. Changes to the token stream change the author information as described before, whereas changes on the file-system level are handled differently. When a file is moved from one location to another, the authorship of the tokens inside will not change. When a file is copied, the person making the copy will become the author of all tokens in the file. This distinction is made because we regard the copying of files as the creation of new clones, whereas moving files should not affect authorship. A file may undergo both types of changes (token and file-system level) in the same `SVN` commit. In such cases we split the change into to operations: (1) a token level operation and (2) a file-system level operation. As an example, if person  $p$  creates file  $f$  and person  $p'$  moves that file to another location  $f'$ , all tokens in  $f'$  will still be authored by  $p$ . When  $p'$  moves and changes  $f$  in the same revision, only the newly added tokens in  $f'$  will be assigned to  $p'$ .

Formally, we analyze all relevant revisions  $r \in R$  from a project’s SVN repository. A revision is relevant, if it contains at least one change to a file of the analyzed programming language. A revision is a tuple  $r = (F, C, a)$ , consisting of a set of existing files  $F$ , a set of changes to files  $C$  as they appear in SVN’s revision log, and an author  $a$  who is the committer of  $r$  according to the SVN log (which we also will refer to as  $a = author(r)$ ). For each file we can extract its list of tokens in a specific revision as  $T(f, r)$ . For each token  $t \in T(f, r)$  we determine its author  $a$ . This gives us a list of authors for all the tokens in the file  $A(f, r)$  that is isomorphic to  $T(f, r)$ . That is, the author of the  $n$ -th token  $t_n \in T(f, r)$  is the  $n$ -th author  $a_n \in A(f, r)$ . The calculated list  $A(f, r)$  is stored along with the revision  $r$  so that it can be used for further analysis.

To obtain the token authors, we process all revisions  $r_i \in R$  in the order in which they were added to the repository. For each  $r_i$  we process all changes  $c \in C$  using the following rules:

1) *Additions*: If  $c$  is an addition of a new file that was not present in  $r_{i-1}$ , we create a new author list  $A(f, r_i)$  with the same size as the token list  $T(f, r_i)$  and assign the revision’s author as the author of all tokens in the file  $f$ . Formally,  $\forall a \in A(f, r_i) : a = author(r_i)$ .

2) *Modifications*: If  $c$  is a modification of a file  $f$  that existed in  $r_{i-1}$ , we compute the differences between  $T(f, r_{i-1})$  and  $T(f, r_i)$  using the Longest Common Subsequence algorithm (LCS) on the tokens. This gives us a list of change deltas  $D$  between the two token lists. Each delta  $d \in D$  is a tuple  $d = (s, l, t)$ , where  $s$  is the start index of the change,  $l$  is the length of the change in tokens, and  $t$  is the type of the change, which is either an addition or a deletion of tokens. The deltas are ordered as they appear in  $f$  from its beginning to its end. To apply the changes to  $f$ ’s author information we first copy the previous author sequence to the current revision by assigning  $A(f, r_i) = A(f, r_{i-1})$ . Then we iterate over each  $d \in D$  in reverse order and apply each  $d$  to  $A(f, r_i)$  using the following rules: If  $d$  is a deletion, we remove  $l$  authors from  $A(f, r_i)$  starting from  $s$ . If  $d$  is an addition, we insert  $author(r_i)$  at position  $s$  into  $A(f, r_i)$  for  $l$  times. This updates the author information for  $f$  in  $r_i$ . Afterwards  $A(f, r_i)$  is isomorphic to  $T(f, r_i)$ . We process the deltas in  $D$  in reverse order, because thereby modifications are applied to  $A(f, r_i)$  back-to-forth, which ensures that the starting positions of the deltas are not invalidated by previously processed modifications. If a deletion and an addition appear at the same token position (tokens have been replaced), the deletion is processed prior to the addition.

3) *Deletions*: If  $c$  is a deletion of a file  $f$  that was present in  $r_{i-1}$ , its author list  $A(f, r_{i-1})$  will not be resumed in  $r_i$ .

4) *File Copies*: If  $c$  copies a file  $f$  that existed in  $r_{i-1}$  to a file  $f'$ , then  $f'$  will be handled as an added file according to the rules above.

5) *Unchanged Files and File Moves*: If a file  $f$  has no corresponding change  $c$  in  $r_i$ , or if it was just moved to  $f'$  in the file system, we copy the author information without changing it. That is, for unchanged files  $A(f, r_i) = A(f, r_{i-1})$

and for moved files  $A(f', r_i) = A(f, r_{i-1})$ .

## B. Code Authorship

The author information for tokens must now be aggregated for code fragments, which are sequences of tokens. A code fragment  $frag = (f, r, s, l)$  is a continuous subsequence of the tokens from  $T(f, r)$ , where  $f$  is the containing file,  $r$  the revision,  $s$  is the start index in  $T(f, r)$ , and  $l$  is the length of the sequence. Since  $T(f, r)$  and  $A(f, r)$  are isomorphic, we can obtain the list of token authors of the fragment from  $A(f, r)$  using  $s$  and  $l$ . The list  $A(f, r)$  contains the author for each token in  $frag$ . We quantify the number of tokens from each author of  $frag$ . The *main author* of  $frag$  is defined as the author who contributed most of the tokens. If two or more authors contributed the same amount of tokens to  $frag$  one of them is randomly chosen as the main author. This may appear to be a too strong simplification of our accurate authorship tracking. Nevertheless, it is required to categorize clones according the number of authors. In the Section V we provide data how much of a fragment is authored by its main author on average. The vast majority of all fragments is predominantly authored by just one person. Even though we loose some accuracy by defining a main author we still benefit from the fact that the token-based approach is not extensively distorted by formatting. Comparisons with SVN’s `annotate` command have shown that these would strongly affect the results in some systems.

## C. Clone Authorship

Our definition for code authorship, finally, needs to be transferred to code clones. In our study clones are exact copies of code fragments. Such a fragment may be copied one or more times. All equal fragments together form a *clone class*. Defining the author of clone fragments is straightforward because they are code fragments for which we defined authorship before. Hence, we can obtain the main author for each clone fragment. In our study we are interested in the distinction between clone classes that have just one author and such that have multiple authors. We now can separate the set of all clone classes  $CC$  into such classes whose fragments have different main authors as  $CC_{multi}$  and such classes whose fragments have the same main author as  $CC_{single}$ . That is,  $CC_{single}$  are the single-author clone classes, whereas the multi-author clone classes are in  $CC_{multi}$ .

## D. Sensitivity to initial checkins

Because of its incremental nature, our approach must not only visit every revision to provide accurate results, it also must begin with the very first revision of the project’s history. If the analysis starts with a revision to which multiple developers contributed, it cannot decide who authored the tokens. Typically, SVN repositories begin with an import of some existing code. To avoid wrong author assignments in such cases, we mark all tokens that were added in the very first revision as unresolved. Such tokens will receive special handling. Clones that contain unresolved tokens over a certain

threshold will be excluded from the analysis. We chose subject systems that have only small initial checkins to limit the amount of unresolved clones. How many clones have been excluded per system will be reported in Section V.

### E. Branches

Branches in *SVN* are effectively clones of the whole code base in the `trunk`. Nevertheless, they cannot be compared to clones created by copy-and-paste programming, because they are explicitly managed via *SVN*. Consequently, our approach analyzes only the `trunk` (the main branch) of each system. Another issue regarding branches is the possibility that new code is created by different developers in a branch and then merged into the trunk by one developer. Although *SVN* documents branch creation in its metadata, earlier versions did not document merge operations between branches. Changes incoming from other branches cannot be reconstructed from the data. In such cases the changes will be assigned to the developer who makes the merge. To reduce the effect of such cases on our measurements, we choose subject systems where the development mainly happens in the trunk and branches are mainly used to propagate bug-fixes to previous releases. We will further discuss this in Section IV.

## IV. STUDY PROCEDURE

We use the approach described in section III to detect the authors in five different software systems. In the following we describe how we detected clones and which systems have been analyzed.

### A. Clone Detection

The incremental clone detector `iclones`<sup>1</sup> was used to detect clones in all relevant revisions. `iclones` detects clones in every revision using a token-based detection approach based on a generalized suffix tree. When it proceeds to the next revision it keeps its internal data structures and updates the information for files that have been added, changed, or removed. Göde and Koschke give a detailed explanation of this approach in [25]. `iclones` does not only detect clones in consecutive versions, it also tracks clone fragments across versions using a token-based LCS approach that is similar to the one we use for authorship detection. Based on this difference information it is determined whether the fragments of a clone class have been changed from one revision to the next. To this end, `iclones` tracks the fragment positions across all revisions and checks for each revision  $r$  whether the fragment  $f$ 's token stream has changed by comparing  $T(f, r_i)$  and  $T(f, r_{i+1})$  using LCS. If at least one fragment of a clone class has changes, the clone class will be marked as changed. The creation of a new fragment will not be regarded as change in our case, because it is not a modification of cloned code.

Furthermore, `iclones` detects whether the fragments of a clone class have been changed consistently. A clone class is changed consistently, when the change deltas applied to all its fragments according to LCS are identical. If at least

one fragment has a different set of change deltas, the change is inconsistent. The detection of consistent and inconsistent changes is further explained in [9].

The results of a clone detector strongly depend on its configuration. In the following we describe how `iclones` was configured and which precautions were taken to improve the quality of the results.

1) *Clone Types*: We decided to detect only exact clones (in contrast to near-miss clones where the fragments may differ to a certain extent), because we differentiate between consistent and inconsistent changes to clones. While change consistency can be exactly defined for exact clones it is more difficult to define for near-miss clones, in which the fragments have differences. An inconsistent change to a near-miss clone may make its fragments actually more similar by removing differences. This is not the kind of inconsistency we are interested in.

2) *Minimum clone length*: For the minimum length of a clone we chose 50, 100, and 150 tokens. Because the results regarding authorship did not vary much between these three data sets, we will only report results for the 100 token setting in the remainder of this paper. In practice 100 tokens roughly correspond to 10 to 15 lines of code.

3) *Code Exclusion*: We excluded code that usually contains coincidental clones, which do not necessarily stem from copy and paste. To this end, files that define test cases or contain generated code have been excluded. A similar case are lists of import statements and long array initialization sequences. We removed these artifacts from the token stream automatically prior to clone and authorship detection.

4) *Method separation*: Token-based approaches may detect many clones that only consist of the end of one method and the signature of the next. Such artifacts are not caused by copy and paste. To avoid them, we detect method boundaries in the token-stream using pattern matching and insert artificial delimiter tokens between methods. The clone detector will stop detecting a clone when it encounters such a delimiter. That is, clones that span over multiple methods are separated at the method boundaries. The resulting clones are kept, if they exceed the minimum clone length.

A clone detection approach based on abstract syntax trees (AST) would have provided the syntactical information, too. Nevertheless, such approaches are difficult to use in evolutionary studies on long-lived systems as ours. It cannot be assumed, that the code in each and every revision will be syntactically correct, which is a requirement for these techniques. Furthermore, AST-based approaches demand more time and space for their computations, which is critical when, as in our study, more than 10,000 revisions are analyzed.

### B. Subject Systems

We analyzed the history of five software systems from different domains. Details on systems are listed in Table I. Only relevant revisions are reported in the revision column. The column labeled *max. coverage* reports the maximum clone rate, that is, the highest percentage of cloned tokens among all

<sup>1</sup><http://www.softwareclones.org/iclones.php>

TABLE I  
SUBJECT SYSTEM PROPERTIES

System	Language	Revisions	Begin	End	Years	max. KLOC	max. coverage [%]	Committers
<i>Ant</i>	Java	8,774	2000-01-13	2011-08-19	11.6	215.0	2.1	46
<i>FindBugs</i>	Java	10,339	2003-03-24	2012-07-26	9.4	264.3	14.9	28
<i>FreeCol</i>	Java	10,930	2005-11-02	2013-01-30	7.3	184.8	4.0	35
<i>Handbrake</i>	C	1,970	2006-01-14	2013-04-01	7.2	88.4	4.8	25
<i>swclones</i>	Java	1,378	2009-06-16	2012-03-21	2.8	58.2	5.0	18

tokens, the systems reached during their lifetime. The number of committers is the number of unique users who committed at least one relevant version. Possible alias names, which are used by some users, have been merged as far as they could be reconstructed.

All five systems either did not use branches at all or used them mainly to maintain old releases. If branches were used for other purposes we will describe these in the following.

1) *Apache Ant*: *Ant*<sup>2</sup> is an open-source build-automation system for Java development. For our study we analyzed its `core` component. Several proposals for a next-generation Ant have been made and developed in the *proposals* folder of the code repository, which can be regarded as a branch folder. Some proposals started with a clone of the code base that was heavily refactored and changed. Nevertheless, all proposals have been discarded eventually. Consequently, they have been excluded.

2) *FindBugs*: *FindBugs*<sup>3</sup> is an open-source static analysis tool for Java source code. It searches for typical patterns that indicate potential programming errors. The high maximum clone coverage is caused by an outlier revision. For most of the time it lay between 0.5% and 3.0%.

3) *FreeCol*: *FreeCol*<sup>4</sup> is an open-source reimplementaion of the commercial strategy game *Colonization*. It has been newly developed from scratch by a team of enthusiasts and is written in Java. It reaches its maximum clone coverage in the middle of the analyzed period. From there on, the coverage steadily declined to 0.7% in the last revision.

4) *Handbrake*: *Handbrake*<sup>5</sup> is an open-source multi-platform video transcoder written in C. It consists of a central library that performs the transcoding and three GUI applications for Linux, Mac and Windows. The Windows GUI was left out in this study, because it is written in C# and not in C.

5) *softwareclones.org*: Our research group develops tools for clone detection and analysis in Java<sup>6</sup>. These are the tools that have been used to execute this study. For brevity we will refer to them as *swclones* in the remainder of this paper. The whole code repository including tools for clone detection, analysis, visualization, and transformation was analyzed for this study. The code is developed by a team of 18 researchers and student assistants. Although the main focus of the tools are

<sup>2</sup><http://ant.apache.org>

<sup>3</sup><http://findbugs.sourceforge.net>

<sup>4</sup><http://www.freecol.org>

<sup>5</sup><http://www.handbrake.fr>

<sup>6</sup><http://www.softwareclones.org>

TABLE II  
FRAGMENT MAIN AUTHOR SHARES AND EXCLUDED CLONE CLASSES

System	All [%]	$CC_{single}$ [%]	Excl. [%]
<i>Ant</i>	90.7	95.0	3.245
<i>FindBugs</i>	98.7	99.0	0.000
<i>FreeCol</i>	80.1	94.9	13.767
<i>Handbrake</i>	94.9	98.1	0.004
<i>swclones</i>	96.5	96.0	0.000

code clones, clone detection has only been used sporadically during the development process.

## V. RESULTS

Before we pursue our research questions, we take a look on the feasibility of our approach for the chosen subject systems. The idea of the main author is based on the assumption that code fragments are usually predominantly authored by one author. If this is not the case and different authors equally contribute to the fragments, our classification would be inaccurate. Consequently, we measure the average percentage of tokens the detected main author owns in a clone fragment. The results are shown in Table II. We present the averages for all fragments, and the value for the fragments of only  $CC_{single}$  classes. The second value is important to validate whether  $CC_{single}$  can indeed be regarded as being mainly authored by just one person. Most of the values are roughly 95% or higher. For *Ant* and *FreeCol* the values for all fragments are lower. However, it can still be claimed, that the detected main author made the essential contributions, since no value lies below 80%.

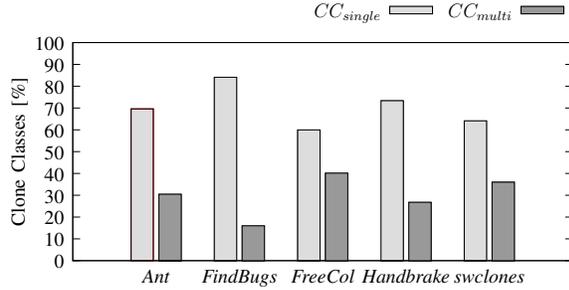
For three of the systems, the SVN history started with the import of an existing code base. That is, for these systems all tokens of the initial revision have been marked as unresolved. As soon as more than one percent of a fragment's tokens are marked as unknown, we excluded the fragment and its whole clone class from the analysis. Higher values of this threshold (up to 5%) did not change the exclusion rate significantly. The exclusion of clone classes is only notable for *Ant* and *FreeCol*. For the latter, 13.8% had to be excluded, because the first 600 revisions in its repository seem to be corrupted and were skipped.

### A. Clone authors

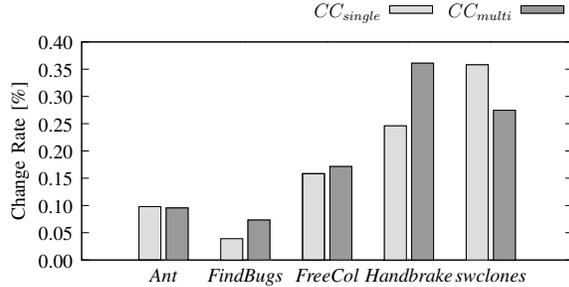
To answer Question 1, that is the question how often clones are authored by more than one developer, we first detect clones

TABLE III  
FREQUENCY OF DIFFERENT CLONE CLASS TYPES (IN THOUSAND)

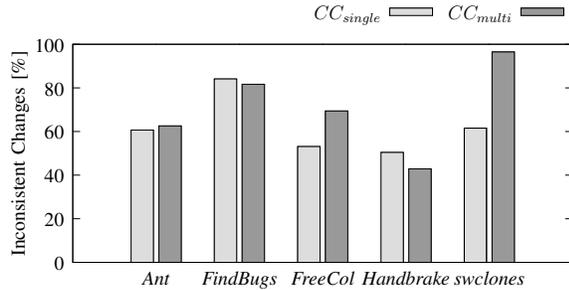
System	All	$CC_{single}$	%	$CC_{multi}$	%
<i>Ant</i>	138.1	96.1	69.6	42.0	30.4
<i>FindBugs</i>	421.0	353.9	84.1	67.1	15.9
<i>FreeCol</i>	413.3	247.6	59.9	165.8	40.1
<i>Handbrake</i>	72.7	53.3	73.3	19.4	26.7
<i>swclones</i>	28.4	18.2	64.1	10.2	35.9



(a) Relative frequency.



(b) Change rate.



(c) Rate of inconsistent changes.

Fig. 1. Comparison of  $CC_{single}$  and  $CC_{multi}$  clone classes

in every relevant revision of each system. All detected clones are categorized into  $CC_{single}$  and  $CC_{multi}$  and counted. If a clone class appears in more than one revision, each occurrence will be counted. The decision whether a clone class is  $CC_{single}$  or  $CC_{multi}$  is made for each occurrence. Table III shows the results which are also visualized in Figure 1a.

For all systems, both  $CC_{single}$  and  $CC_{multi}$  do exist in a notable quantity. The number of  $CC_{single}$  is always higher than the number of  $CC_{multi}$  clone classes. That is, most of the clones we have found in the project histories are mainly

TABLE IV  
CHANGES TO CLONES

System	Sample	CR [%]	CC [%]	IC [%]
<i>Ant</i>	$CC_{single}$	0.098	39.4	60.6
	$CC_{multi}$	0.095	37.5	62.5
	all	0.097	38.8	61.2
<i>FindBugs</i>	$CC_{single}$	0.039	15.9	84.1
	$CC_{multi}$	0.073	18.4	81.6
	all	0.044	16.6	83.4
<i>FreeCol</i>	$CC_{single}$	0.158	46.9	53.1
	$CC_{multi}$	0.171	30.6	69.4
	all	0.164	40.1	59.9
<i>Handbrake</i>	$CC_{single}$	0.246	49.6	50.4
	$CC_{multi}$	0.361	57.1	42.9
	all	0.277	52.2	47.8
<i>swclones</i>	$CC_{single}$	0.358	38.5	61.5
	$CC_{multi}$	0.274	3.6	96.4
	all	0.328	28.0	72.0

authored by just a single developer. Nevertheless, 15 to 40% of the clone classes contain fragments that are mainly authored by different developers.

### B. Change frequency

To answer Question 2 we analyze whether  $CC_{single}$  and  $CC_{multi}$  change differently. First, we analyze whether clone classes of the two categories have the same probability to change. For each clone class we detect in revision  $r_i$ , we check whether its fragments are changed in the transition to  $r_{i+1}$ . If the tokens of at least one fragment were changed, the clone class is marked as changed. The change rate of  $CC_{single}$  clone classes can then be computed as the fraction of the changed  $CC_{single}$  clone classes and the total number of  $CC_{single}$  clone classes. The change rate of  $CC_{multi}$  clone classes is computed accordingly. That is, these change rates reflect the probability that a clone class, found in one revision, is changed in the next revision.

Table IV shows the results. The CR column represents the change rate of the respective clone classes. Figure 1a illustrates the change frequency of  $CC_{single}$  and  $CC_{multi}$  for each system in comparison. First of all, it is to be noted that clones do not change too often in general. In all systems less than 0.4% of all clone classes, we detected during the system's evolution, changed in the next version. As an example, we detected 138,096 clone classes in all revisions of *Ant*, but only 134 of these changed within the 11.6 years we analyzed. These data support findings of a previous study on this topic [9]. The clones in *Handbrake* and *swclones* have a notably higher change rate compared with the clones in the other systems. Both systems have a much shorter history in terms of revisions than the others. It is also these two systems which change rates for  $CC_{single}$  and  $CC_{multi}$  expose the largest difference. In *Handbrake* the change probability of  $CC_{multi}$  clone classes is 55.5% higher than the change probability of  $CC_{single}$  clone classes. In *swclones* the opposite is the case and  $CC_{single}$

clone classes change more frequently. The highest relative difference can be observed in *FindBugs* where the change rate of  $CC_{multi}$  is twice as high as for  $CC_{single}$ . For *Ant* and *FreeCol* no significant difference was measured.

In summary, there is no systematic difference that indicates a general effect clone authorship has on the probability that a clone class changes. The effect seems to be system dependent.

### C. Change consistency

We further analyze the consistency of the changes to answer Question 3. If a clone class changed and all its fragments underwent identical changes, we consider the change to be consistent. If the fragments changed differently or some fragments did not change, while others in the same class did, we consider the change as inconsistent. Table IV shows how often  $CC_{single}$  and  $CC_{multi}$  changed consistently or inconsistently. Figure 1c shows only the percentage of inconsistent changes in comparison for the subject systems. For most systems the rate of inconsistent changes does not differ notably for  $CC_{single}$  and  $CC_{multi}$ . In *FreeCol* and *swclones*  $CC_{multi}$  are changed more often inconsistently than  $CC_{single}$ . In *Handbrake*  $CC_{single}$  have a slightly higher rate of inconsistent changes. Again, there is no systematic difference in the analyzed systems.

### D. Intentions and Bugs

To answer question 4 and to learn more about the intentions and possible defects connected to  $CC_{single}$  and  $CC_{multi}$  clone classes, we manually inspected a large part of the results. To this end, the clone evolution visualization tool *cyclone*<sup>7</sup> was used [26]. *Cyclone* visualizes clone histories, which eases the reconstruction of changes and intentions. For *swclones* and *Ant* we evaluated each change to clones on four levels: (1) For inconsistent changes, whether the inconsistency was intentional, (2) whether the inconsistency caused a bug or is an incomplete bug removal, (3) the rationale of the cloning and the changes to the clones, (4) whether the fragments of the changed clone class were located in the same or different files.

1) *swclones*: We inspected all changes to the clones in *swclones*, because for this system we measured the largest difference in change consistency. Multi-author clones changed more often inconsistently than single-author clones. Furthermore, we have in-depth knowledge about the code which allows us to judge the intentions behind changes and whether an inconsistency is a bug.

The inconsistent changes to clones were mostly intentional for both single-author and multi-author clones. Among the multi-author clones only one out of 25 inconsistent changes was unintentionally inconsistent. A bug-fix regarding the canonicalization of path names was applied only in the default input module of our clone detector. A similar module for network input contains the same bug, which was not fixed. Nevertheless, this module has not been used since. Hence, we

regard this case as rather unproblematic. Surprisingly, 3 out of the 38 inconsistent changes to single-author clones (7.9%) were unintentional and caused actual bugs that still existed in the most recent revision. They have in common that the bugs are located in complex algorithms and that their cloned occurrences lie closely together in the same file. In each case only one of two bug occurrences was fixed.

Regarding bugs we found that multi-author clones were less often subject to bug-fixing activity compared with single-author clones (7.7% vs. 20.6%). Most of these fixes, except the ones mentioned above, were applied consistently.

Single-author and multi-author clones differ in their rationale. Almost all multi-author clones (21 out of 27) are copies between two different viewer applications. One of these was started by copying existing GUI components from the other. The plan to reintegrate both applications at a later point of time was soon abandoned, because it was found that the different functional requirements could be better satisfied in separate tools. Consequently, the two GUIs purposefully developed independently which caused the high volume of inconsistent changes to multi-author clones in *swclones*. The remaining changes to multi-author clones were related to similar cloning across different implementations of the same architectural component.

Not one multi-author clone class had all its fragments in the same file. In contrast, the single-author clones lie to 57.7% in the same file. They often copy logic snippets that are difficult or impossible to unify. For instance, because they implement the same logic for different primitive types.

To summarize, multi-author clones were mostly caused by file cloning, which was less often the case for single-author clones. Furthermore, the changes to multi-author clones contained less problematic cases, such as bug fixes, compared with single-authored ones.

2) *Ant*: As second system, we inspected all 134 changes to clones in *Ant*. According to our statistics single-author and multi-author clones do not differ significantly regarding their change frequency and their change consistency. Since we are not familiar with the code, we cannot safely judge whether inconsistent changes were made intentionally. Nevertheless, we can analyze how single-author and multi-author clones differ in this system. We further inspected the commit messages to evaluate whether the changes were bug fixes.

We found bug-fixing activity to be more common in single-author than in multi-author clones (31.2% vs. 13.2%). Inconsistencies that were obviously unintended, because the missing changes were propagated to all fragments later, rather occurred in the single-authored clones. We clearly identified three such cases for single-author clones, one of which took three attempts over eight months to be fixed consistently. Among the changes to multi-author clones only one such late propagation was found. In this case the inconsistency was fixed in the following revision where the clone was also removed.

The rationale of cloning is less discriminative than in *swclones*, the tendency, however, is the same. The changed multi-author clones lie to 25.64% within the same file. For

<sup>7</sup><http://www.softwareclones.org/cyclone.php>

single-author clones this is the case in 30.11%. This smaller difference may be caused by *Ant*'s highly generic architecture that decomposes the build-process into tasks, each of which are implemented in their own class. These tasks are often similar, especially when they implement similar functionality, such as different commands of the same source code management system.

More interesting is why the clones are changed. Most of the consistent changes to multi-author clones are directed to warnings from static defect checkers, which are usually issues of style, performance, program comprehensibility, or typical programming mistakes. These issues are automatically detected by tools. We did not consider them as bugs. If they are cloned, the tool will point out all locations to the user automatically. Hence, it does not come as a surprise that these are issues are removed consistently. While 71.4% of the consistent changes to multi-author clones are directed to such issues, only 35.1% of the consistent changes to single-author clones fall into this category. Among the inconsistent changes, such issues are rarely the reason for the change (9.1% for multi-author clones vs. 5.6% for single-author clones).

To summarize, we did not find multi-author clones to be related to problematic changes more often than single-author clones. Most inconsistencies are intentional and the rare cases of clearly unintentional inconsistencies were rather found among the single-author clone classes. When multi-author clone classes changed consistently, the reason mostly were automatically detected warnings about coding conventions and style.

## VI. DISCUSSION

Our analysis of code authorship was motivated by the conjecture that problems with clones may be caused when the copied fragments were authored by more than one developer. If this was the case, the authorship of clones could be used to prioritize clones in clone management and to develop measures to prevent such problems. Our results show that multi-author cloning is common in software, albeit most clones are created and mainly maintained by just one developer. The analysis of the change frequency and consistency did not reveal a notable and systematic difference for the multi-author clones. The analysis of further systems may reveal a general tendency towards single-author or multi-author clones. Nevertheless, to be a useful metric for clone management, multiple authors would need to have an effect of considerable size on the harmfulness of clones, which they do not seem to have. Given the small number of changes to clones in general, the relative differences we measured for the change rate and the consistency translate to a very small absolute number of actual cases. To be practically relevant the differences had to be much larger.

Our manual inspection further contradicts our initial conjecture, as the few harmful clones we found were rather single-author than multi-author. The difference between these two kinds of clones is certainly not significant. Nevertheless, our analysis indicates the possibility that single-author clones

could be more likely to cause harm. A possible explanation lies in the different rationales we observed for single-author and multi-author clones. As our manual inspection revealed, multi-author clones tend to be larger copies of whole structures, whereas single-author clones were more often smaller copies of complex algorithmic parts. As an example, an inconsistency in the layout of two GUI dialogs should be less harmful than an inconsistency in some critical part of the application logic. We observed this tendency for both *swclones* and *Ant*.

In both systems, for which we did manual inspections, we observed that consistent changes were very often directed to style issues, while inconsistent ones rarely are. When multiple authors were involved in the cloning, the amount of style related changes was higher in *Ant*. This is also true for *swclones*, but this result is based on only one existing consistent change to multi-author clones.

Our study investigates the effect of multiple developers of clones from only one perspective. Besides the question whether it makes a difference if different developers create and maintain the copied fragments, it may also be asked, whether unwanted inconsistencies occur when a clone authored by one developer is changed by another. This, however, is another question that should be asked in broader context because changing someone else's code may cause problems in general and may not be limited to clones only. Future research should be directed to this question.

## VII. THREATS TO VALIDITY

In the following we describe possible threats to the validity of our study.

### A. Internal Validity

The results of our study depend on the metadata provided by *SVN*. As any other SCM it can only provide what was explicitly inserted. Information may be lost if files are moved or copied without using *SVN*'s commands for this purpose. In such cases, the authorship may change incorrectly. If changes are proposed as patches by an external developer, the project member who commits the changes appears as their author. This is not exactly true, however, the committing author is still responsible for the correctness of the changes she or he makes. In some projects the same developer contributes under different user names. We merged such users as far as we could reconstruct their aliases.

Another possible threat is the Longest Common Subsequence (LCS) algorithm, which we use to analyze changes to the token stream. In practice, LCS is usually able to reconstruct the changes that were applied to a sequence correctly. Nevertheless, is not able to track code movements inside a file. If, for instance, a method is moved inside a file, our approach will regard this as a deletion and an addition that was authored by the developer who moved the method. Such cases may appear in refactorings. To evaluate whether such changes do largely affect the authorship detection, we analyzed how the amount of authored tokens changes over time for each developer. We did not find abrupt changes in authorship, which would be

visible if such situations occurred frequently and affected the measurements thoroughly.

### B. External Validity

Although we chose multiple long-lived systems from different domains, we cannot generalize our results on all other software systems. Most of the analyzed systems are written in Java and developed as open-source software. Systems written in other languages or in an industrial context may expose different properties. Furthermore, our results indicate that the relationship between developers and the effects of clones, may be system dependent. Future research should analyze further systems, written in different languages and with an industrial context.

The results of our study are valid for exact clones. We did not analyze clones with different variable names or gapped clones, because the definition of change consistency is problematic for these. Inconsistent changes may split previously found exact clones into multiple classes. In our manual inspection of all changes to the clones in *swclones* and *Ant*, in which we also analyzed clone genealogies, such cases did not appear frequently. Future research should further investigate how multiple authors affect the evolution of near-miss clones.

## VIII. CONCLUSION

In this study we analyzed how multiple developers affect the evolution of exact code clones. Indeed, we found differences between single-author and multi-author clones. Nevertheless, our results do not indicate that the involvement of more than one developer is a typical cause of unwanted inconsistencies and clone-related bugs. The differences we have found are rather directed to the rationale of cloning and the changes applied to the clones. The multi-author clones in the analyzed systems seem to have an even reduced risk potential, because the inconsistent development is intended or changes are mostly directed to style issues and do not affect semantics. These first results, however, certainly require to be further confirmed by future research.

Because of the absence of a strong relation between the number of authors of a clone and the probability of unintended inconsistent changes and bugs, we cannot recommend to use the number of authors of a clone as a metric to prioritize clone management activity.

## ACKNOWLEDGMENTS

We would like to thank Marcel Steinbeck for his support on implementing technical features of our analysis. This work was supported by a research grant of the *Deutsche Forschungsgemeinschaft* (DFG).

## REFERENCES

- [1] M. Balint, R. Marinescu, and T. Girba, "How Developers Copy," in *International Conference on Program Comprehension*, Jun. 2006, pp. 56–68.
- [2] D. Cai and M. Kim, "An empirical study of long-lived code clones," in *International Conference on Fundamental Approaches to Software Engineering*, 2011, pp. 432–446.
- [3] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *International Conference on Software Engineering*, 2012, pp. 255–265.
- [4] J. Harder and N. Göde, "Quo vadis, clone management?" in *International Workshop on Software Clones*, 2010, pp. 85–86.
- [5] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, no. 06301, 2007.
- [6] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.
- [7] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: a systematic review," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 261–283, 2013.
- [8] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2005, pp. 187–196.
- [9] N. Göde, "Evolution of type-1 clones," in *International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 77–86.
- [10] J. Krinke, "Is cloned code more stable than non-cloned code?" in *International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 57–66.
- [11] J. Harder and N. Göde, "Cloned code: stable code," *Journal of Software: Evolution and Process*, 2012, published online.
- [12] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software," in *Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010, pp. 73–82.
- [13] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *SIGAPP Applied Computing Review*, vol. 12, no. 3, pp. 20–36, Sep. 2012.
- [14] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Working Conference on Reverse Engineering*, 2007, pp. 170–178.
- [15] S. Bazrafshan, "Evolution of near-miss clones," in *Conference on Source Code Analysis and Manipulation*, 2012, pp. 74–83.
- [16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *International Conference on Software Engineering*, 2009, pp. 485–495.
- [17] J. Harder and R. Tiarks, "A controlled experiment on software clones," in *International Conference on Program Comprehension*, 2012, pp. 219–228.
- [18] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Working Conference on Mining Software Repositories*, 2010, pp. 72–81.
- [19] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *International Conference on Software Engineering*, 2011, pp. 311–320.
- [20] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.
- [21] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, Oct. 2008.
- [22] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Ownership on Software Quality," in *European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, 2011, pp. 4–14.
- [23] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *International Workshop on Principles of Software Evolution*, 2005, pp. 113–122.
- [24] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *International Conference on Software Engineering*, 2011, pp. 491–500.
- [25] N. Göde and R. Koschke, "Incremental clone detection," in *European Conference on Software Maintenance and Reengineering*, 2009, pp. 219–228.
- [26] J. Harder and N. Göde, "Efficiently handling clone data: RCF and Cyclone," in *International Workshop on Software Clones*, 2011, pp. 81–82.

# Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules

Wenyi Qian\*, Xin Peng\*, Zhenchang Xing†, Stan Jarzabek‡, and Wenyun Zhao\*

\*Software School, Fudan University, Shanghai, China

Email: {11212010025, pengxin, wyzhao}@fudan.edu.cn

†School of Computer Engineering, Nanyang Technological University, Singapore

Email: zcxing@ntu.edu.sg

‡School of Computing, National University of Singapore, Singapore

Email: dcssj@nus.edu.sg

**Abstract**—Software systems contain many implicit application-specific business and programming rules. These rules represent high-level logical structures and processes for application-specific business and programming concerns. They are crucial for program understanding, consistent evolution, and systematic reuse. However, existing pattern mining and analysis approaches cannot effectively mine such application-specific rules. In this paper, we present an approach for mining logical clones in software that reveal high-level business and programming rules. Our approach extracts a program model from source code, and enriches the program model with code clone information, functional clusters (i.e., a set of methods dealing with similar topics or concerns), and abstract entity classes (representing sibling entity classes). It then analyzes the enriched program model for mining recurring logical structures as logical clones. We have implemented our approach in a tool called MiLoCo (Mining Logical Clone) and conducted a case study with an open-source ERP and CRM software. Our results show that MiLoCo can identify meaningful and useful logical clones for program understanding, evolution and reuse.

**Keywords**—logical clone; semantic clustering; program comprehension; evolution; reuse;

## I. INTRODUCTION

The development and maintenance of a software system often follows many business and programming rules. Take the diagram in Figure 1 as an example. This diagram represents a logical business and programming rule implemented in a publication management system. It consists of various kinds of code entities such as methods, code clone sets, entity classes, functional clusters, and invoke/contain/access relationships between these code entities.

This rule shows business process as well as program convention for adding a publication into the system. The system currently deals with three types of publications, i.e., conference papers, journal articles, and books. To add a specific publication, the system invokes the corresponding add operation of `PublicationProcess`. The add operation first accesses information from the `Publication` entity object representing the publication to be added. It then invokes `User.checkAuthority()` to examine if the user has the access right to add a new publication. After that, the add operation checks the integrity of publication information by invoking the corresponding check operation of `PublicationProcess`. Next, it invokes

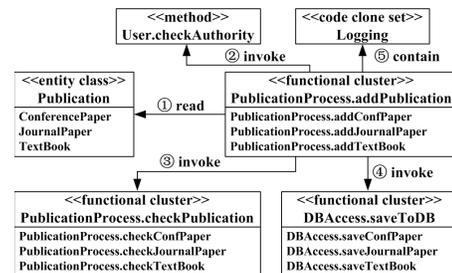


Fig. 1. An Example of Logical Clone

`DBAccess.saveToDB()` to store publication information into database. Finally, the add operation logs the transaction.

Such logical rules are crucial for program understanding, consistent evolution, and systematic reuse. For example, suppose the system needs to be extended with a new kind of publication (e.g., CD). Given the above logical rule, a developer is clear about what he needs to do. He will introduce a new `Publication` entity class, a new add and check operation in `PublicationProcess` for CD, and also a new save operation to `DBAccess`. He can duplicate logging code from existing add operations. Furthermore, he can also learn the proper business process for adding a CD from the ordering of operations captured by the logical rule. Without this explicit logical rule that can be followed, the developer may face the risk of introducing inconsistent evolution or even bugs. For example, he may forget to check information integrity before storing a publication into database.

This logical rule may also provide useful hints for reengineering decisions because it provides a high-level overview of several related business and entity objects. For example, a developer may consider moving information integrity checking operation from `PublicationProcess` to corresponding `Publication` entity classes, because these checking operations suffer from feature envy [1] in the overall business process.

In spite of the importance of logical business and programming rules, they are often not well documented during software development and maintenance. Instead, they are only implicitly present in system implementation. Making such

rules explicit and understood by developers improves developers' efficiency in software maintenance and also quality of the software system. However, detecting business and programming rules such as the one in Figure 1 poses unique challenges to existing pattern mining and analysis approaches.

The application of design patterns [2] may result in similar design structures. As design patterns provide clear descriptions about static structures and dynamic behaviors of the documented patterns, certain structural or behavioral templates [3], [4] can be defined to identify design structures following design patterns. However, implicit business and programming rules are application-specific. They are essentially open-ended and emerging from system implementation. No templates can be predefined. Furthermore, design patterns describe good design in an abstract manner. Detecting instances of design patterns is not concerned with application-specific semantics. In contrast, business and program rules reflect application-specific semantics, for example the meaning of a given method invocation sequence.

Software clones refer to similar code fragments or code structures. Business and programming rules are implemented as high-level similar structures and processes across several classes and methods. They are at a much higher level of abstraction than code clones and their instances may consist of one or more code clones (such as `Logging` in our example). Structural clones [5] can reveal high-level duplications in a system which may consist of multiple cross-cutting simple code clones. However, the elements in a business or programming rule that deal with the same or similar business concerns may not be similar enough to be detected as structural clones. For example, the three `add` operations of `PublicationProcess` are very different in implementation because they deal with different kind of publication. However, they share similar topics and play the same role in the business processes of the publication management system.

In this paper, we present an automatic approach for mining implicit business and programming rules in a software system. We call mined rules logical clones. A logical clone represents a high-level logical structure and process for an application-specific business or programming concern. Our approach takes as input source code of a software system. It mines logical clones in two steps. First, it identifies various kinds of similar program elements. Our approach analyzes methods, entity classes, and persistent data objects. It groups similar methods into functional clusters using semantic clustering technique [6], and detects similar code fragments inside methods (i.e., code clones) using clone detectors (such as Simian [7]). The first step generates a large graph whose nodes represent methods, entity classes, persistent data objects, and the identified functional clusters and code clones, and whose edges represent `invoke/access/contain` relations between program elements. Next, our approach uses subgraph pattern mining technique to mine logical clones in the system. The mined logical clones consist of functional clusters (i.e., a set of similar methods), single methods, code clone sets, entity classes (concrete or abstract), persistent data objects, and their

`invoke/access/contain` relations (see Figure 1 for an example).

We have implemented our approach in a tool called MiLoCo (Mining Logical Clone). To evaluate the effectiveness and usefulness of logical clones that MiLoCo reports, we conducted a case study with Opentaps [8], an open-source Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) software. In this study, MiLoCo reported 1,690 logical clones that involve more than 24% classes and 5% methods in the subject system. These logical clones reveal recurring program conventions, design structures, business tasks, and business processes in Opentaps. We also surveyed five experienced developers about the usefulness of the mined logic clones. Our survey suggests that most of the mined logical clones were classified as meaningful and useful for program understanding and evolution.

The remainder of the paper is organized as follows. Section II presents some related work and compares them with ours. Section III describes the proposed approach for mining logical clones. Section IV presents the supporting tool MiLoCo. Section V reports the evaluation based on the case study with Opentaps. Section VI discusses some related issues. Finally, Section VII concludes the paper and outlines the future work.

## II. RELATED WORK

Related work of our research spans five aspects, i.e., simple clone detection, high-level clone detection, design pattern detection, model clone detection, and API usage pattern detection.

Simple clone means similar or identical code fragments reflecting duplication at the code level. They can be introduced by copy-paste-modify practice, implementation of similar requirements, or following usage specification of APIs. Simple clone detection has been a well-researched area in recent years and a number of clone detection techniques have been proposed. These techniques detect simple code clones by analyzing program text [9], program tokens [10], and code metrics [11]. Marcus et al. [12] proposed an approach that uses an information retrieval technique (i.e., latent semantic indexing) to identify identify implementations of similar high-level concepts (e.g., abstract data types). Some techniques have been proposed to detect clones by comparing Abstract Syntax Tree (AST) [13], or Program Dependency Graph (PDG) [14]. These techniques can find code fragments that are lexically different but structurally similar. However, clones being reported are still at the code level within methods.

Different from simple code clones, structural clones [5], [15] are larger duplicated program structures consisting of multiple fragments of duplicated code. Such high-level clones can also represent important domain or design concepts. Basit et al. [5] proposed a data mining approach to detect structural clones at different levels, including method clones, file clones and directory clones. Their follow-up study [15] showed that over 50% of simple clones in the subject systems can be captured by structural clones. Structural clones are recurring configurations of simple clones. In contrast, logical clone detection combines several techniques, such as simple clone

detection, semantic clustering and so on, to get more complex clones. Besides simple clones, the elements in logical clone can also be methods sharing similar topics or entity classes inherited from the same superclass. Furthermore, the elements in a logical clone can be distributed in different classes and connected by invocation and access relations.

Some researchers focus on detecting similar design structures caused by application of design patterns. Tsantalis et al. [3] proposed an approach that uses the similarity score between graph representation of a system and a design pattern to detect instances of design patterns. Romano et al. [4] presented an approach that applies text clustering on classes of a system and then uses existing tools such as DPR [16] and Pattern4 [3] to identify design pattern instances based on obtained clusters. These design pattern detection approaches are usually based on predefined descriptions about the structure and behaviors of a design pattern, while logical clones represent application-specific business and programming rules that are implicitly applied in system implementation. Furthermore, design pattern detection is not concerned with application-specific semantics. In contrast, logical clones reveal application-specific semantics, for example what a set of similar methods do.

Model clone detection is focused on detecting similar or identical fragments in software models. Alalfi et al. [17] proposed an approach to identify structurally meaningful subsystem clones in graphical models such as Simulink models. Different from model clone detection, our approach on logical clone detection is concerned with application-specific semantics such as topics of program elements.

In recent years, there has been some research [18], [19], [20] focusing on detecting API usage patterns, which reflect similar API call sequences and pre-conditions when using a set of APIs. Compared with API usage patterns, logical clones reveal high-level system structures involving elements from multiple classes and richer structural relations such as accessing entity classes in addition to invoking APIs. Moreover, logical clone detection can identify invocations of similar methods rather than invocations to exactly the same APIs.

### III. APPROACH

In this section, we first give an overview of our approach for mining logical clones. Figure 2 shows an overview of our approach. Our approach takes as input source code of a software system and reports a set of logical clones. It consists of two steps, i.e., model extraction and graph mining. Model extraction is to build a program model from source code for the subsequent graph mining step. The initial program model consists of methods, entity classes, persistent data objects, and their invoke/access relations. Similar methods are then clustered into functional clusters, code clones are detected by clone detection tools, and sibling entity-classes are grouped by their inheritance relations. Based on the extracted program model, a subgraph pattern mining algorithm is used to mine logical clones. This mining algorithm first generates initial logical clones with only one functional cluster and then

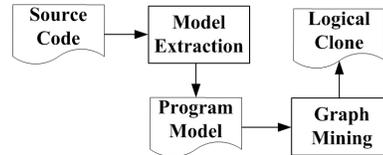


Fig. 2. Approach Overview

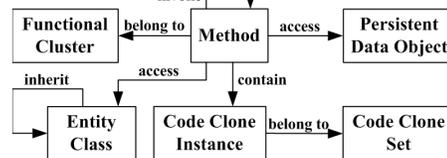


Fig. 3. Metamodel of Extracted Program Models

incrementally extends current logical clones by including one neighboring node at a time.

#### A. Model Extraction

Figure 3 presents the metamodel of program model being extracted in our approach. The program model is a directed graph. The initial program model contains methods, entity classes, and persistent objects. A method can invoke other methods, and access (create, read, or update) entity classes and persistent data objects. An entity class can inherit other entity class. Given the initial program model, individual methods that share similar topics can be grouped into functional clusters based on their lexical descriptions using semantic clustering [6]. The cloned code fragments (i.e., clone instances) in methods are identified by a clone detector (such as Simian [7]). Clone detector usually reports code clone instances in clone sets. Sibling entity classes that are the descendant classes of the same classes are grouped together as an abstract entity class.

1) *Methods and Functional Clusters*: Methods of a software system and their invocation relations can be easily extracted from source code by static analysis. We exclude simple getter and setter methods from the program model, because our logical clone analysis is focused on meaningful operations for high-level business or programming concerns. However, it is important to note that we exploit getter and setter methods to discover data access relations between methods and entity classes. For example, we consider that a method reads information from an entity class if the method invokes getter methods of the entity class.

To reveal application-specific topics of several methods, we use semantic clustering [6] to group methods that share similar vocabulary into functional clusters. Our approach first transforms all the methods in the program model into a corpus of documents for clustering. For each method, its method name (including package and class name) and parameter names are used together as document title. Method body is transformed into content of document. The identifier names and comments in method body are extracted and transformed by standard Information Retrieval (IR) preprocessing steps including tokenization, stop-words filtering and word stemming.

Given the generated corpus of documents, our approach then generates a Term-Frequency/Inverse-Document-Frequency (T-F/IDF) vector for each method document. It uses a bisecting K-means clustering algorithm to group generated document vectors into different clusters. Bisecting K-means clustering [21] is an extension of K-means clustering, which does not require a predefined K value. We implemented a bisecting K-means clustering algorithm based on the K-means clustering algorithm provided by Weka [22] (see Algorithm 1).

The algorithm takes as input a set of document vectors  $V$  and produces a set of functional clusters. It performs an iterative clustering process with a queue initialized with  $V$ . In each iteration, it dequeues a set  $V'$  of document vectors from the queue and divides it into two subsets (i.e., clusters) by K-means clustering with  $K=2$ . For each of produced clusters, we use  $Diff$  function (see the following formula) to measure the overall difference between the document vectors in the cluster and the mean vector of the cluster, where  $distance$  function returns Euclidean distance between a vector  $x_i$  in the cluster and the mean vector  $\bar{x}$ .

$$Diff(Cluster) = \sqrt{\frac{1}{|Cluster|} \sum_{i=1}^{|Cluster|} distance(x_i, \bar{x})^2}$$

If this overall difference of the cluster is higher than a given threshold  $\delta$ , it is added to the queue for further division. Otherwise, it is added to  $Clusters$  as a functional cluster. The iterative process ends when the queue is empty.

---

**Algorithm 1** Bisecting K-means Clustering Algorithm

---

```

1: function BISECTINGCLUSTERING( $V$ )
2:    $Clusters = [ ], queue = [ ]$ 
3:    $queue.enqueue(V)$ 
4:   while  $queue.length > 0$  do
5:      $V' = queue.dequeue$ 
6:      $C = KmeansClustering(V', 2)$ 
7:     for  $(i = 1; i \leq 2; i++)$  do
8:       if  $Diff(C[i]) > \delta$  then
9:          $queue.enqueue(C[i])$ 
10:      else
11:        add  $C[i]$  to  $Clusters$ 
12:      end if
13:    end for
14:  end while
15:  return  $Clusters$ 
16: end function

```

---

2) *Entity Classes*: An entity class encapsulates information of a business entity and corresponding getter and setter methods. The access (i.e., create, read, update) relations between a method and an entity class can be identified as follows. If a method invokes constructors of an entity class, it creates instances of the entity class. If a method reads properties or invokes getter methods of an entity class, it reads information from the entity class. If a method modifies properties or invokes setter methods of an entity class, it updates the entity class.

Inheritance relations between entity classes are also extracted by static analysis. We consider entity classes inheriting from the same superclass defined in the system as playing the same role in a logical clone.

Access relations between a method and an entity class will be generalized as access relations between the method and the abstract entity class. In our running example, three `Publication` subclasses are grouped into an abstract entity class representing `Publication` entity classes. The read relations from different add operations to sibling entity classes `ConferencePaper`, `JournalPaper`, and `TextBook` can be generalized as a read relation to the abstract entity class `Publication`.

3) *Code Clones*: In our approach we detect code clones using existing clone detection tools. Our current implementation uses text-based tool (e.g., Simian) for clone detection. Clone detectors usually report code clones in clone sets. Each clone set contains two or more code clone instances (i.e., similar code fragments) in several methods. For example, three add operations of `PublicationProcess` contain code clones in a code clone set. These cloned code fragments implement the same feature, i.e., logging transaction for adding publication.

4) *Persistent Data Objects*: Persistent data objects represent data tables in database or data entries in files (e.g., XML files). They can usually be identified from data dictionaries or data schemas. However, different applications may use different methods to access persistent data objects. For example, in some applications, methods may directly access database tables with SQL queries. In other applications, database access may be implemented using Object-Relational (O/R) mapping frameworks (such as Hibernate [23]). Therefore, access relations between methods and persistent data objects need to be analyzed depending on application-specific persistent data access strategy. For example, for the first case mentioned above, we can intercept SQL queries issued in each method to determine persistent data objects that the method accesses. For the second case, we can analyze XML configuration files of O/R mapping frameworks.

### B. Subgraph Pattern Mining

Figure 4 shows metamodel of logical clones mined using our subgraph pattern mining algorithm. A logical clone is represented as a graph that consists of functional clusters that invoke other functional clusters and/or methods, access entity classes (concrete or abstract classes), and contain code clone sets. For example, the logical clone shown in Figure 1 consists of six graph nodes. The functional cluster `PublicationProcess.addPublication` consists of three similar add operations. It reads abstract entity class `Publication` that consists of three sibling subclasses of `Publication`. It invokes method `User.checkAuthority` and two other functional clusters `PublicationProcess.checkPublication` and `DBAccess.saveToDB`. Finally, it logs transaction using cloned code fragments found in all add operations.

Based on the program model produced by model extraction step, we then use a subgraph pattern mining algorithm to detect logical clones (see Algorithm 2). The algorithm takes as input a program model *Model*, which includes a set of functional

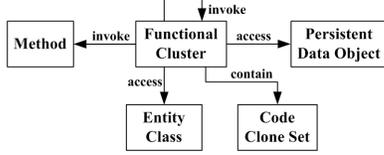


Fig. 4. Metamodel of Logical Clones

clusters  $FCSet$ , a set of methods  $MESet$ , a set of entity classes  $ECSet$ , a set of code clone instances  $CISet$ , a set of code clone sets  $CSSet$ , and a set of persistent data objects  $DOSet$ . It produces a set of logical clones, each of which can be represented as a graph as defined by the logical clone metamodel (see Figure 4).

The algorithm first generates an initial set of logical clones that consists of only one node and then incrementally extends existing logical clones by including neighboring nodes one at a time. Two thresholds are used to eliminate logical clones with too few nodes (lower than  $threshold_{node}$ ) or too few instances (lower than  $threshold_{instance}$ ). An instance of a logical clone is a program structure consisting of methods, code clone instances, entity classes, persistent data objects, each of which is an instance of the corresponding node in the logical clone.

The algorithm first initiates a set of logical clones with only one node (Line 3-9). For each functional cluster with the number of methods greater than  $threshold_{instance}$ ,  $newLCNode$  function creates a logical clone node  $node$  using all of the methods in the functional cluster as the instances of the logical clone node. This logical clone node is then added to a queue.

Next, an incremental analysis process is conducted until the queue is empty (Line 10-25). In each iteration, the algorithm dequeues a candidate logical clone  $candLC$  and considers possible extensions to it. Each functional cluster in  $candLC$  is considered as an extension point (Line 13-21). For each extension point,  $Extend$  function (see Algorithm 3) generates a set of extended logical clones based on  $candLC$ . These extended logical clones are added to the queue for further extension.

For each  $candLC$ , a flag is used to indicate whether it is contained by another logical clone extended from it. The  $contain$  function returns whether an extended logical clone  $newLC$  contains  $candLC$ , i.e., all instances of  $candLC$  are contained by the instances of the extended logical clone  $newLC$ . If the flag is false (i.e.,  $candLC$  cannot be contained by any extended logical clones) and the size of  $candLC$  (i.e., the number of nodes in the candidate logical clone) is greater than  $threshold_{node}$ , this  $candLC$  is added to  $LCSet$  as a detected logical clone (Line 22-24).

$Extend$  function (see Algorithm 3) generates a set of extended logical clones based on a given logical clone  $candLC$ . Each of the generated logical clone extends the original  $candLC$  with one more node from the given extension point  $node$ . The newly included node  $el$  can be: a method that some instances (i.e., methods belonging to the functional cluster) of  $node$  invoke (Line 4-6); a functional cluster that some

## Algorithm 2 Logical Clone Mining Algorithm

```

1: function LOGICALCLONEDetect( $Model$ )
2:    $LCSet = \{ \}, queue = [ ]$ 
3:   for each  $fc \in Model.FCSet$  do
4:      $methods = \{m | m \in Model.MESet \wedge m \text{ belong to } fc\}$ 
5:     if  $methods.size \geq threshold_{instance}$  then
6:        $node = newLCNode(fc, methods)$ 
7:        $queue.enqueue(\{node\})$ 
8:     end if
9:   end for
10:  while  $queue.length > 0$  do
11:     $candLC = queue.dequeue()$ 
12:     $flag = False$ 
13:    for each  $node \in candLC \wedge node.type = FC$  do
14:       $newLCS = Extend(Model, candLC, node)$ 
15:      for each  $newLC \in newLCS$  do
16:         $queue.enqueue(newLC)$ 
17:        if  $contain(newLC, candLC)$  then
18:           $flag = True$ 
19:        end if
20:      end for
21:    end for
22:    if  $\neg flag \wedge candLC.size \geq threshold_{node}$  then
23:       $LCSet = LCSet \cup candLC$ 
24:    end if
25:  end while
26:  return  $LCSet$ 
27: end function
  
```

instances of  $node$  invoke (Line 7-9); an entity class whose subclasses (including itself) are accessed by some instances of  $node$  (Line 10-12); a persistent data object that some instances of  $node$  access (Line 13-15); a code clone set whose clone instances are contained in some instances of  $node$  (Line 16-18). Note that not all the instances of the extension point  $node$  are required to have certain types of relations with  $el$ . Therefore, the instances of a logical clone extended from  $candLC$  may be less than those of  $candLC$ . Only those extended logical clones whose instance numbers are greater than  $threshold_{instance}$  are returned (Line 19-22).

### C. Representation of Logical Clones

To help developers understand mined logical clones, we generate meaningful labels and natural language summary for each logical clone.

1) *Labeling*: Our approach generates a meaningful label for each node of a logical clone using different strategies for different kinds of nodes. For method node or persistent data object node, method name or object name is used as its label. For functional cluster node, a label is generalized from names of all the methods in the cluster. Our approach splits method names into word sequences. It keeps longest common subsequence of method names and replace differences in method names using wildcards. For example, the names of two methods  $updateImportServices$  and  $updateExportServices$  can be generalized to a label  $update * Services$ , which can be interpreted as “update different kinds of services”. For code clone set node, a set of tags are generated as its label using

---

**Algorithm 3** Logical Clone Extension Algorithm

---

```
1: function EXTEND(Model, candLC, node)
2:   ExLCSet = { }, queue = [ ]
3:   for each el ∈ Model.elements ∧ el ∉ candLC do
4:     if el ∈ Model.MESet then
5:       INS = {n | n ∈ node.instances ∧ n invoke el}
6:     end if
7:     if el ∈ Model.FCSet then
8:       INS = {n | n ∈ node.instances ∧ ∃m ∈ {m' | m' ∈
9:         Model.MESet ∧ m' belong to el ∧ n invoke m'}}
10:    end if
11:    if el ∈ Model.ECSet then
12:      INS = {n | n ∈ node.instances ∧ ∃m ∈ {m' | m' ∈
13:        Model.ECSet ∧ m' inherit el ∧ n access m'}}
14:    end if
15:    if el ∈ Model.DOSet then
16:      INS = {n | n ∈ node.instances ∧ n access el}
17:    end if
18:    if el ∈ Model.CSSet then
19:      INS = {n | n ∈ node.instances ∧ ∃m ∈ {m' | m' ∈
20:        Model.CISet ∧ m' belong to el ∧ n contain m'}}
21:    end if
22:    if INS.size ≥ thresholdinstance then
23:      newLC = candLC ∪ {el}
24:      ExLCSet = ExLCSet ∪ {newLC}
25:    end if
26:  end for
27:  return ExLCSet
28: end function
```

---

topic mining techniques. For entity-class node, the name of the common superclass that all entity classes inherit is used as its label. Furthermore, our approach generates labels for different types of data access relations (i.e., create, read or update).

To partially reflect the sequence of method invocations and data accesses, our approach produces numbers indicating the order of invoke/access by analyzing the position of the invoke/access relations in source code. As a functional cluster node can have multiple instances, the ordering number is only produced when the orders in all its instances are consistent.

2) *Topics and Natural Language Summary*: For each logical clone, our approach generates a set of topics from source code of all its instances using topic mining techniques. Furthermore, it generates a natural language summary based on a template defined according to the logical clone metamodel. This natural language summary provides an overview of the logical clone. For example, our approach generates a natural language summary for the logical clone shown in Figure 1 as follows.

*The addPublication method, read information of Publication entities, invoke User.method, invoke checkPublication methods, invoke saveToDB methods, contain clone for Logging.*

#### IV. TOOL IMPLEMENTATION

We have implemented our approach in a tool called MiLoCo (Mining Logical Clone). Figure 5 shows a screen shot of the tool as it has been used to analyze logical clones in Opentaps [8] in our case study.

MiLoCo currently supports logical clone analysis in Java-based software systems. Our current implementation uses Simian [7], a text-based clone detector, for clone detection in code. In our current implementation of bisecting K-means clustering algorithm, we set the maximum number of iterations to be executed to 30 in K-means.

MiLoCo displays mined logical clones in a list view (the upper part). The list can be sorted by the number of nodes in logical clones or the number of instances in logical clones. For each logical clone, the list shows its ID (a unique identifier generated by our approach), the number of nodes, the number of instances, topics, and labels of the logical clone.

A user can inspect the details of a logical clone by double-clicking the logical clone in the list. The detail panel (the lower part) provides three views for inspecting logical clone, i.e., summary view on the left, graph model view in the middle, and source code view on the right. The summary view provides natural language description of the selected logical clone and its basic information such as topics, the number of nodes, and the number of instances. The graph model view shows nodes of the logical clone and their relations. For example, the logical clone shown in Figure 5 contains eight graph nodes. Different kinds of nodes are shown in different colors, for example, blue nodes for functional clusters, red nodes for code clone sets, pink nodes for persistent data objects, green nodes for entity classes. For each functional cluster or code clone set node, the user can expand or shrink the node to inspect its instances in the cluster or clone set by double clicking the node on the graph model. The source code view shows source code of all the instances of a node. When a node is selected on the graph model, the source code of all its instances is shown in different tab views in source code view. For example, the source code view in Figure 5 shows the source code of all the instances of a code clone set. The yellow highlight shows cloned code clone fragments.

#### V. EVALUATION

To evaluate whether the proposed approach and MiLoCo can identify meaningful logical clones that are useful for program understanding, evolution, and reuse, we conducted a case study with Opentaps [8] to investigate the following three research questions:

- RQ1 What kinds of logical clones can be mined from the system? What knowledge do these logical clones represent?
- RQ2 How well does MiLoCo mine high-level clones compared with existing approaches?
- RQ3 How do developers evaluate mined logical clones in terms of their usefulness for program understanding, evolution and reuse?

In the following subsections, we first introduce the basic results of the case study and then present our answers to the three research questions.

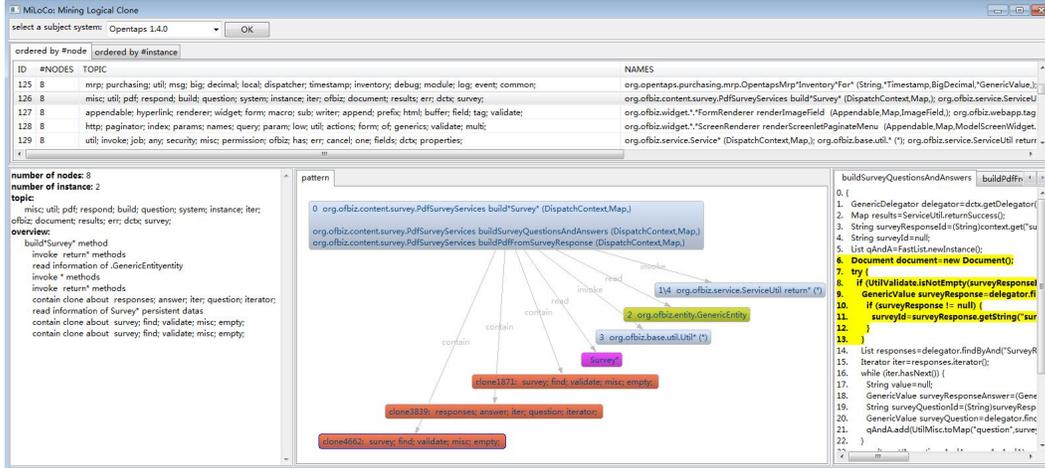


Fig. 5. MiLoCo Layout

### A. Basic Results

Optentaps is a large open-source ERP and CRM system developed in Java. The subject system we used was Optentaps 1.4.0, which has 14,351 classes and interfaces, 253,743 methods, and about three million lines of code. The whole logical clone mining process, including model extraction, subgraph pattern mining and representation, took about four hours and no more than 170MB disk space with a 2.9GHz dual-core CPU and 8GB RAM.

In the case study, MiLoCo reports 1,690 logical clones with the settings of  $threshold_{node} = 3$  and  $threshold_{instance} = 2$ , i.e., having at least three nodes and two instances. These logical clones involve 3,553 (24.8%) classes and interfaces and 14,053 (5.5%) methods of the system.

These 1,690 logical clones contain 3 to 38 nodes (11.2 on average) and 2 to 2020 instances (7.9 on average). The distribution of logical clones with different numbers of nodes and instances is shown in Figure 6 and Figure 7.

All 1,690 logical clones include 19,005 nodes in total, out of which 5,760 are functional clusters (30.3%), 955 are methods (5.0%), 7,527 are code clone sets (39.6%), 4,681 are entity classes (24.6%), 82 are persistent data objects (0.5%).

After analyzing the results, we found that a large part (1,050) of the detected logical clones have very similar structures as shown in Figure 8. These logical clones reflect the programming convention of creating new service instances: a service instance is created, and then the service user is obtained from input and set to the created service instance. Therefore, we merged these similar logical clones into one and obtained 641 logical clones for further analysis and evaluation.

### B. RQ1: Categories of Mined Logical Clones

Having analyzed all 641 logical clones, we found that they can be categorized into the following four types based on abstraction level, scope and complexity of duplicated logical structures. The former two types (i.e., programming convention and design structure) reflect duplicated logical structures

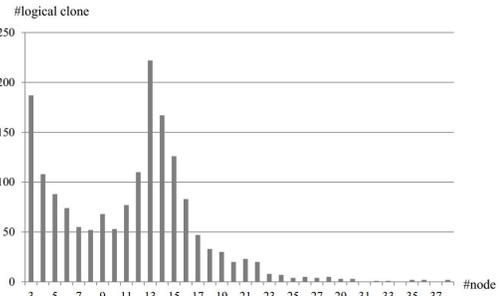


Fig. 6. Distribution of Logical Clones with Different Numbers of Nodes

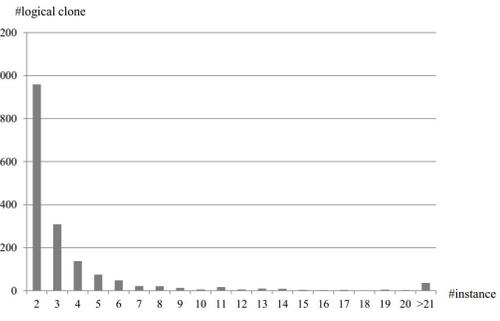


Fig. 7. Distribution of Logical Clones with Different Numbers of Instances

on technical issues. The latter two types (i.e., business task and business process) reflect duplicated logical structures on business issues. Due to the limitation of space, the examples used in this paper may only show a part of logical clone nodes.

1) *Programming Convention*: A programming convention conveys similar ways of using a set of related operations when implementing similar functions. It is similar to API usage pattern, but may involve multiple classes and similar methods playing the same role. An example of programming convention is shown in Figure 8. In this example, *fromInputWith \* Service* method in a *Service* class gets a *Service* instance as input and invokes *inputMap* method to transfer data of the

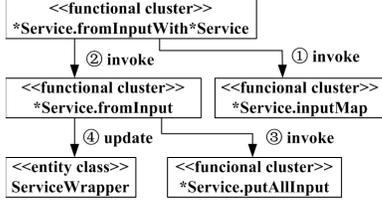


Fig. 8. An Example Logical Clone of Programming Convention

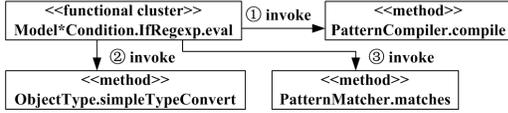


Fig. 9. An Example Logical Clone of Design Structure

instance into a *Map*, and then invokes *fromInput* method to process *Map*, including storing the data by *putAllInput* and updating the entity class *ServiceWrapper*. This specific logical clone conveys how to use a set of operations to process a service instance.

2) *Design Structure*: A design structure reflects similar interaction structures in different parts of a system. It differs from programming convention in that it involves more complex interaction structures among multiple classes. An example of design structure is shown in Figure 9. This example indicates the interaction structure among four classes, i.e., *Model\*Condition*, *ObjectType*, *PatternMatcher*, *PatternCompiler*, for the purpose of evaluating a regular expression. Furthermore, logical clones representing program conventions appear more frequently than those representing design structures.

3) *Business Task*: A business task conveys similar ways of using a set of related operations when implementing similar business tasks. An example of business task is shown in Figure 10. This example tells how to combine a set of operations to implement the tasks of showing some information on POS (Point Of Sells) screen: it first invokes a *PosScreen.show\*WithString* method, and this method in turn reads information from *PosScreen* and invokes another method to *refresh* the *PosScreen* or *showDialog* on *PosDialog*.

4) *Business Process*: A business process reflects similar business processes or sub-processes in different parts of a system. It differs from business task in that it involves multiple business tasks or steps. An example of business process is shown in Figure 11. This example reflects the steps involved in POS logout or shutdown: it first reads information from *PosScreen*, then shows *PosScreen*, and finally invokes *PosTransaction.closeTx* to close POS transaction.

The distribution of different types of logical clones is shown in Figure 12. Among all the detected logical clones, 37% are programming conventions, 24% are design structures, 23% are business tasks, 16% are business processes.

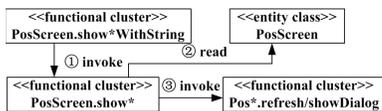


Fig. 10. An Example Logical Clone of Business Task

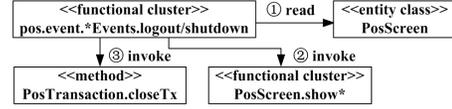


Fig. 11. An Example Logical Clone of Business Process

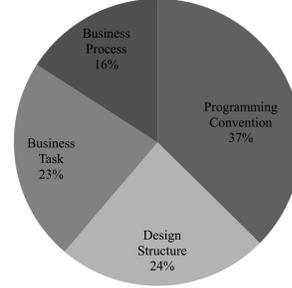


Fig. 12. Distribution of Different Types of Logical Clones

In summary, logical clones detected by our approach can reveal software maintenance knowledge spanning different levels of abstraction, including programming convention, design structure, business task, and business process.

### C. RQ2: Mining High-Level Clones

To evaluate the capability of our approach in mining high-level clones, we compared the logical clones detected by MiLoCo and the structural clones detected by CloneMiner. CloneMiner [5] is a structural clone detection tool developed based on token-based simple clone detector. It can detect structural clones at the method, file, and directory levels. We compared the detected logical clones with file-level structural clones, since basic elements of both logical clones and structural clones are methods. In this comparative evaluation, we chose a package of Opentaps for analysis, which has 129 Java files and 812 methods. We configured CloneMiner with the following token threshold settings: simple clone 30, method clone 50, and file clone 50.

From the package, CloneMiner detected 15 file clones and MiLoCo detected 96 logical clones. It can be seen that MiLoCo can mine much more high-level clones, since it can use semantic clustering to identify methods that share similar topics but are not similar enough in their source code. For example, the two functional clusters of the logical clone shown in Figure 13 were not detected as method clones by CloneMiner, but they were grouped by semantic clustering. Moreover, MiLoCo can identify logical clones with various kinds of nodes distributed in different classes (files).

For the 15 file clones detected by CloneMiner, we found that 7 of them can be roughly covered by the detected logical clones. All the method clones in these file clones are included in the detected logical clones, but may distribute in different logical clones. For example, two source files *RemoveList*

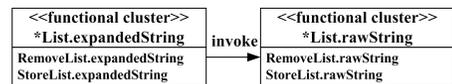


Fig. 13. Functional Clusters not Detected as Method Clones by CloneMiner

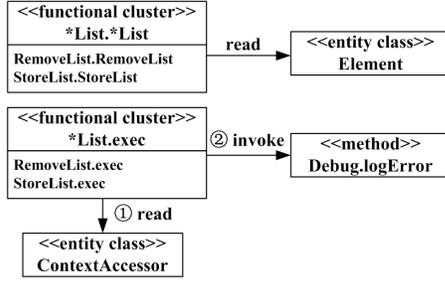


Fig. 14. Method Clones Distributed in Different Logical Clones

and *StoreList* were detected as file clones with two method clones, i.e., constructor and *exec* method. These two sets of similar methods distribute in two different logical clones shown in Figure 14. These logical clones together provide more information about duplicated logical structures (e.g., accessing entity classes *Element* and *ContextAccessor*, invoking method *Debug.logError*) than the detected file clones. For the other 8 file clones, some of their method clones were not covered by the detected logical clones.

In summary, MiLoCo can detect much more high-level clones than CloneMiner, but in some cases method clones detected by CloneMiner cannot be identified by MiLoCo. Therefore, it may be helpful to integrate method clones detected by CloneMiner as another kind of duplicated elements in program models. In fact, method clones can be easily integrated into logical clone mining process, because we only need to extend our metamodel to include not only simple code clones but also method clones.

#### D. RQ3: Developer Evaluation

This question concerns developers' evaluation on the usefulness of the detected logical clones. To answer this question, we selected five senior graduate students from our school of computer science as the developers for survey. These students have 3 to 5 years (3.8 years on average) of experience on Java development and experience of developing enterprise software. All of them described themselves as "Java experts" and had rich experience with ERP software.

Before the survey, we gave them a two-hour tutorial about background, business, and architecture of Opentaps. We also provided them with useful documents (e.g., user manual) obtained from Opentaps website. After the tutorial, the participants had a general understanding of the requirements and design of Opentaps. We then randomly selected 100 logical clones from all the 641 logical clones and surveyed the developers' evaluation on the usefulness of these logical clones. For each logical clone, the developers were asked to answer the following two questions:

*Programming understanding: Do you think this logical clone is meaningful and useful for you to understand the system?*

*Reuse/Evolution: Do you think it is helpful or even necessary to follow this business or programming rule when implementing similar functions or maintaining existing implementations?*

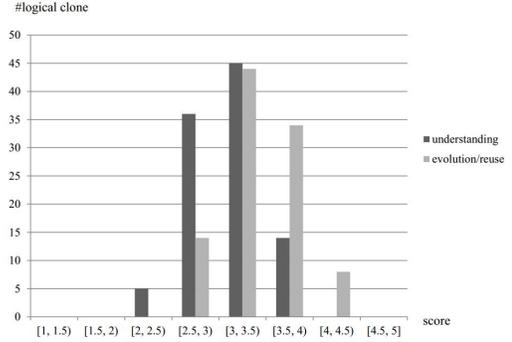


Fig. 15. Developers' Evaluation on the Logical Clones

For each question, the developers were asked to give a score from 1 to 5 (1 being useless/helpless, 3 being useful/helpful, and 5 being very useful/helpful). The survey was finished in three hours, during which the developers can check source code and documents of Opentaps to help them better understand the meaning of the logical clones.

The results of the survey are presented in Figure 15. It can be seen that most of the logical clones are thought to be useful and helpful ( $score \geq 3$ ) for program understanding (59%) and reuse/evolution (86%).

It is interesting to note that developers' answers to the second question are much more positive than their answers to the first question. After discussing the results with them, we found this reflects the real situation. Due to the lack of domain knowledge, they cannot fully understand those logical clones in a short time. Therefore, their answers to the first question (i.e., the usefulness for program understanding) were cautious. In contrast, they were more confident that these logical clones reflect programming and business rules that need to be followed when implementing similar functions or maintaining existing implementations. Therefore, their answers to the second question (i.e., the helpfulness for reuse and evolution) were more positive.

## VI. DISCUSSION

From Section V-D, it can be seen that 41 out of the 100 logical clones under evaluation were thought to be not so meaningful for program understanding ( $score < 3$ ). In contrast, only 14 out of the 100 logical clones were thought to be not so helpful ( $score < 3$ ) for software evolution and reuse. According to our discussion with the participants after the survey, the main reason that they evaluated a logical clone to be not so meaningful for program understanding is that they felt it hard to fully understand what logical clones reveal about the system. In contrast, they often thought a logical clone to be helpful for software evolution and reuse even though they could not fully understand it. Their explanation was that they can recognize the value of the revealed logical structures for consistent implementations for similar topics or concerns. Therefore, they thought that when implementing new similar topics or concerns or maintaining existing implementations it is necessary to follow the rules captured by the logical clones.

The main factors that caused insufficient understanding of some of the detected logical clones include lack of domain knowledge and less intuitive representation mechanisms for logical clones. The domain knowledge required for the understanding of logical clones include the meaning of business vocabulary and the understanding about business requirements and technical framework. Due to the lack of domain knowledge, the participants may feel it hard to understand the meaning of a logical clone even when some meaningful labels and tags are available. In some cases, they can realize the meaning of a logical clone soon after we explain the meaning of some logical clone nodes to them. Therefore, we think for professional developers with necessary domain knowledge it should be easier for them to understand the detected logical clones. For the problem of insufficient representation mechanisms, the UI (user interface) of MiLoCo can be improved from several aspects, including better visualization of logical clones, more intuitive summary, and more interactive navigation.

To make full use of the benefit of logical clones for software reuse and maintenance, it is necessary to integrate MiLoCo with IDEs (integrated development environments) like Eclipse and extend MiLoCo to support the management and reuse of logical clones in addition to mining and representation. With proper management mechanism, a developer can organize logical clones and their instances in a structured way, monitor their evolution, and discover potential problems. For example, when inconsistent changes to the instances of a logical clone are made, or a newly developed functionality does not follow the logical clones involved in the implementations of similar functionalities, the developer can be notified about the problem and take necessary actions. The reuse mechanism can be combined with code completion mechanisms supported by IDEs to generate skeleton code when the IDE detects that the developer is implementing a functionality that needs to follow certain logical clones.

## VII. CONCLUSION

In this paper, we have proposed the concept of logical clones, i.e., similar logical structures that involve similar or relevant concerns and topics but are not necessarily similar at the code level. We have presented an approach for mining logical clones. The approach first extracts from source code a program model consisting of methods, entity classes, persistent data objects, and invoke/access relations between them. To identify implicit duplications, we use semantic clustering to group methods into functional clusters and detect code clones using code clone detectors. Then the approach uses a graph mining algorithm to mine logical clones by detecting duplicated logical structures. Additional representation mechanisms are provided to help developers understand mined logical clones.

We have implemented our approach in a tool called MiLoCo to support logical clone mining and representation of logical clones. We conducted a case study with an open-source ERP and CRM system. The results show that MiLoCo can detect a rich set of logical clones spanning different levels, including programming convention, design structure, business task,

and business process. A developer evaluation shows that the detected logical clones are useful for program understanding, reuse, and maintenance.

In the future work, we will conduct more case studies with industrial and open-source systems of different types to discover more representative logical clones. We will further investigate how detected logical clones can be used by developers in software maintenance tasks and develop tools to support the management and usage of logical clones.

## ACKNOWLEDGMENT

This work is supported by National High Technology Development 863 Program of China under Grant No.2012AA011202, and by NTU SUG M4081029.020.

## REFERENCES

- [1] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *ICSM*, 2007, pp. 519–520.
- [2] R. J. J. V. Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [3] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, 2006.
- [4] S. Romano, G. Scanniello, M. Risi, and C. Gravino, "Clustering and lexical information support for the recovery of design pattern in source code," in *ICSM*, 2011, pp. 500–503.
- [5] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 497–514, 2009.
- [6] A. Kuhn, S. Ducasse, and T. Gırba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, 2007.
- [7] "Simian." [Online]. Available: <http://www.harukizaemon.com/simian/>
- [8] "Opentaps." [Online]. Available: <http://opentaps.org/>
- [9] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *ICSM*, 1999, pp. 109–118.
- [10] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.
- [11] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM*, 1996, pp. 244–253.
- [12] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *ASE*, 2001, pp. 107–114.
- [13] L. Jiang, G. Misherggi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.
- [14] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.
- [15] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in *ICSM*, 2012, pp. 275–284.
- [16] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177 – 1193, 2009.
- [17] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *ICSM*, 2012, pp. 295–304.
- [18] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending API usage patterns," in *ECOOP*, 2009, pp. 318–343.
- [19] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *ASE*, 2009, pp. 283–294.
- [20] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *ASE*, 2011, pp. 456–459.
- [21] G. Hamerly and C. Elkan, "Learning the k in k-means," in *NIPS*, 2003.
- [22] "Weka 3: Data mining software in java." [Online]. Available: <http://www.cs.waikato.ac.nz/ml/weka/>
- [23] "Hibernate." [Online]. Available: <http://www.hibernate.org/>

# An Empirical Study of Clone Removals

Saman Bazrafshan, Rainer Koschke  
 Software Engineering Group  
 University of Bremen, Germany  
 {saman.bazrafshan,koschke}@informatik.uni-bremen.de

**Abstract**—It is often claimed that duplicated source code is a threat to the maintainability of a software system and that developers should manage code duplication. A previous study analyzed the evolution of four software systems and found a remarkable discrepancy between code clones detected by a state-of-the-art clone detector and those deliberately removed by developers as the scope of the clones hardly ever matched. However, the results are based on a relatively small amount of data and need to be validated by a more extensive analysis. In this paper, we present an extension of this study by analyzing deliberate as well as accidental removals of code duplication in the evolution of eleven systems. Based on our findings, we could confirm the results of the previous study. Beyond that we found that accidental removals of cloned code occur slightly more often than deliberate removals and that many clone removals were in fact incomplete.

**Index Terms**—Clone removal, clone evolution, software maintenance

## I. INTRODUCTION

It is often claimed that duplicated source code—so called clones—increases the maintenance effort in software systems and, therefore, is harmful. This claim is based on the assumption that because of clones the source code becomes redundant, change effort increases, comprehensibility decreases, and inconsistent changes can introduce new defects or prevent the removal of existing ones. Nonetheless, recent studies [12], [19], [21], [27], [28] showed that clones cannot be considered a threat to software maintenance in general. *Copy&Paste* programming is a common practice to speed up software development, to reuse existing and reliable code, or to use existing code as starting point for new functionality [20], [25].

To support developers to keep track of and handle clones plenty of clone management tools have been introduced that detect clones [22], [30], support refactorings [4], [16] and change propagation [9], [34] as well as monitoring to prevent unwanted inconsistencies [29]. The efficiency of such tools has been successively improved over the years so that even very large systems can be efficiently analyzed. Still, clone management has not yet become an integral part of the daily work of programmers. A major hindrance to the use of clone management tools is a missing relevance ranking of clones. Because results provided by state-of-the-art clone tools are based on only structural similarity in the source code, users are exposed to a vast number of clone information. The amount of data increases the effort or makes it impractical to filter useful clones just manually. Therefore, clones that have been automatically detected need to be ranked by the relevance to a

specific maintenance tasks. Studying clones that were actually removed by developers may give indicators for a suitable relevance ranking.

For this reason, Göde [11] investigated deliberate clone removals in the evolution of four open-source systems to campaign for more observance of a maintainer's view. Göde found a large discrepancy between clones detected by a state-of-the-art clone detector and code clones removed by developers as the scopes of the clones hardly ever matched. His results suggest that value can be added to clone management tools by considering knowledge about duplicated code that has been selected for refactoring by a programmer.

In this paper, we extend the previous study as it is based on a relatively small number of subject systems and less than one year of the system's evolution has been analyzed. To validate the results in a more extensive analysis, we analyze the evolution of eleven open-source systems over a period of two years and investigate further characteristics of clone removals or refactorings, respectively. Based on the findings of this study, we will answer the following research questions:

**Question 1** — *How often and by what kind of refactorings is duplicated code deliberately removed?*

Just as Göde [11] did, we investigated whether and with what frequency code clones have been deliberately removed by developers. An answer gives a clue about the programmer's view on managing duplications. In addition, the refactorings applied to remove duplicated code are compared to clones detected by a state-of-the-art detection tool. As a result of the comparison existing weaknesses in automated detection can be uncovered and eliminated.

**Question 2** — *How often and by what kind of refactorings is duplicated code accidentally removed?*

Apart from deliberate removals that have been subject of the previous study by Göde, clones might be removed accidentally as a side effect by some other refactorings. We analyzed how often duplicated source code is removed by arbitrary code modifications by chance. Frequent accidental clone removals could be a reason for developers not to manage clones actively as the problem will resolve itself in the long-term. Moreover, information on refactorings that were actually not meant to remove cloned code but even so did, might contribute to the improvement of clone management tools by uncovering characteristics that indicate good clone candidates for a removal—focusing exclusively on deliberately removed clones might not reveal these characteristics.

**Question 3** — *Are there clones that are missed by a refactoring?*

Besides analyzing clones that have been removed by refactorings, we also investigate whether or not duplicated code that could have been removed in consequence of a performed refactoring were missed by developers—which was no subject of Göde’s study. Developers may miss clones when performing a suitable refactoring if they are not aware of them. This risk could be mitigated using automated tool support that provides useful information.

**Question 4** — *What kind of measurable code characteristics may help in ranking clone candidates for removal?*

Having detected deliberate and accidental clone removals in the evolution of software systems, it would be useful to extract code characteristics that flag clones to be removed. The approach presented by Göde [11] to detect clone removals, which has been adapted in this paper, is only semi-automated. Being able to use code characteristics to further automate the detection process would contribute to the improvement of clone management tools as the information helps to provide only meaningful data to the user. Göde investigated a few preliminary metrics, but his results suggest that those metrics do not clearly indicate good candidates. We will extend his preliminary study by analyzing additional metrics.

**Contribution.** To provide further insights and answer our research questions, we replicate and extend the study by Göde [11] who considered only a small number of subject systems and investigated exclusively those clones that were deliberately removed. To overcome these shortcomings in our study, we analyze the evolution of eleven subject systems over a two year time period. Moreover, we extend the study by also considering accidental clone removals and collect additional metrics that might indicate good candidates for clone removal through refactoring.

**Outline.** The remainder of this paper is organized as follows. Section II presents related work, including Göde’s study on deliberate clone removals. Our approach of analyzing clone removals is described in Section III. Section IV presents our case study and the results. Section V concludes.

## II. RELATED WORK

Extracting and analyzing the evolution of clones have been subject to recent studies in clone research. This section summarizes previous work that is related to ours.

### A. Clone Evolution

Antoniol and colleagues conducted a study on the extent and the evolution of clones in 19 releases of the Linux kernel [1]. They found that the amount of clones was rather small and that the clone ratio tends to remain stable across versions.

Kim and colleagues presented the first clone evolution model based on the mapping of code clones across multiple versions [21]. They investigated the evolution of clones in

two Java systems by mapping clone classes<sup>1</sup> of consecutive versions. Based on the results they concluded that the detected clones were either very volatile or hard to remove.

The evolution of the Linux kernel has also been studied by Livieri and colleagues who used metrics on the clone ratio [26]. Examining 136 versions of the kernel they found that the amount of clones was proportional to the system size, whereas most clones were caused by one particular subsystem.

Aversano and colleagues expanded the work of Kim and colleagues by investigating the same software systems, but adding further patterns regarding the evolution of clones [2]. They investigated how fragments of the same clone class were maintained by differentiating between inconsistently changed fragments that are continuously maintained independently and those that are made consistent in a later version (late propagation). It was found that most clone classes are maintained consistently, whereas the majority of inconsistent changes to clone classes were intended.

Göde presented a model for describing the evolution of cloned fragments and integrated it into the incremental clone detector *iClones*<sup>6</sup> [10]. Empirical data have been obtained by analyzing the evolution of identical clones in nine open-source systems. It was found that the clone ratio decreased in the majority of the systems and that cloned fragments existed more than a year on average. Moreover, he found that either consistent or inconsistent changes to clone classes were more frequent depending on the system.

Saha and colleagues expanded the work of Kim and colleagues [21] by studying different aspects of clone genealogies at release level [31]. They analyzed 17 open source systems covering four different programming languages. Their results show that the majority of detected clones were either not changed or changed consistently and that many genealogies remain alive during the evolution.

In another study Saha and colleagues [32] evaluated clone genealogies of three open-source projects. They manually analyzed many of the detected genealogies considering predefined change patterns and conclude that their approach is scalable while maintaining high precision and recall.

We replicated and extended the study by Göde [10] by also considering near-miss clones [6]. By analyzing seven open-source systems we found that the clone detector reported remarkably more cloning related to near-miss clones compared to identical clones.

### B. Changing Clones

Jarzabek and colleagues performed a case study on cloning in the Java Buffer library [18]. They propose a technique to unify clones in situations in which it is difficult to eliminate them with conventional program design techniques. The approach has been evaluated in qualitative and quantitative ways as well as in an controlled experiment. It was found that unifying clones reduced conceptual complexity and enhanced the changeability at rates proportional to code size reduction.

<sup>1</sup>A clone class contains all clone fragments that are sufficiently similar to each other.

Bakota and colleagues investigated change patterns in clone evolution using an AST based machine learning approach [3]. They used different similarity metrics to map individual cloned fragments of consecutive versions. Different patterns have been found that were related to bugs.

Krinke performed two separate studies to investigate the stability of exact clones [23], [24]. In the first study he analyzed five open-source systems looking at weekly snapshots over a 200-week period of time. He found that clones are changed inconsistently about half of the time, but late propagations are rare. In the second study he focused on whether cloned code is more stable than non-cloned code. Göde and Harder partially replicated and extended this study [12]. In general both found that cloned code is more stable than non-cloned code except for code deletions. In addition, Göde and Harder observed that varying parameters of the clone detector do influence the results, still conserving the relation.

Bettenburg and colleagues investigated changes to clones at release level focusing on inconsistencies [8]. In a case study on two open-source systems they observed that the risk of unintended inconsistencies is low and that only a very small number of inconsistent clones introduced defects.

Thummalapenta and colleagues conducted a study on four open-source systems to analyze specific patterns in the evolution of code clones and focused on examining characteristics of the late propagation pattern [33]. It was found that occurrences of the late propagation pattern were often related to defect-correcting changes. Barbour and colleagues expanded this study by using clone genealogies from two open-source systems to examine more characteristics of late propagations [5]. They found that the late propagation pattern indicates a risky cloning behavior regarding defects.

Another case study towards the correlation of late propagations and defects was conducted by Hui Mui [17]. Analyzing four Java systems based on log information provided by software repositories it was found that late propagations are not very common and about one quarter of the detected ones cause a bug—concluding that the overall impact is moderate.

Göde presented the first study of code clone removals [11]. He investigated different aspects of deliberate clone removals to get a clue of the developers’ view on clones. Analyzing four open-source systems he found a number of intentional clone removals, but the gap to clones detected by a clone detector was remarkable though. Moreover, it was found that the scope of the refactorings hardly ever matched the scope of detected clones indicating that the programmers lacked awareness of the extent of clones in the projects or that clone tools are not accurate enough. Regarding metrics that might be useful to detect duplications that are good candidates for refactorings the findings did not provide clear results. Finally, Göde analyzed the committers of clone removals and found that the less people were involved in the projects, the more intentional removal of code clones took place.

Göde and colleagues also studied the frequency and risks of changes to clones in the history of three subject systems [14]. They found that 12.2% of the clones were changed more than

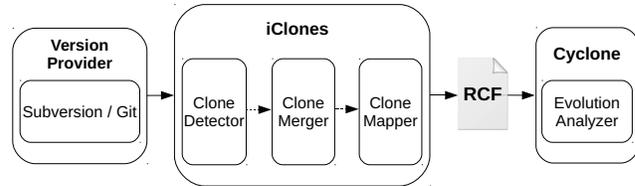


Fig. 1. *iClones* detects all clones in the evolution of a system’s source code provided by the *Version Provider* and writes the clone data in an *RCF*<sup>6</sup> file. Afterwards, *Cyclone* reads the *RCF* to track and visualize the clone data based on an evolution graph.

once and that nearly 14.8% of all changes were accidentally inconsistent. Based on the results they conclude that the history of clones provides important insights to determine their relevance regarding maintenance tasks. Göde and Harder confirmed their findings in a follow up study [13].

Volanschi presented a refactoring technique for code clones [35]. The method is supposed to guarantee strong safety while leaving the spectrum of refactoring techniques open, for instance, to manual interventions. Volanschi evaluated the approach prototypically on a subset of a real-world legacy asset concluding that the results are promising.

### III. ANALYZING CLONE REMOVALS

This section describes the semi-automated detection and the analysis of deliberate and accidental removal of code duplication in the evolution of a software system. The approach proposed in this paper can be separated into different steps that are repeated for each version<sup>2</sup> of the subject systems. Each step will be described in the following. An overview of the framework is given in Figure 1.

#### A. Repository Mining

The *Version Provider* extracts necessary information from the particular software repository—currently Subversion<sup>3</sup> and Git<sup>4</sup> are supported. Only source files and corresponding log information are considered that match the programming language of the subject system under study (e.g., property and documentation files are ignored). Due to the incremental approach used in our clone detector, it is not necessary to analyze each version from scratch. Instead, source code changes of consecutive versions are determined and processed. After analyzing and extracting relevant files the *Version Provider* passes the fetched information to *iClones*.

#### B. Clone Detection

A clone detector is used to analyze all versions of the software system under study. The proposed framework in this paper uses an enhanced version of the clone detector used by Göde [11], namely *iClones*. *iClones* detects clones in two separate steps and, afterwards, maps clones between consecutive versions to build the evolution model for each

<sup>2</sup>A version is a snapshot of a program’s source code at a given time. Accordingly we denote a version at a particular point of time  $t$  as  $v_t$ .

<sup>3</sup><http://www.subversion.apache.org/>

<sup>4</sup><http://www.git-scm.com/>

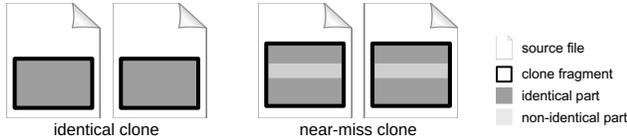


Fig. 2. Fragments of identical as well as near-miss clones need to have at least 50 tokens to be reported by *iClones*. In addition, each identical part of a near-miss clone fragment needs to have at least 10 tokens and the non-identical part needs to have at least the same size as the shortest neighboring identical part.

system [6], [10]. First, all type-1 clones (code fragments without any differences) of the current version are detected by the *Clone Detector*. In the second step these clones are merged to larger near-miss<sup>5</sup> clones by the *Clone Merger*, if possible. The algorithm is able to merge multiple adjacent code fragments to larger near-miss clones. To be merged, each code fragment has to be at least 10 tokens in size and the number of different tokens between two fragments needs to be smaller in size than the shorter of the two fragments. In addition, *iClones* has been configured to report only identical and near-miss clones with a minimum total length of 50 tokens. The values of 10 and 50 tokens for the minimum length thresholds is based on our experience from former studies using *iClones* [6], [10], [11], [15]. Figure 2 illustrates these bounds for identical and near-miss clones. Having detected all clone fragments of a version, every fragment is grouped with its clones into a clone class.

The detection process is repeated for each version of a program. During this process the *Clone Mapper* component uses a diff-based approach to map each fragment of version  $v_{n+1}$  to its ancestor fragment in version  $v_n$  by determining applied source code changes between the versions. By tracking each fragment throughout the history of the subject systems our clone evolution model is built, which enables analyzing clone fragments and their attributes independently over time. Due to its technical nature the mapping process is not described in detail in this work—we refer to the original publication for details on the mapping [6].

After analyzing every version of a system *iClones* writes the clone data into an *RCF* file that is used by our clone inspection tool *Cyclone*<sup>6</sup> to detect clone removals in the next step.

### C. Clone Removal Identification

Detecting clone removals is quite difficult and time consuming as there is no method to do the task completely automated. However, Göde [11] presented an approach that is intended to help detecting duplicated code that has been removed between two consecutive version  $v_n$  and  $v_{n+1}$  of a software system. We adapted this method to detect deliberate and accidental clone removals and integrated the approach into our clone inspection tool *Cyclone*. The identification of removals consists of two

<sup>5</sup>We summarize type-2 (differences regarding identifiers and literals) and type-3 (differences apart from identifier and literals) clones as near-miss clones.

<sup>6</sup><http://www.softwareclones.org>

steps. The first is to filter all clones that cannot be part of any removal and the second is to manually decide which of the left-over clones are in fact affected by removals of duplication.

**Filtering:** The data generated by *iClones* contains all relevant data including the required information to build the evolution model. Therefore, *Cyclone* is solely used to inspect the huge amount of data computed by *iClones* and has no impact on the detection of cloned code, and hence, on precision and recall of the results. *Cyclone* is used to visualize the evolution model and helps the user to inspect different aspects of cloned code over time. To identify the removal of cloned code, the first objective is to determine the set of clone fragments that have been either modified or deleted. Cloned fragments that have not been changed can be ignored. In addition, fragments are filtered that have only been marginally changed between two versions of a system and, therefore, can most probably be neglected from further considerations. *Cyclone* offers a threshold that specifies the minimum bound of modified and deleted tokens in a cloned code fragment. In this study we used the same bound of 15 tokens as Göde did. His study showed that this is an appropriate bound to filter many uninteresting changes with only a small risk to miss true removals [11]. Let  $F$  be the resulting set of clone fragments that have been changed sufficiently. It is notable that it is not sufficient to consider only clone fragments that have been completely removed from  $v_n$  to  $v_{n+1}$  as the scope of a refactoring hardly ever matches the scope of the detected clone fragments precisely.

The set  $F$  for each system is automatically detected by *Cyclone* based on source-code changes between consecutive versions determined using a standard diff algorithm<sup>7</sup>. Every clone fragment that is part of  $F$  is marked in the evolution model and the user is able to navigate through the set.

**Decision Process:** To decide from which fragments of the set  $F$  code duplications have been removed either deliberately or accidentally and whether possible refactorings have been missed by the developers must be done manually. For each fragment of  $F$  we checked the commit messages and reviewed the source code before and after the corresponding changes to judge whether code duplication has been removed or not. To assist the manual examination of commit messages *Cyclone* marks versions in the evolution graph whose commit messages include indicating keywords, e.g., removal, refactoring or duplication. Depending on the commit messages we sometimes had clear indications that code duplication was removed and where these changes took place exactly. However, from our experience commit messages are often imprecise or inaccurate so we used them only to get a first impression of what was done and not as decisive criterion. This means that we always analyzed the source code changes even if the commit message already gave a clear hint or gave no hint at all.

Afterwards, we analyzed the source code changes between two versions to get more reliable information. The code review was done on different levels. First, we used the integrated

<sup>7</sup><http://code.google.com/p/google-diff-match-patch/>

source-code view of *Cyclone*, which focuses on the cloned fragments. It gives us the files, the exact positions of the cloned fragments and for near-miss clones the identical and non-identical parts at a glance. Especially, the non-identical parts of near-miss clones have to be kept in mind further on to separate changes to them from changes to the actual cloned parts. Due to the fact that our evolution model is based on clone classes, detected clone fragments might appear in more than one clone class of the same version, for instance, if fragments of different clone classes partially overlap the same code segments. Therefore, we used information about the fragments' file paths and source locations for grouping overlapping fragments of different clone classes for save us from analyzing changes to the same code multiple times.

For a detailed code review we used the visual diff tool *Meld*<sup>8</sup> and *Eclipse*<sup>9</sup>. Based on the information collected using *Cyclone* we used *Meld* to extract and review all modifications of files affected by changes that might have led to a removal of code duplication. In some cases we were not able to decide whether or not cloned code has been removed based on the change information provided by *Meld*. In such cases it was necessary to review more general project attributes in addition to the *Meld* diff, for instance, library and API updates. We used *Eclipse* to review the corresponding project attributes before and after the changes happened.

If we were able to detect the elimination of code clones by a refactoring, we exported the information of the corresponding clone fragments using *Cyclone*. The export is done automatically and collects different kinds of data. Among others we have exported basic information such as the version in which the refactoring took place, the file paths and source locations of the affected fragments and the date of change. In addition, we also exported different metrics, for instance, *LOC*, the number of consistent and inconsistent changes to the clone fragments before the refactoring and the *Cyclomatic Complexity* of the fragments. These metrics are used in our case study to investigate whether or not a ranking of clones can be based on these metrics. Such a ranking is needed to reduce the amount of clone data delivered to the users of clone management tools. If, for example, the number of changes to a certain clone fragment was high before it was removed, it might be an indication that the developers wanted to avoid the extra effort of continuously changing the cloned code and decided to refactor the corresponding source code.

**Categorization and grouping:** Based on this decision process we manually identified refactorings that removed code duplication and categorized the observed code removals. A schematic illustration of the resulting categories is shown in Figure 3. We started our categorization based on the initial set  $F$ . In the first step we split  $F$  into the sets  $Miss$  and  $F'$ .  $Miss$  includes all clone fragments related to refactorings that removed code duplication but missed some clones that could have been removed by the refactoring, too. Accordingly, the

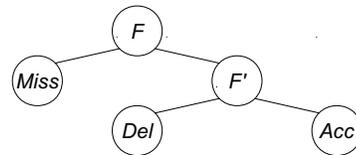


Fig. 3. Schematic illustration of our categorization of clones.

```

1 ...
2 int options = Regexp.MATCH_DEFAULT;
3 if (!casesensitive) {
4   options |= Regexp.MATCH_CASE_INSENSITIVE;
5 }
6 ...
  
```

(a) Prior to change

```

1 ...
2 int options = RegexpUtil.asOptions(casesensitive);
3 ...
  
```

(b) After change (2010-08-17)

Fig. 4. Deliberate removal of duplicated code in *Ant*. The initialization of the *options* variable has been replaced in four different classes by a method call to a new method implemented in a general utility class.

set  $F'$  includes clone fragments related to refactorings that removed all suitable clones. We further split the set  $F'$  into the two sub-sets  $Del$  and  $Acc$ .  $Del$  includes clone fragments that were deliberately removed using appropriate refactorings. Figure 4 provides an example taken from *Ant* to illustrate our decisions on deliberate clone removals. Besides refactorings that were mainly performed to remove code clones, we also identified refactorings that removed code duplication accidentally as a side effect. This means that we found clear indications that the goal of the corresponding changes was actually not to remove clones. Clone fragments related to such refactorings are assigned to the set  $Acc$ . We encountered different kinds of refactorings removing duplication as side effect, for instance, the removal of deprecated or dead code and updates of libraries and APIs. Note that clone fragments related to a refactoring that missed suitable clones are assigned to set  $Miss$  independent from the fact whether the refactoring was meant to remove duplication or not. The number of missed clones in this set is a helpful indicator when it comes to the question how much developers can benefit from using clone management tools that could pinpoint to similar code during code editing.

Finally, we grouped clone fragments within the sets  $Del$ ,  $Acc$  and  $Miss$  further. Clone fragments within the same set that are part of the same refactoring activity were counted as a ROD (Removal of Duplication) unit. The term ROD is adapted from Göde [11] and denotes a set of changes to remove duplication in which all clone fragments affected by one or more refactorings originated from a common intention, for instance, extracting equal functionality of a class into an method. RODs are manually identified and may contain fragments affected by refactorings that have been performed in different versions—corresponding to different commits—of a system as long as all of them have been carried out for the same reason. As an example, if three clone fragments

<sup>8</sup><http://meldmerge.org/>  
<sup>9</sup><http://www.eclipse.org/>

were affected by the removal of code duplication during the introduction of a new utility class that includes a method providing the same functionality as the three deleted fragments and the fragments have been replaced by a method call to the new class, we count the removed fragments as one ROD. It does not matter whether all fragments have been replaced by the method call in the same or in different commits. Note that RODs can even have only one clone fragment assigned. For instance, if two statement sequences  $A$  and  $B$  are clones and  $B$  is the body of a function  $f$ . If  $A$  is replaced by a call to  $f$ ,  $A$  would be the sole fragment in the corresponding ROD.

#### IV. CASE STUDY

In this section, we present the results of our evaluation. We analyzed eleven realistic open-source systems to gather empirical data that supplement previous findings and answer our research questions.

##### A. Study Setup

We selected the open-source systems based on the following criteria:

- The systems are open-source and maintained using a publicly available *Subversion* or *Git* repository.
- Each system has a reasonable size to provide a sufficient amount of data to obtain meaningful results, but is still manageable for manual inspection.
- We included systems that have already been analyzed by Göde [11] to allow for comparisons to his results.

The systems selected are *Ant*, *FileZilla*, *FindBugs*, *FreeCol*, Apache’s *httpd*, *JabRef*, *Nautilus*, *Umbrello*, *ADempiere-Client*, *ArgoUML* and *TortoiseSVN*. The last three were analyzed in Göde’s study [11]. Göde also investigated *KDE-Utils*, which we did not because *KDE-Utils* is a collection of different tools. Removing code duplication within one software system compared to the removal of clones between different systems is a different use case and brings in other aspects that need to be considered. The tools of *KDE-Utils* are related but though have their own source code and, therefore, present a combination of intra-system and inter-system clone detection use case. We investigated two years of each system’s history, starting from January 2009 to December 2010. We chose a time period that subsumes the time period investigated by Göde [11], which was from January 7 to October 29, 2009. We decided to increase the time period under study, because Göde reported that he had to dismiss different software systems from investigation as he was not able to find indications of clone removals. To lessen the risk of having the same problem and to check the impact the time period under study has on the results, we analyzed a longer period of the evolution of the subject systems. Snapshots have been analyzed on an interval of one day. Dates that did not contain any commits including changes to the source code were skipped. Details of the subject systems are given in Table I.

We have configured *iClones* to analyze the subject systems and report clone fragments with a overall minimum length of 50 tokens and a minimum length of 10 tokens for identical

TABLE I  
SUBJECT SYSTEMS

System	Language	KSLOC	Clone Classes	Clone Fragments
<i>ADempiere-Client</i>	Java	68 – 63	1,908,109	5,010,007
<i>Ant</i>	Java	175 – 160	696,446	1,813,720
<i>ArgoUML</i>	C++	181 – 259	873,485	2,249,690
<i>FileZilla</i>	C++	103 – 113	513,912	1,150,498
<i>FindBugs</i>	Java	160 – 216	260,678	628,940
<i>FreeCol</i>	Java	94 – 108	614,193	1,605,787
<i>httpd</i>	C	187 – 200	747,684	1,734,741
<i>JabRef</i>	Java	80 – 103	347,460	888,642
<i>Nautilus</i>	C	136 – 115	330,957	807,891
<i>TortoiseSVN</i>	C++	150 – 260	1054,250	2,438,919
<i>Umbrello</i>	C++	200 – 202	734,862	1,774,858

parts regarding near-miss clones as described in Section III-B. The reported output of each analysis was analyzed using *Cyclone*. In total we inspected 5152 clone fragments in 2967 clone classes that have been assigned to the set  $F$  of cloned code fragments, using our approach and counting overlapping fragments that appeared in different clone classes of the same version just once. From these, we found 1293 fragments to be directly related to either deliberate, accidental, or missed removals of duplication. Unifying fragments that were related to the same refactoring task we identified 224 RODs.

To ensure the accuracy of the manual inspection and lessen the impact of subjectivity of the oracle, the two authors performed the manual inspection separately from each other. The first author did the manual analysis as described in Section III, before the second author checked 20% of the results for each subject system under study in a sampling process using the same approach as the first one did—not knowing the decision of the first author. The result of the sampled cross-check uncovered only few differences regarding the assessment of the refactorings performed, for instance, the scope of refactorings compared to the scope of detected clone fragments. The disagreements were all minor and could be resolved in short discussions among both authors. Further, we validated our results as far as possible using the study from Göde [11]. For those systems that have been covered and for those aspects that have been investigated by both studies we also found a general agreement regarding our results.

##### B. Removal of Duplication

Table II shows the results of our manual inspection and partially answers our research questions 1–3. The second and third columns show the number of clone classes and clone fragments detected by *iClones* that are related to RODs. The number of resulting RODs is given in column four. It can be seen that the numbers vary for each system with *TortoiseSVN* contributing the largest and *Umbrello* the smallest numbers of RODs. The last three columns present the absolute numbers of detected RODs within the corresponding set of clone fragments. The overall numbers of RODs generally comply with the numbers of affected clone classes and fragments. Again, the most activity regarding the removal of code clones were found in *TortoiseSVN*. Except for *TortoiseSVN*, *ArgoUML* and *FreeCol* the number of removals in the other systems is rather

TABLE II  
REMOVAL OF DUPLICATION

System	Classes	Fragments	ROD	Del	Acc	Miss
<i>ADempiere-Client</i>	30	57	13	5	5	3
<i>Ant</i>	26	42	6	1	3	2
<i>ArgoUML</i>	165	368	36	14	16	6
<i>FileZilla</i>	32	55	11	7	3	1
<i>FindBugs</i>	17	30	8	4	3	1
<i>FreeCol</i>	64	133	31	10	18	3
<i>htpd</i>	17	37	7	3	1	3
<i>JabRef</i>	2	5	2	1	0	1
<i>Nautilus</i>	38	75	23	0	22	1
<i>TortoiseSVN</i>	270	483	88	41	29	18
<i>Umbrello</i>	4	8	1	0	0	1

small and for three systems we did not discover any removal in some change categories. Nonetheless, comparing deliberate and accidental removals, it can be observed that they are relatively balanced. The only striking gap is for *Nautilus* for which no deliberate, but 22 accidental RODs were found. This suggests that for the systems under study active clone management has been performed by the developers, but clones have also been managed unknowingly as a side effect of other refactoring activities. Regarding code clones that have been missed by suitable refactorings, we were able to detect only a few, except for *TortoiseSVN*. The results show that there may be an opportunity for clone detectors to reduce the likelihood of missed refactorings.

Overall, we can confirm the results of Göde’s study on deliberate clone removals for the software systems he studied [11], too, for the shorter time frame. Göde reported for *ArgoUML* one and for *TortoiseSVN* four more deliberate removals for the time period shared by both studies. The reason for the differences is that we classified these RODs as accidental removals. We assume that our more differentiated categorization compared to Göde’s contributed to this divergent judgment. Göde did not further classify RODs. Our differentiation may have led us to judge more strictly on deliberate removals.

### C. Refactorings

To investigate how clones are removed by developers and complete the answers to our research questions 1–3, we have investigated the refactorings related to the detected RODs presented in Section IV-B. A better understanding of which clones attract the attention of developers might help to improve existing detection and management tools to produce more useful results by ranking the results. Our case study setup suits the use case of removing code duplication and therefore we inspect what kind of refactorings developers used to remove clones and how well the scope of the refactorings fits the scope of the clone fragments reported by our clone detector. Göde inspected refactorings and their scope in his previous study [11]. He stated that a good and comprehensive matching is the basis for the use of clone management tools. On the contrary, a bad matching indicates either automated clone detection is not accurate enough or developers are not aware of the duplication’s extent. He found that there is a remarkable discrepancy between clones detected by a state-of-the-art clone

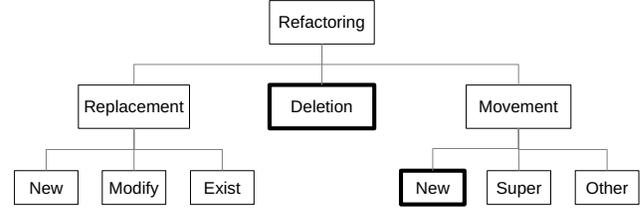


Fig. 5. Classification of refactorings based on manually detected clone removals.

detection tool and duplications removed by developers—the scope of the refactorings hardly matched the scope of the detected clone fragments.

We adapted the approach to categorize RODs from Göde [11] to our results. The taxonomy we used to classify refactorings is an extended version of his taxonomy and emerged as a result of our manual inspection process to detect refactorings that removed code clones. Our classification is based on refactorings commonly quoted as suitable to remove duplications, for instance, *Extract Method* and *Pull Up Method*. Figure 5 shows the resulting classification. The bold rectangles depict which parts have been added to the classification compared to the one presented by Göde.

We detected three categories of refactorings that were applied to remove clones. In contrast to Göde we do not separate unifications into a category on its own, because gathering code that provides equal functionality in a single place is based on the movement of the corresponding code and, therefore, we count unifications in our *Movement* category.

**Replacement** We distinguish two cases in this category. The first one is that a sequence of statements has been replaced by a single method call either to a newly introduced method (*New*) or to a method that already existed before the refactoring took place (*Exist*). The second one includes the replacement of a sequence of statements within methods by another sequence of statements (*Modify*). In most cases we detected that the nesting within methods has been reduced due to high complexity—which also led to the removal of clones.

**Movement** This category includes refactorings—such as pull-up field or method—that gather two or more field declarations or methods providing equal functionality in a single class. We differentiate between three types of movements. First, the code is moved to a new class that did not exist before (*New*). Second, the code of the affected classes is moved to a superclass (*Super*) of them. Last, the code is moved to an existing class that provides functionality used by a range of classes, e.g., a utility class, but that class is not a superclass of the classes from which the code is moved (*Other*).

**Deletion** Field declarations or methods are removed. Mainly this category includes deletions of dead or deprecated code. We categorized deletions of deprecated code as deliberate clone removals if it was still in use before the corresponding refactoring deleted it and the refactoring’s

TABLE III  
SCOPE OF REFACTORINGS COMPARED TO CLONE FRAGMENTS

Refactoring	RODs	Match	Containing	Contained
<b>Replacement</b>	<b>39</b>	<b>10</b>	<b>17</b>	<b>12</b>
New	31	9	14	8
Modify	-	-	-	-
Exist	8	1	3	4
<b>Movement</b>	<b>45</b>	<b>5</b>	<b>9</b>	<b>30</b>
New	12	1	1	10
Super	33	5	8	20
Other	-	-	-	-
<b>Deletion</b>	<b>2</b>	<b>2</b>	-	-
<b>Total</b>	86	17	26	43

purpose was to eliminate duplication.

In addition to the taxonomy, we adapted the approach of comparing the scope of refactorings to the scope of clone fragments reported by our clone detector from Göde [11]. For each ROD the scope of its clone fragments is manually compared to the related refactoring performed. An advantage of the manual comparison is that we can tolerate various artifacts of token-based clone detection at the beginning and end of fragments, for instance brackets. The comparison results in each fragment being classified to either:

- *Match*: if all statements of the clone fragment are part of the refactoring and vice versa
- *Containing*<sup>10</sup>: if the changes refer only to statements within the bounds of a clone fragment
- *Contained*<sup>10</sup> if the scope of the refactoring (the statements changed by the refactoring) subsumes the clones either completely or partially, that is, just overlaps with the clone fragment

We found that for all clone fragments in the same ROD the comparison of scopes led to the same classification because of which we report the resulting numbers based on RODs rather than on clone fragments. Table III presents the results of deliberate RODs classified as either *Match*, *Containing* and *Contained*. Although, we have analyzed accidental and missed RODs and their corresponding scope as well, we leave out the exact numbers, because these RODs were not meant to remove code duplication in the first place. It is arguable whether the scope of such refactorings needs to match the scope of the clone fragments at all. However, analyzing the results for accidental and missed RODs, we generally found the same trend as presented for deliberate RODs. Note that the two categories *Modify Existing Method* and *Move to Existing Class* of our classification were added because of refactorings related to accidental or missed removals of code clones. There were no instances of deliberate removals in these categories.

Table III shows that our results confirm the large discrepancy between the scope of detected refactorings and the scope of detected clone fragments stated by Göde [11]. Overall only about 20 % of all refactorings have been categorized to

<sup>10</sup>Göde used the term Superior instead of Containing and Inferior instead of Contained—we considered the terms Superior and Inferior mistakable

match the scope of the detected clone fragments. Göde found 16 % of his analyzed refactorings to match the scope of clone fragments. Focusing on refactorings that did not match the scope of detected fragments we found that for the category *Movement* clearly more refactorings go beyond the scope of clone fragments. In contrast to that, there is no trend for refactorings that replaced source code regarding how well their scope fits the scope of detected clones. Looking at refactorings that replace or move code, respectively, their occurrence is almost balanced. In comparison, deliberate clone removals by refactorings that fall into the category *Deletion* are quite rare.

Finally, we analyzed how many of the clone classes that included clone fragments affected by deliberate or accidental removals of duplication disappeared completely in the version after the corresponding refactoring was applied. We found that about 90 % of the clone classes affected by deliberate clone removals and about 80 % of the clone classes affected by accidental removals disappeared in the following version. This could be assumed to prove a high success rate in reducing clones regarding the refactorings chosen by developers. However, it should be kept in mind that the amount of clone classes and fragments related to the reduction of duplications is quite small compared to the overall numbers of clone classes and fragments detected by our clone detector.

#### D. Ranking Clones

To investigate whether and what measurable characteristics may help in ranking clone candidates for removal and answer our research question 4, we collected different clone metrics based on our retrospective analysis. Göde investigated different aspects of clone removals to contribute to the ranking of clones regarding the use case under study [11]. He limited his study to the following attributes: length, similarity, distance in the source tree, and number of source code files that contain the fragments of a ROD. Overall his results did not yield clear results, only for the distance attribute there was a trend that developers mostly removed duplicated code located in the same source code file. Göde assumed that other attributes may help in ranking clones for removal. To contribute and extend Göde’s preliminary analysis we collected and evaluated different clone characteristics based on the clone classes and clone fragments related to our detected RODs.

The first aspect we investigated is whether developers tend to perform more refactorings on code clones that are identical or just similar. We analyzed how often RODs were related to identical and how often to near-miss clones based on detected clone classes rather than on clone fragments. This is sufficient as all clone fragments of the same clone class have the same clone type, because clone fragments are grouped in clone classes based on their level of similarity. Table IV shows the result considering type-1, type-2, and type-3 clones. The first column depicts the three sets of refactorings we have investigated, the second column the number of clone classes affected by removals of duplication and the last three columns depict how many of these clone classes have been assigned to either type-1, type-2, or type-3 clones.

TABLE IV  
TYPE OF REFACTORED CLONE CLASSES

Refactoring	Classes	Type-1 [%]	Type-2 [%]	Type-3 [%]
<i>Del</i>	331	37.0	28.0	35.0
<i>Acc</i>	247	32.3	29.7	38.0
<i>Miss</i>	87	29.0	32.3	38.7
<b>Total</b>	665	32.8	30.0	37.2

It was found that there is no real tendency towards one of the analyzed clone types. Each type roughly constitutes one third of all detected RODs independent from the classification of the refactorings. However, the numbers indicate that type-1 clones are more often deliberately removed and that type-2 and type-3 clones are slightly more often affected by accidental removals as a side effect. The results are a bit surprising as we expected a more distinct trend towards deliberate removal of identical clones, because a semantic preserving refactoring of near-miss clones naturally requires more effort and often there is no automated tool support that can be used to perform the refactorings. On the other hand, we assume differences of type-2 clones to be rather moderate what makes them close to identical and, hence, easier to refactor.

Apart from the clone type, we collected data regarding the length of fragments and how many fragments affected by RODs decreased in size. Regarding changes to clone fragments we have analyzed how often detected clone fragments have been changed consistently or inconsistently, respectively, over time until they have been refactored by developers. The change frequency of clone fragments may indicate which clones might be good candidates for a removal. Assuming that a high change frequency of cloned code causes additional effort to keep the clones synchronized, a high change frequency is costly for developers and a refactoring probably pays off in the long term as changes have to be done at only one single place afterwards. The last characteristic of clone fragments we looked into is the *Cyclomatic Complexity*. A high *Cyclomatic Complexity* is an indicator for source code that has a more involved control flow. As a consequence changes need more effort to be performed and tested. Based on this assumption the *Cyclomatic Complexity* may be useful to identify clone fragments that are good candidates for refactorings, too.

Table V shows the results for the collected metrics. The first column specifies whether a deliberate, accidental, or missed clone removal has been performed. The second column gives the average size of the affected clone fragments in number of tokens. By how many tokens the clone fragments have been decreased on average is given in the third column. The next two columns depict the maximum numbers of consistent (FCC) and inconsistent (FIC) changes to clone fragments related to RODs before they have been refactored. The last column gives the average *Cyclomatic Complexity* (CC) of clone fragments related to RODs.

The results show that the clone fragments related to RODs are clearly reduced in size by the performed refactoring. For deliberate and accidental removal of duplication we have

TABLE V  
TYPE OF REFACTORED CLONE CLASSES

Refactoring	Token [Ø]	-Tokens [Ø]	FCC	FIC	CC [Ø]
<i>Del</i>	112.1	65.2	3	2	5.2
<i>Acc</i>	114.2	72.1	2	2	4.7
<i>Miss</i>	74.4	46.0	1	0	2.7

measured an average decrease around 70% of the fragment's size. Regarding partially missed fragments we have an average of nearly 50% decrease. Analyzing how frequent the affected clone fragments have been changed before the removals took place, we detected very few changes at all. Looking at the maximum number of consistent and inconsistent changes to clone fragments, we see that they range from 0 to 3. On average less than 1% of clone fragments related to RODs changed before the detected refactorings—no matter whether consistent or inconsistent. These numbers are rather small and, therefore, we assume that the change frequency of clone fragments cannot be used for an automated ranking of clones. Analyzing the *Cyclomatic Complexity* we detected a diverse distribution between fragments. The absolute numbers range from 1 to 35 without any clear tendency. On average the *Cyclomatic Complexity* ranges from 2.7 up to 5.2 per fragment as shown in Table V. Because of the large diversity among clone fragments, we could not identify any pattern to rank clones using the *Cyclomatic Complexity*.

#### E. Threats to Validity

**Tools** The accuracy of our results depends on recall and precision of *iClones*. *iClones* uses a token-based detection technique and token-based techniques are considered to provide high recall and reasonable precision [7] and that the manual inspection of the reported data reduced the threat of low precision heavily.

**Subjectivity** Our results are partially exposed to our subjective assessment of clones and refactorings. To mitigate the threat, the manual analysis was first performed by one of the authors and, afterwards, the second author independently checked a sample of 20% of the findings to verify the results.

**Subject Systems** We based our study on eleven open-source systems from different domains. Yet, this sample may still not be representative. Only a part of their history has been analyzed. Using a different period of time or analyzing snapshots at a different interval might affect the findings.

## V. CONCLUSION

We adapted the semi-automated approach from Göde [11] to detect removals of duplication in the history of a software system and integrated it into the clone inspection tool *Cyclone*. Based on this approach we tracked individual clones of eleven open-source systems over a period of two years investigating cloned code to answer our four research questions.

To answer our first research question, we analyzed deliberate clone removals and were able to find occurrences of removals in all systems except for *Nautilus* and *Umbrello*. The applied refactorings mainly replaced existing code by calls

to newly introduced methods and gathered common code of specialized classes in their superclass. Investigating accidental clone removals and answering our second research question, we found that accidental removals of duplication occur slightly more often than deliberate ones. Moreover, the refactorings used are basically the same as used for deliberate removals. That is, there is a high chance that existing refactoring support can be used for clone management, too. Based on the overall similarity in the frequency and type of the applied refactorings that removed clones deliberately as well as accidentally, we conjecture that an integration contributes to the acceptance of clone management tools as integral part of the development. The answer to our third research question whether refactorings removing clones may be incomplete further supports this point. We detected situations in which refactorings missed some clone fragments that could have been removed, too. This observation suggests that further research should investigate whether developers aided by automated clone detection during refactoring remove clones more completely.

To answer the last research question we measured different characteristics of cloned code that have been removed to observe whether they help in ranking clone candidates for removal. Interestingly, more near-miss clones were removed in the systems than identical clones. At least, for identical clones we expected to detect more deliberate removals, because removing near-miss clones normally needs more sophisticated refactorings. Analyzing the change frequency in the evolution of clone fragments revealed that the removed clone fragments rarely changed before, contrary to our expectations. Finally, we did not find any relation between control flow complexity measured as *Cyclomatic Complexity* and clone removal.

In future work we plan to conduct a survey based on a questionnaire and interviews of the developers who performed the refactorings related to the RODs of our analysis to gain further insights into a developer's view and how that knowledge can be used to rank code clones and improve existing tools.

#### ACKNOWLEDGMENTS

We thank Emil Baybulatov for his support on implementing technical features. This work was supported by a research grant of the *Deutsche Forschungsgemeinschaft* (DFG).

#### REFERENCES

- [1] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.
- [2] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. of the 11th CSMR*, pages 81–90. IEEE, 2007.
- [3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proc. of the 23rd ICSM*, pages 24–33. IEEE, 2007.
- [4] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of the 7th WCRE*, pages 98–107. IEEE, 2000.
- [5] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of the 27th ICSM*, pages 273–282. IEEE, 2011.
- [6] S. Bazrafshan. Evolution of near-miss clones. In *Proc. of the 12th SCAM*, pages 74–83. IEEE, 2012.
- [7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept. 2007.
- [8] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proc. of the 27th ICSM*, pages 85–94. IEEE, 2009.
- [9] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proc. of the 25th ICSM*, pages 169–178. IEEE, 2009.
- [10] N. Göde. Evolution of type-1 clones. In *Proc. of the 9th SCAM*, pages 77–86. IEEE, 2009.
- [11] N. Göde. Clone removal: fact or fiction? In *Proc. of the 4th IWSC*, pages 33–40. ACM, 2010.
- [12] N. Göde and J. Harder. Clone stability. In *Proc. of the 15th CSMR*, pages 65–74. IEEE, 2011.
- [13] N. Göde and J. Harder. Oops!... I changed it again. In *Proc. of the 5th IWSC*, pages 14–20. ACM, 2011.
- [14] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of the 33rd ICSE*, pages 311–320. ACM, 2011.
- [15] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.
- [16] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *PROFES*, pages 220–233, 2004.
- [17] H. Hui Mui. Studying late propagations using software repository mining. Masters thesis, Delft University of Technology, 2010.
- [18] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *J. Softw. Maint. Evol.*, 18(4):267–292, 2006.
- [19] C. Kapsner and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proc. of the 13th WCRE*, pages 19–28. IEEE, 2006.
- [20] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE*. IEEE, 2004.
- [21] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196. ACM, 2005.
- [22] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. IBFI, Schloss Dagstuhl.
- [23] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. of the 14th WCRE*, pages 170–178. IEEE, 2007.
- [24] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the 8th SCAM*, pages 57–66. IEEE, 2008.
- [25] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of the 28th ICSE*. ACM, 2006.
- [26] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the Linux kernel evolution using code clone coverage. In *Proc. of the 4th MSR*, pages 22–25. IEEE, 2007.
- [27] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proc. of the 4th MSR*, pages 18–21. IEEE, 2007.
- [28] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proc. of the 27th SAC*, 2012.
- [29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *Proc. of the 24th ASE*, pages 123–134. IEEE, 2009.
- [30] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University at Kingston, Ontario, Canada, 2007.
- [31] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Proc. of the 10th SCAM*, pages 87–96. IEEE, 2010.
- [32] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proc. of the 27th ICSM*, pages 293–302. ACM, 2011.
- [33] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *ESE*, 15(1):1–34, 2009.
- [34] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proc. of the 2004 VL/HCC*, pages 173–180. IEEE, 2004.
- [35] N. Volanschi. Safe clone-based refactoring through stereotype identification and iso-generation. In *Proc. of the 6th IWSC*, pages 50–56. IEEE, 2012.

# Content Categorization of API Discussions

Daqing Hou

Department of Electrical and Computer Engineering  
Clarkson University  
Potsdam, New York 13699  
Email: dhou@clarkson.edu

Lingfeng Mo

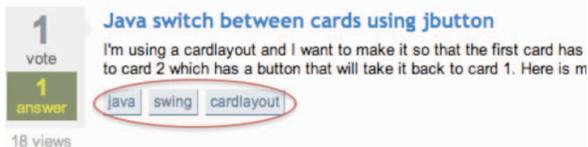
Department of Computer Science  
Clarkson University  
Potsdam, New York 13699  
Email: mol@clarkson.edu

**Abstract**—Text categorization, automatically labeling natural language text with pre-defined semantic categories, is an essential task for managing the abundant online data. An example of such data in Software Engineering is the large, ever-growing volume of forum discussions on how to use particular APIs. We have conducted a study to explore the question as to how well machine learning algorithms can be applied to categorize API discussions based on their content. Our goal is two-fold: (1) Can a relatively straightforward algorithm such as Naïve Bayes work sufficiently well for this task? (2) If yes, how can we control its performance? We have achieved the best test accuracy mean (TAM) of 94.1% with our largest training data set for the AWT/Swing API, which consists of 833 forum discussions distributed over eight categories/topics. We have also investigated factors that impact classification accuracy, with the most important two being the size of the training set and multi-label documents (the phenomenon that some discussions involve more than one category).

**Index Terms**—APIs; Online Forums; Text Categorization; Naïve Bayes; MALLET; AWT/Swing.

## I. INTRODUCTION

Online forums are commonly used by software developers to exchange information, asking questions and seeking help. Over time, these forums have become repositories of valuable programming tips and knowledge that many developers have learned to revisit routinely. To facilitate the effective browsing and searching of software forums, however, we need to find additional ways to index the forum data and better reveal their semantics, such as the Stack Overflow tags [1] shown below.



However, Stack Overflow relies on users to manually assign tags. Text categorization, automatically labeling natural language texts with pre-defined categories based on their content, can be applied to automate this task [20].

In this paper, we investigate text categorization as a fundamental approach to learn the semantic information needed for tagging forum data with pre-defined categories. This additional piece of information can then be used, for example, to annotate the master list of discussion threads in the Swing Forum that is shown below, to make it more explicit and informative as to what each entry is about.

	JTree doesn't display expanded node	lesto	2	Feb 18, 2013 5:40 AM Last Post By: lesto >
	How to load/add JPanel components to JFrame during run-time.	988385	1	Feb 17, 2013 11:12 PM Last Post By: Stanislav >
	showing image in itable..problem of update?	815769	4	Feb 15, 2013 6:50 AM Last Post By: Darryl Burke >

Our goal has been two-fold: First, we want to investigate the feasibility of using existing machine learning algorithms to categorize forum discussions. Second, in order to guide similar efforts in the future, we want to gain insights as to exactly how a learning algorithm works for this task.

We chose to start with the Naïve Bayes algorithm [13] to categorize the API discussions in the Swing Forum<sup>1</sup>. Historically, Naïve Bayes has been found performing “remarkably well” for many tasks [12], [14]. Although more recent studies have compared it with other algorithms [22], [10], due to the empirical nature of such comparisons, the results may not be universally true. Moreover, if Naïve Bayes works reasonably well for our task, other more “superior” algorithms should only perform even better, e.g., the SVM comparison in Section IV-J.

We chose the Swing Forum because it is a very active forum; as of February 12, 2013, it contains 47,258 discussion threads with 212,734 messages. We treat a whole discussion thread, which typically contains multiple messages/posts, as a training document. So it is crucial that we assign a label that accurately reflects the content of each discussion thread. An advantage of choosing the Swing Forum is that we can leverage the considerable prior research that we have conducted with the forum, e.g., [9], [19], so we can have greater confidence with the quality of our training data.

The contributions of this work are summarized as follows:

- Publicly available training data sets: To experiment with Naïve Bayes, we manually labeled three sets, a total of approximately 1,000 API discussions collected from the Swing Forum. Our largest training data set consists of 833 API discussions across eight categories specific to Java Swing. We have made our data publicly available<sup>2</sup>.
- Using our data, we show that the Naïve Bayes classifier can achieve an average test accuracy as high as 94.1%.
- We demonstrate empirically that the training sample size is the most important factor for improving classification accuracy, but this increase of accuracy plateaus once the training sample size surpasses a certain threshold.

<sup>1</sup><https://forums.oracle.com/forums/forum.jspa?forumID=950&start=0> (All URLs verified Feb. 12, 2013)

<sup>2</sup><http://www.clarkson.edu/~dhou/projects/swingForum2012.tar.gz>

- We demonstrate that multi-label documents are the major cause for classification errors.

The remainder of this paper is organized as follows. Section II describes related work. Section III covers the background for the Naïve Bayes algorithm. Section IV presents our experimentation process and main results. Finally, Section V concludes the paper.

## II. RELATED WORK

We survey some applications of machine learning to classification problems in Software Engineering. In general, different from related work, our study uses semantics-oriented classification categories. Moreover, we not only show that the algorithm works, but also investigate why and how it works.

### A. *Software Forums and Email Communication*

Gottipati, Lo, and Jiang present an approach where tags automatically inferred for posts in software forum threads are used to find relevant answers in the forums [7]. The tag inference algorithm automatically assigns seven different tags to posts: question, answer, clarifying question, clarifying answer, positive feedback, negative feedback, and junk. Experiments show that their tag inference could achieve up to 67% precision, 71% recall, and 69% F-measure. They also build a semantic search engine by leveraging the inferred tags to find relevant answers. Unlike ours, the unit of classification in their work is a message/post, rather than a whole discussion. Furthermore, their tags are about the conversational role that a message/post plays in a discussion thread, rather than the API-specific semantics of the whole discussion.

Bacchelli et al. present an approach to classify email lines in five categories (i.e., text, junk, code, patch, and stack trace) so that one can subsequently apply appropriate ad hoc analysis techniques to each category [5].

In this study, we have expanded two sets of Swing Forum discussions that we collected and analyzed as part of two of our own prior research projects [9], [19]. The first set consists of 172 discussions, which were analyzed to understand the kinds of API obstacles programmers may have in practice [9]. The second set consists of 117 discussions about the GUI layout issues, which were analyzed to drive the development of a program analysis tool supporting the use of APIs [19]. These two sets are included in our final data set of 833 discussions.

### B. *Bug Repositories*

Open source development projects typically support an open bug repository to which both developers and users can report bugs. The reports that appear in this repository must be triaged to determine if the report is one which requires attention and if it is, which developer will be assigned the responsibility of resolving the report. Large open source developments are burdened by the rate at which new bug reports appear in the bug repository. To ease the task of assigning reports to a developer, Anvik, Hiew, and Murphy [4] present a semi-automated approach where they apply a machine learning algorithm to the open bug repository to learn the kinds of

reports each developer resolves. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable to resolve the report. They have reached precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively, as well as less positive results for the gcc open source development.

A related problem in open bug repositories is duplicate bug reports that require triaging. To help with the triage process, Sun et al. [21] present a machine learning approach that can be used to classify whether a new bug report is a duplicate of any existing ones. Their techniques have been validated on three large software bug repositories from Firefox, Eclipse, and OpenOffice, showing 17–31%, 22–26%, and 35–43% relative improvement over state-of-the-art techniques.

Antoniol et al. [3] study how machine learning algorithms can be used to separate true bug reports from other change requests. They find that alternating decision trees, Naïve Bayes classifiers, and logistic regression can be used to accurately distinguish bugs from other kinds of issues. Results from empirical studies performed on issues for Mozilla, Eclipse, and JBoss indicate that issues can be classified with between 77% and 82% of correct decisions.

### C. *Change Classification*

Large software systems undergo significant evolution during their lifespan, yet often individual changes are not well documented. Hindle et al. [8] present a study to automatically classify large changes into various categories of maintenance tasks - corrective, adaptive, perfective, feature addition, and non-functional improvement - using machine learning techniques. They find that for most large commits, the Source Control System (SCS) commit messages plus the commit author identity are enough to accurately and automatically categorize the nature of the maintenance task. Using 10-fold cross validation, they achieved accuracies consistently above 50%, indicating good to fair results.

Kim et al. [11] study a new technique for finding latent software bugs called change classification. Change classification uses a machine learning classifier to determine whether a new software change is more similar to prior buggy changes, or clean changes. In this manner, change classification predicts the existence of bugs in software changes. The classifier is trained using features (in the machine learning sense) extracted from the revision history of a software project, as stored in its software configuration management repository. The trained classifier can classify changes as buggy or clean with 78% accuracy and 65% buggy change recall (on average). Murgia et al. [17] present another study with similar goals.

## III. THE NAÏVE BAYES ALGORITHM

Text categorization involves the classification of text documents into a set of categories [20]. When classifying API discussions, the text documents are the discussion threads, and the categories are the central topics that are discussed in the threads. In machine learning, documents are called instances and attributes of an instance are called features. Instances may

also have a label that indicates the category, or class, to which it belongs. A supervised machine learning algorithm takes as input a set of instances with known labels and generates a classifier, which can then be used to assign a label to an unknown instance. The process of creating a classifier from a set of instances is known as *training the classifier*.

Our work focuses on using supervised learning for classifying API discussions. We used the MALLET machine learning toolkit for language processing [15]. In what follows, we briefly review the Naïve Bayes algorithm, about which more details can be found elsewhere [13], [16], [18], [12].

Naïve Bayes learns a classifier from a training sample that is made of multiple training instances, where each instance is labeled with one of  $k$  pre-defined categories  $C_1, C_2, \dots, C_k$ . Given a textual document  $D$  with an unknown category, the learned Naïve Bayes classifier assigns  $D$  a category  $C$ , which is the category  $C_i$  that yields the highest conditional probability  $P(C_i | D)$ . Formally,  $C$  is chosen according to Equation 1:

$$C = \arg_{C_i} \max P(C_i | D) \quad (1)$$

Therefore, to choose  $C$ , it is necessary to learn to calculate  $P(C_i | D)$ . Based on the Bayes Theorem shown in Equation 2,

$$P(A | B) * P(B) = P(B | A) * P(A) \quad (2)$$

$P(C_i | D)$  can be calculated according to Equation 3:

$$P(C_i | D) = \frac{P(D | C_i) * P(C_i)}{P(D)} \quad (3)$$

Among the three terms in the right-hand side of Equation 3, the denominator  $P(D)$  is effectively constant, and  $P(C_i)$  can be calculated as the ratio between the number of training instances in category  $C_i$  and the total number of training instances in the training sample. Therefore, for the purpose of choosing  $C$  according to Equation 1, it is only necessary to learn to calculate  $P(D | C_i)$ , and this is where the “Naïve” assumption comes into play.

Given a vocabulary of  $n$  terms (features)  $T_1, T_2, \dots, T_n$  selected from the training data, Naïve Bayes assumes that these terms be independent of context and position. Based on this independence assumption,  $P(D | C_i)$  can be approximated by  $\bar{P}(D | C_i)$ , which is calculated as the product of  $P(T_j | C_i)$ , the probability of term  $T_j$  appearing in category  $C_i$ :

$$\bar{P}(D | C_i) = \prod_{1 \leq j \leq n} P(T_j | C_i)^{\#(T_j, D)} \quad (4)$$

where  $\#(T_j, D)$  represents the frequency by which term  $T_j$  appears in document  $D$ . This formulation is essentially the multinomial model introduced in [16], [13] but with some combinatorial coefficients removed for simplification. Although the Naïve Bayes assumption does not hold for natural languages in general, in practice, this algorithm has been shown to work sufficiently well for many applications [12].

$P(T_j | C_i)$  is in turn approximated by the sample probability  $\bar{P}(T_j | C_i)$ , which is calculated according to Equation 5,

$$\bar{P}(T_j | C_i) = \frac{1 + \sum_k \#(T_j, D_{i,k})}{n + \sum_k \#(D_{i,k})} \quad (5)$$

where  $k$  ranges over all training documents belonging to category  $C_i$ ,  $\#(T_j, D_{i,k})$  the frequency by which term  $T_j$  appears in document  $D_{i,k}$ ,  $n$  the vocabulary size, and  $\#(D_{i,k})$  the total number of term occurrences in document  $D_{i,k}$ .

Due to the product of many terms<sup>3</sup> in Equation 4, underflow may occur. To prevent underflow, the actual implementation of Naïve Bayes in MALLET [15] uses the logarithm of  $\bar{P}(D | C_i)$  instead. Therefore,  $\bar{P}(T_j | C_i)$  cannot be zero. The **1** in Equation 5 is added to ensure that  $\bar{P}(T_j | C_i)$  is never zero.

MALLET’s implementation of Naïve Bayes uses the multinomial model, which has been shown to work better than the multi-variate Bernoulli model [16] for categorization tasks that have varying document lengths and large vocabularies.

#### IV. EXPERIMENTAL PROCESS AND RESULTS

In this study, we want to explore two research questions:

**RQ1** Will existing machine learning algorithms in general and Naïve Bayes in particular perform “sufficiently well” for categorizing API discussions? Since initially we did not know what to expect, we considered that a rather modest average test accuracy of 70% or above would be “sufficiently well” for practical use in Software Engineering.

**RQ2** If the answer to the first question is yes, we would like to further explore the major factors that impact the classification accuracy.

Since our goal is not to advance the state-of-art in machine learning, but to investigate the feasibility of using existing algorithms to solve particular software engineering problems, we chose to start with the Naïve Bayes algorithm [13]. This is because historically Naïve Bayes has been found performing “remarkably well” for many tasks [12], [14]. Although more recent studies have compared it with other algorithms [22], [10], because of the empirical nature of such comparisons, the results may not be universally true. Finally, if Naïve Bayes works reasonably well for our task, other more “superior” algorithms should only perform even better.

We chose to categorize the API discussions from the Swing Forum because it is a very active forum. Another advantage is that we can leverage the considerable prior research that we have conducted with the forum, e.g., [9], [19], so we can have greater confidence with the quality of our training data. We treat a whole discussion thread, which typically has multiple messages/posts, as a training document.

##### A. Data Collection

The most time-consuming task in text categorization is preparing training data [18]. In our case, unlike other studies [3], [8], the categories are domain-specific and do not exist beforehand. Therefore, we need to come up with both

<sup>3</sup>Even with stop words removed, our Version 3 training data produces a vocabulary of 9,144 words!

the categories and the training documents for each category. Moreover, to ensure data authenticity, we must clearly define each category. In addition, we were not sure about what would be the minimum number of documents for each category in order to achieve reliable, good-enough learning. Consequently, we have evolved three versions of single-labeled training data. Table I depicts the three versions of data, and Table II summarizes the classification accuracy for all of our experiments.

Version 1 was created quickly to test the Naïve Bayes algorithm. It contains ten categories and 46 discussion threads, with the minimum and maximum numbers of documents for each category being 3 and 10, respectively. As seen from Table II, the average test accuracy for Version 1 is low.

Version 2 increased the training sample size to 158 documents and 17 categories, with the minimum and maximum numbers of documents for each category being 6 and 13, respectively. Our goal was to investigate the effect of sample size on classification accuracy. Although the average test accuracy for Version 2 is not adequate either, the improvements in comparison with Version 1 are noticeable. Notice that as a result of refining and improving the definitions of the ten categories in Version 1, we have added seven new categories to Version 2, which are included for the sake of completeness.

In order to further confirm the effect of increased sample size on classification accuracy, we populated the “layout” category with 117 layout-related discussions that we have analyzed thoroughly in a prior study [19]. Because the result showed clearly the effectiveness of sample size for improving classification accuracy, we selected eight categories from Version 2 and aggressively increased the size of each of them to about 100 documents to form Version 3. Using Version 3, we were able to answer **RQ1** satisfactorily with an above-90% average classification accuracy and to investigate **RQ2**.

In general, in order to have sound judgments about label assignment in training data preparation, the experimenter must

- Possess sound knowledge for the subject matter to be categorized.
- Fully understand the content of a discussion. It is critical for ensuring the quality of our study and accurately assigning labels for documents. As a result, we carefully read through the discussions to understand the details contained in the collected documents.
- Understand that code quoted in a discussion may complement verbal descriptions. This is because the discussion participants may not always be able to accurately describe their problems, especially in the case of an API novice.
- Avoid the tendency to assign a category based on inferred information that is not described literally in the discussion, such as the fact that there is a certain bug.

### B. MALLET Commands and Evaluation Metrics

We used the MALLET machine learning toolkit for language processing [15]. Training data are organized as two-level file folders. Documents belonging to the same category are put under the same folder. The folders for all categories are in turn put under a top-level folder.

TABLE I: Three versions of training data collected from Java Swing Forum and their breakdown by categories

Data Version	Categories/Labels	#Documents	Total
V1.0	border	4	46
	dispose	3	
	drawing	3	
	focus	3	
	layout	8	
	action	10	
	icon	5	
	rendererEditor	4	
	title	3	
	others	3	
V2.0	borderAndMargin	13	158
	dispose	9	
	drawing	8	
	focus	9	
	layout	9	
	action	8	
	loadingIcons	8	
	rendererEditor	13	
	titleBar	6	
	titleBarFont	10	
	textIconPosition	10	
	dynamicHierarchy	10	
	defaultButton	11	
	mouseMotionPosition	12	
	threading	8	
	OOP	7	
	social	7	
V3.0	borderAndMargin	82	833
	dispose	103	
	drawing	108	
	focus	104	
	layout	125	
	titleBar	106	
	textIconPosition	96	
	dynamicHierarchy	109	

As the first step, MALLET imports and converts the training data into the vector space format that can be used for machine learning, indicating the frequencies of the terms in the text, with the following command:

```
% mallet import-dir --input v10/*
--output v10.mallet
```

This example converts Version 1 text stored in v10 into a file named “v10.mallet” to be used for machine learning.

The next step is to train a classifier with the data file created above, with the following command:

```
% mallet train-classifier --input
v10.mallet --training-portion 0.9
--num-trials 10000
```

Recall that when training a classifier with a machine learning algorithm, the training instances with pre-classified categories are split into two sets, the *training (-and-validation) set* and the *test set*, to be used for training and testing the classifier, respectively [20]. According to the `training-portion` option for the command above, MALLET will randomly split the training data into 90% training (-and-validating) instances for learning the classifier, and 10% for testing it. Furthermore, according to the `num-trials` option, this training-and-testing process will be repeated 10,000 times, each of which is called a trial.

The output for each trial includes a confusion matrix, each

```

Trial 9
Trial 9 Training NaiveBayesTrainer with 750 instances
Trial 9 Training NaiveBayesTrainer Finished
Trial 9 Trainer NaiveBayesTrainer training data accuracy= 0.9933333333333333
Trial 9 Trainer NaiveBayesTrainer Test Data Confusion Matrix
Confusion Matrix, row=true, column=predicted accuracy=0.9397590361445783
  Label  0  1  2  3  4  5  6  7  total
0 borderAndMargin  9  -  -  -  -  1  -  -  110
1 dispose  - 11  -  -  -  -  -  -  112
2 drawing  -  - 11  -  -  -  -  -  112
3 dynamicHierarchy -  -  -  8  -  -  -  -  18
4 focus  1  -  -  -  9  -  -  -  110
5 layout  -  -  -  1  - 13  -  -  114
6 textIconPosition -  -  -  -  -  -  7  -  17
7 titleBar  -  -  -  -  -  -  - 10  110
Trial 9 Trainer NaiveBayesTrainer test data accuracy= 0.9397590361445783

```

Fig. 1: Sample confusion matrix as a result of one trial with Version 3 (Training set of 750 documents; 83 test documents; 5 of the 83 test documents incorrectly classified -the off-diagonal cells-, resulting in a test accuracy of 93.98%.)

row of which represents the classification results for one category and whose cells show how the learned classifier labels the test instances of that category. Figure 1 depicts an example of a confusion matrix. For each row, the diagonal element shows the number of correct predictions, and the off-diagonal elements show the numbers of classification errors.

The test accuracy is defined as the ratio between the number of correctly labeled instances and the total number of test instances. After all trials are done, an overall measure of average test accuracy, TAM (Test Accuracy Mean), is calculated, as a metric for the accuracy of the learned classifier.

### C. Feature Selection

We have applied three feature selection methods, whose impact on classification accuracy is summarized by Table II.

1) *Stop Words Removal (SWR)*: Common, generic function words, such as “a”, “the”, “how”, “which”, “what”, and “where”, which are also known as stop words, do not contribute to the discrimination power needed for text categorization and, thus, can be removed from the vocabulary. MALLET provides two options to remove stop words from the input text during the importing process. The first option instructs MALLET to ignore a list of standard stop words that MALLET defines as default (524 common English adverbs, conjunctions, pronouns, and prepositions). The second option allows for adding extra stop words beyond the default.

The vocabulary size for Version 3 is 9,617, and 9,144 after stop words removal; apparently our data do not contain all the standard stop words.

2) *Words Splitting (WS)*: Source code is commonly quoted in API discussions. The lexical identifiers from the quoted source code usually contain useful information for defining the main topics of a discussion, which may complement the verbal description in important ways. For example, the appearance of a type name such as `JActionListener`, or a method call such as `addActionListener(aListener)` in a discussion, indicates that the discussion is at least partially about action handling.

However, due to the adoption of popular naming conventions such as Camel Cases and Underscoring, the lexical

TABLE II: Test Accuracy Mean (TAM) and the associated stddev for all the training-and-testing experiments with the three versions of training data, all in percentage. Best accuracy results for each version are highlighted in bold. The following feature selection methods are used: RAW, using the original training documents as is; SWR, Stop Words Removal; WS, Word Splitting.

Training Data	RAW	SWR	WS	SWR + WS
v1.0 - original	27.5, 18.1	37.0, 21.3	35.8, 20.0	46.3, 21.6
v1.0 - code	33.7, 20.5	34.9, 20.6	38.2, 20.3	41.7, 21.1
v1.0 - text	34.5, 20.3	<b>57.3, 21.3</b>	33.8, 20.3	56.3, 21.1
v2.0 - original	49.5, 12.3	60.4, 12.0	55.8, 12.3	<b>62.7, 11.7</b>
v2.0 - code	56.2, 11.8	57.1, 12.0	61.9, 11.7	61.4, 11.6
v2.0 - text	60.0, 12.6	62.2, 11.7	62.1, 11.6	62.2, 11.6
v3.0 - original	92.1, 3.0	<b>93.6, 2.6</b>	91.3, 3.0	91.8, 3.0
v3.0 - code	91.5, 3.0	92.1, 2.9	89.7, 3.2	89.6, 3.2
v3.0 - text	92.1, 2.9	93.0, 2.8	92.0, 3.0	92.7, 2.8

identifiers from source code cannot be used effectively for text categorization if they are used as is, because each of them will be counted as a different word. To address this problem, we have implemented a simple words splitting algorithm to preprocess the API discussions before they are sent to the learning algorithm. The words splitting algorithm, for example, would split “set\_value” into two words, “set” and “value”, “ImageIcon” into “image” and “icon”, “actionPerformed” into “action” and “performed”, and “button3” into “button” and “3”. Apparently, many of these words are meaningful for our text categorization task. The words splitting algorithm increases the frequencies of these words and, thus, may have some impact on the classification accuracy, which we have tested empirically.

3) *Using Code or Text Alone*: To experiment with the effects that code may have on classification accuracy, as shown in Table II, for each version of the training data, we also create a version that contains the quoted source code only and a version that contains the text only.

### D. Experimental Results and Observations

Our first research question **RQ1** is to find out if the Naïve Bayes algorithm can help us achieve an adequate classification accuracy. To seek a satisfactory answer for **RQ1**, we have created three versions of training data. From each version, we have extracted two more sets of training data, which consist of only code and only text from the original discussion documents, respectively. We tested each set of training data with four different feature selection methods (using the original documents as is (**RAW**), with stop words removed (**SWR**), with words splitting (**WS**), and with the combination of stop words removal and words splitting (**SWR+WS**). See Section IV-C for details on feature selection.

Table II shows a summary of all the experiments that we have run for **RQ1**. All experiments were run using the MALLET commands and evaluation metrics described in Section IV-B. For each experiment, we show the classification accuracy mean and the associated standard deviation, which measures the amount of variance in the resulting accuracy.

Based on the experimental results shown in Table II, we can make the following observations:

- 1) “Stop Words Removal” is always helpful for getting better training results. Its effect of improvement is more noticeable when applied to smaller training samples (Versions 1 and 2) and almost negligible for the large training sample (Version 3).
- 2) “Words Splitting” is helpful for getting better training results when applied to the smaller training samples (Versions 1 and 2). It does not help for the large training sample (Version 3).
- 3) When the sample size is small (Versions 1 and 2), both “Code Only” data and “Text Only” data help improve classification accuracy. However, they are not helpful for the large training sample, although their accuracies are very close to that of the original documents (Version 3).
- 4) Overall, an adequate training sample size is perhaps the single most effective factor for improving the test accuracy mean TAM.

#### E. A Theoretical Interpretation for Classification Accuracy

As shown in Section III, in a probabilistic learning framework, the goal is for the learning algorithm to estimate the actual conditional probability  $P(C_i | D)$  based on information obtained from the labeled training instances. In [6], Friedman analyzes in detail the ways by which *estimation bias* (the difference between the actual and the estimated conditional probabilities) and *estimation variance* (the variance of the estimated conditional probability) together influence classification error and accuracy. In particular, Friedman shows why under certain conditions, the classification algorithm may still work very well, despite the presence of significant estimation bias caused by the crude estimation that a simple algorithm such as the Naïve Bayes produces [6].

Friedman also concludes that estimation variance (the variance of the estimated conditional probability) is critical for controlling classification error/accuracy: The lower the variance, the higher the classification accuracy [6]. This theory of estimation variance is particularly important and relevant for this paper as it appears to be consistent with and can be used to explain our experimental results and observations in Section IV-D.

One important way to reduce probability estimation variance is to increase the training sample size [6]. This method has been used previously, but only empirically and implicitly without sound rationale, e.g., [16]. As can be seen from Table II, when the training samples (Versions 1 and 2) are not big enough, the classification accuracy tends to be low with higher variances (compare the standard deviations). By contrast, the largest training sample, Version 3, produces the highest classification accuracy with the smallest stddev.

“Stop Words Removal” helps reduce the size of the classifier’s vocabulary. As a result, its application would help reduce the estimation variance and, thus, increase classification accuracy (See Equation 4). However, as can be seen from

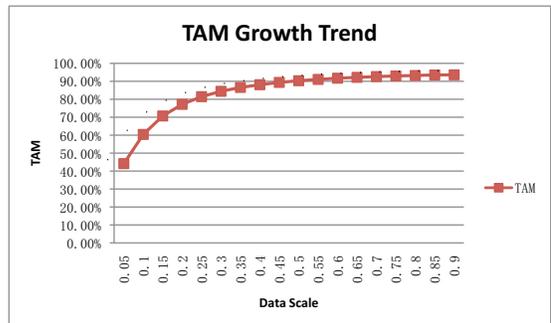


Fig. 2: Average Test Accuracy (TAM) as a function of training set size

Table II, its effect for classification accuracy is robust across all versions, but more noticeable for smaller samples.

The application of “Words Splitting” would increase the frequencies of certain words. For smaller samples, this seems to have helped reduce the estimation variance and, thus, increase classification accuracy. However, understandably, for the largest training sample, its application shows no positive effect for classification accuracy.

Like others [16], we have not applied stemming to our training samples. Based on Friedman’s theory [6], we predict that the effect of stemming will be similar to “Words Splitting” since in general, stemming also increases word frequencies.

#### F. Training Set Size versus Classification Accuracy

As shown in Table II, we were able to achieve an average test accuracy as high as 93.6% with the Version 3 training data. However, our experience with manually collecting 833 documents for Version 3 as well as that of others is that it is costly to create large, high-quality training data for machine learning [18]. Therefore, it becomes only natural to ask the question whether the 833 documents in Version 3 are all necessary and what is the minimal sample size that can still yield a comparable classification accuracy.

To find out, we ran the Naïve Bayes algorithm 18 times using the second command in Section IV-B, varying its training portion from 5% to 90% with an increase of 5% each time. Figure 2 depicts the relation between the training set size and Test Accuracy Mean (TAM). It shows that the TAM is increasing when the training data grow from 5% to 50%. However, after the training data portion reached 50%, that is, when 417 documents are used for training, the classifier tends to perform reliably with a TAM over 90%. It appears that the other 40%, or 333 documents, increase the TAM only by less than 4%. For our task, if an above-90% is considered acceptable and the cost of collecting training data is a concern, we would have been able to achieve an above-90% test accuracy mean with 480, or 60 documents per category.

#### G. Feature Word Frequencies versus Classification Accuracy

The highest average test accuracy in Table II is 93.6%, implying that there is an average error rate of 6.4%, or

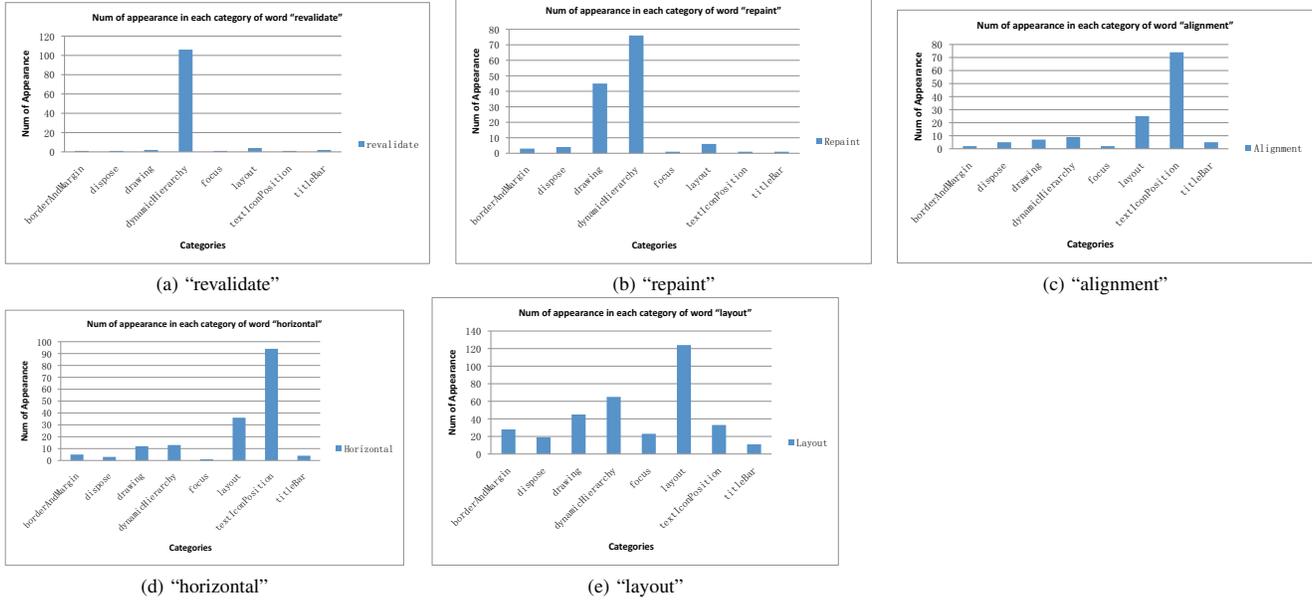


Fig. 3: Frequency distributions of five selected feature words over the eight categories in Version 3

that on average, five documents (6.4% of the 833\*10%, or 83 test instances) are incorrectly classified. To investigate what may have caused the errors as well as to confirm our theoretical understanding on how Naïve Bayes works (cf. Section III), we’ve selected five words that we suspect should have contributed higher discriminative power for separating the categories. We compare their occurrence frequencies in each category. As a result, we have found two related causes for prediction errors. One is that while some of these words are indeed highly discriminative, the more general case is that a category tends to be defined by multiple words, and the same word may appear in more than one category, leading to classification errors. The other finding is that a discussion may inherently involve multiple categories/topics. We were also able to demonstrate a solution to reduce the errors.

The five selected words are “revalidate,” “repaint,” “alignment,” “horizontal,” and “layout.” Figure 3 shows their frequency distributions over the eight categories.

- Discussions in Category `dynamicHierarchy` are about how to dynamically update a GUI, for which the API method `revalidate()` must be called in order to calculate the layout of a changed GUI hierarchy. Figure 3a shows that the word “revalidate” is a distinct feature that helps clearly separate Category `dynamicHierarchy` from others.
- The word “repaint” is related to `dynamicHierarchy` as well. The Swing API method `repaint()` must be called to make the changed GUI hierarchy visible on screen. However, as can be seen from Figure 3b, this word is commonly used in not only the “dynamicHierarchy” category, but also the “drawing” category.
- Discussions in Category `textIconPosition` are about how to manipulate the relative positions between the text and

the icon in a button or a label, which can be aligned either horizontally or vertically [9]. As a result, both words “alignment” and “horizontal” appear in this category. Figures 3c and 3d depict the occurrence distributions of these two words. However, people may also discuss how to align GUI components in other categories, especially in the “layout” category. Similarly, the word “horizontal” is often used in the layout category to describe the direction of laying out a set of components.

- Discussions in the layout category are about how to compose and position multiple widgets in order to make a graphical user interface [19]. Figure 3e depicts the frequencies by which the word “layout” appears in the eight categories. Interestingly, different from the other four, in addition to Category `layout`, the word `layout` also appears in all of the other seven categories.

Why is the word `layout` contained in all eight categories in Figure 3e? This is likely due to the fact that `layout` is a basic feature required by almost all Swing programs. Even for a discussion that is not centered on layout, as long as it quotes some source code, the word `layout` is likely to be included. Furthermore, the word may also be used when the posters explain how the program under discussion works.

We have also observed that documents from other categories are most often mistakenly classified into the layout category. We hypothesize that this is due to the combined effects of the common occurrences of the word `layout` and the quoted source code. In particular, to test the hypothesis that the quoted code may have caused documents from the other categories to be categorized as layout, we created a new dataset for Version 3, where we used the “Text Only” data for the layout category in order to remove the confusion due to the quoted code. We

TABLE III: Test Accuracy Mean (TAM) and the associated stddev for the N-label Naïve Bayes Classifier

#labels	TAM, stddev
N = 2	97.8, 1.6
N = 3	99.3, 0.9
N = 4	99.7, 0.5

trained a new Naïve Bayes classifier with stop words removed. When tested, the new classifier yielded an improved average test accuracy of 94.1%, which is 0.48% higher than the best average test accuracy 93.6% in Table II.

#### H. Multi-label Text Categorization

So far, we have focused on the case in which exactly one category is assigned to each document, which is often called the *single-label* (aka *non-overlapping categories*) case [20]. In Section IV-G, we have also seen the case in which multiple categories may be assigned to the same document, which is dubbed the *multi-label* (aka *overlapping categories*) case [20].

When treated as a single-label case<sup>4</sup>, a document that inherently involves more than one category (topic) would naturally cause a prediction error. In Version 3 of our training data, we have 833 documents, of which 10% (83 documents) are used for testing. Since the highest Test Accuracy Mean (TAM) is 93.6% (Table II), in each trial, there are on average about 5 documents (6.4% \* 83) wrongly predicted. To investigate the impact that the multi-label problem may have on classification accuracy, we have modified the MALLETT’s Naïve Bayes algorithm to output the top-N categories that yield the highest conditional probabilities  $\bar{P}(C_i | D)$ , rather than only the top category with the highest probability. Under this modification, a classification decision is considered correct as long as the N categories that it predicts include the manually assigned label for the test document.

Table III shows the classification accuracy for N equals 2, 3, and 4, respectively. It appears that small values for N are enough to drive the prediction accuracy to a nearly perfect score. If this is also true for any arbitrary number of categories, it would imply that our multi-label approach may be acceptable for practical use; if out of every N documents there is at least one relevant, a software developer would not mind to inspect a few additional documents so long as he or she is sure that at least one of them is relevant.

#### I. Two Examples of Multi-label API Discussions

In this section, we present two sample API discussions to illustrate the multi-label phenomenon. To simplify presentation, we have edited the discussions to highlight only the most relevant pieces. However, the two original discussions are also available online for further inspection<sup>5 6</sup>.

The first sample discussion involves both the layout category and the drawing category. It was manually labeled as layout, but the learned Naïve Bayes classifier classified it as drawing.

<sup>4</sup>Our initial goal was to label each thread by a single, core topic / problem.

<sup>5</sup><https://forums.oracle.com/forums/thread.jspa?messageID=5865516>

<sup>6</sup><https://forums.oracle.com/forums/thread.jspa?messageID=5861973>

In the first message shown next, the OP (original poster) asks for a solution and also posts code for other participants to review. The original task is to figure out the reason why his “canvas” and buttons are displayed on the left hand side. From the code we can see the OP was using “borderLayout” and trying to add panels to the mainContainer. But since the OP does not set the border layout correctly, the program positions the “canvas” in the wrong place. Therefore, the OP’s problem is about how to correctly set up a layout manager. So this is indeed a “layout” topic.

Message Title: Layout Problems

Can someone help me understand why my canvas here shows up on the left hand side along with my buttons.

However, in addition to layout-related words, the posted code also includes a set of keywords that are commonly used when describing drawing issues, such as repaint, Graphics, setX, draw, and repaint. So the discussion is also about drawing, although the central topic is layout instead of drawing.

```
public class Painter extends JFrame ...{
    ...
    // Adding panels
    mainContainer.setLayout BorderLayout;
    mainContainer.add Buttons, BorderLayout;
    mainContainer.add BorderLayout CENTER;
    ...
    void clear(){ repaint(); }

    public actionPerformed (ActionEvent e){
    ... repaint(); ...
    }

    public mousePressed (MouseEvent e){
    myShape.setX1(e.getX());
    myShape.setY1(e.getY());
    }

    public paint(Graphics g){
    g.setColor(line);
    myShape.draw(g);
    }
}
```

The last post in the discussion, which is shown next, would contribute more feature words to the layout topic, e.g., center, layout, north, and components.

Re: Layout Problems

...
well, since you dont' seem to put anything in the ``canvas'', I dont' see how you know that it's doing what you say...

I can tell you that since it's in the

center of the border layout, it fills whatever space isn't used by the north, south, east and west components (if any).

Our second sample API discussion involves both the topics of `borderAndMargin` and `rendererEditor`. The discussion was manually labelled as `rendererEditor`, but the classifier assigns it to the `borderAndMargin` category. The OP's task is to find out how to enhance a selected cell in a `JTable`, which can be implemented by customizing the renderer of the `JTable`.

```
Message title: cell renderer with border
...
Dear friends,
every time I apply a cell render over a
JTable, the selected row is not enhanced
... why ?
```

The next piece of code posted would contribute to the `borderAndMargin` topic due to the use of feature words such as `setBorder` and `BorderFactory`:

```
Re: cell renderer with border
...
into the tutorial I found the following:
if (isBordered) {
    if (isSelected) {
        if (selectedBorder == null) {
            selectedBorder = BorderFactory.
                createMatteBorder(2,5,2,5,
                table.getSelectionBackground());
        }
        setBorder(selectedBorder);
    } else {
        if (unselectedBorder == null) {
            unselectedBorder = BorderFactory.
                createMatteBorder(2,5,2,5,
                table.getBackground());
        }
        setBorder(unselectedBorder);
    }
}
```

Thats working fine, but it only enhance the row, not the selected cell. ??  
How to obtain the ``normal'' selection effect - i.e., that one where I not using renderers?

But the next piece of code posted clearly contributes to the `rendererEditor` category:

```
Re: cell renderer with border
...
How to obtain the ``normal'' selection
effect - i.e., that one where I not using
renderers?
```

Extend the `DefaultTableCellRenderer`:

```
class MyRenderer extends
DefaultTableCellRenderer
{
    public Component
    getTableCellRendererComponent(...)
    {
        super.getTableCellRendererComponent...
        // add your code here
        return this;
    }
}
```

The last two messages would also contribute to the `borderAndMargin` topic due to the occurrences of feature words such as `alignment`, `setAlignmentX`, and `setHorizontalAlignment`:

```
Re: cell renderer with border
...
thank you very much, everything is
``almost ok'' now :)
the problem now is the cell alignment...
Im trying:
setAlignmentX(JLabel.RIGHT_ALIGNMENT);
but the cell doesnt change it alignment
in runtime... why ?
```

```
Re: cell renderer with border
...
I believe that should be
setHorizontalAlignment(JLabel.RIGHT);
```

#### J. Experimentation with Support Vector Machines

Although we have primarily focused on an in-depth investigation of the Naïve Bayes, it is also relevant to compare it with other machine learning algorithms, such as Decision Trees and SVM (Support Vector Machines). SVM, in particular, has been shown to be very effective for text classification [10]. Since our Version 3 data is readily available, we have also tried out SVM for a quick comparison. In general, SVM is shown to outperform Naïve Bayes by one to four percent in terms of the average test accuracy (compare Tables II and IV).

The Naïve Bayes is a multi-category classifier (eight in our case), whereas SVM classifiers are binary. As a result, we have trained one SVM classifier for each of the eight categories. We took 10 percent of instances from each category to make a test set, and used the other 90 percent to make the training datasets. We applied the leave-one-out cross validation approach.

When making a training dataset for each SVM classifier, the negative instances far outnumber the positive instances, so such a dataset is considered *imbalanced* [2]. When trained with such imbalanced datasets, the performance of SVM has been shown to drop significantly [2]. To create a balanced dataset, we have preprocessed the data by *oversampling* the positive class (duplicating the positive instances such that their number matches that of negative instances) and *undersampling* the negative classes (proportionally reducing the size

TABLE IV: Performance of eight binary SVM classifiers trained with three datasets derived from Version 3: Imbalanced, Undersampling, and Oversampling. Each cell contains Accuracy/Precision/Recall on test set for the respective binary classifier.

Classifiers	Imbalanced	Under-sampling	Over-sampling
borderAndMargin	93.41% / 75.00% / 60.00%	87.91% / 47.06% / 80.00%	98.99% / 100.0% / 90.91%
dispose	95.60% / 100.0% / 60.00%	86.81% / 45.45% / 100.0%	96.97% / 100.0% / 72.73%
drawing	98.90% / 100.0% / 90.91%	97.80% / 90.91% / 90.91%	99.07% / 100.0% / 90.91%
focus	96.70% / 100.0% / 72.73%	96.70% / 90.00% / 81.82%	96.97% / 90.00% / 81.82%
layout	93.41% / 85.71% / 54.55%	92.31% / 66.67% / 72.73%	96.00% / 76.92% / 90.91%
titleBar	93.41% / 72.73% / 72.73%	93.41% / 72.73% / 72.73%	94.05% / 75.00% / 81.82%
textIconPosition	96.70% / 100.0% / 72.73%	100.0% / 100.0% / 100.0%	98.98% / 100.0% / 90.91%
dynamicHierarchy	95.60% / 100.0% / 63.64%	81.32% / 39.29% / 100.0%	98.98% / 100.0% / 90.91%
<b>average</b>	<b>95.47%</b> / 91.68% / 68.41%	<b>92.03%</b> / 69.01% / 87.27%	<b>97.50%</b> / 92.74% / 86.37%

of each negative class such that the sum of all negative classes matches that of the positive class). Table IV depicts the accuracy/precision/recall of the SVM binary classifiers trained with the three datasets (imbalanced, undersampling, and oversampling). As shown in Table IV, oversampling is the best strategy that yields the highest average test accuracy, whereas undersampling the worst.

#### V. CONCLUSION AND FUTURE WORK

Software libraries and APIs are important productivity tools, but they are difficult to use due to their extensive content and rich details. As a result, developers have widely used online software forums to exchange information and seek help to solve problems in using APIs. Tens of thousands of discussions have been archived for popular frameworks such as AWT/Swing, which can be referenced by future developers to improve productivity. However, manually finding relevant discussions from the vast amount of discussion threads is a painstaking task for the user. Therefore, it can be useful to automatically classify these discussions into meaningful semantic categories so as to facilitate searching and browsing.

To address this problem, we have explored using the Naïve Bayes algorithm to categorize API discussions into API-specific topics. In our case study, we analyzed over 1,000 discussion threads from the Java Swing Forum. We labeled these API discussions with predefined semantic categories based on their textual content. We've built three groups of training data consisting of 46, 158, and 833 documents, respectively. With our largest training data set (833 documents over eight categories), we have successfully trained Naïve Bayes classifiers that achieve the highest average test accuracy of 94.1%. In addition, we investigated the impacts of various feature selection methods on classification accuracy, including stop words removal, words splitting, "Code Only," and "Text Only." We found that the size of training set is a key factor for improving classification accuracy. We have also identified that multi-label documents are a main cause for classification errors and proposed one solution. Lastly, we investigated why Naïve Bayes works so well by inspecting the frequency distributions of selected words from our training data, and by seeking theoretical interpretation from the machine learning literature.

Future work should apply the same process to other APIs and with larger numbers of categories to demonstrate both the generality and scalability of this approach.

#### REFERENCES

- [1] stackoverflow. [Online]. Available: <http://stackoverflow.com/>
- [2] R. Akbani, S. Kwek, and N. Japkowicz, "Applying support vector machines to imbalanced datasets," in *ECML*, 2004, pp. 39–50.
- [3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *CASCON*, 2008, pp. 23:304–23:318.
- [4] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE*, 2006, pp. 361–370.
- [5] A. Bacchelli, T. D. Sasso, M. D'Ambrosio, and M. Lanza, "Content classification of development emails," in *ICSE*, 2012, pp. 375–385.
- [6] J. H. Friedman, "On bias, variance, 0/1-loss, and the curse-of-dimensionality," *Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 55–77, Jan. 1997.
- [7] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *ASE*, 2011, pp. 323–332.
- [8] A. Hindle, D. M. German, R. C. Holt, and M. W. Godfrey, "Automatic classification of large changes into maintenance categories," in *ICPC*, 2009, pp. 30–39.
- [9] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *ICPC*, 2011, pp. 91–100.
- [10] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *ECML*, 1998, pp. 137–142.
- [11] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.
- [12] P. Langley, W. Iba, and K. Thompson, "An analysis of bayesian classifiers," in *AAAI*, 1992, pp. 223–228.
- [13] D. D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *ECML*, 1998, pp. 4–15.
- [14] D. D. Lewis and M. Ringuette, "A comparison of two learning algorithms for text categorization," in *Third Annual Symposium on Document Analysis and Information Retrieval*, 1994, pp. 81–93.
- [15] A. McCallum and A. Kachites. (2002) MALLET: A machine learning for language toolkit. [Online]. Available: <http://mallet.cs.umass.edu>
- [16] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *AAAI-98 Workshop on "Learning for Text Categorization"*, 1998.
- [17] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on CVS," in *ESEM*, 2010, pp. 6:1–6:10.
- [18] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell, "Text classification from labeled and unlabeled documents using EM," *Machine Learning*, vol. 39, pp. 103–134, 2000.
- [19] C. R. Rupakheti and D. Hou, "Evaluating forum discussions to inform the design of an API critic," in *ICPC*, 2012, pp. 53–62.
- [20] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, Mar. 2002.
- [21] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010, pp. 45–54.
- [22] Y. Yang and X. Liu, "A re-examination of text categorization methods," in *SIGIR*, 1999, pp. 42–49.

# An Empirical Study of API Stability and Adoption in the Android Ecosystem

Tyler McDonnell, Baishakhi Ray, Miryung Kim  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin  
 Austin, TX, USA

Email: tscottmcdonnell@gmail.com, rayb@utexas.edu, miryung@ece.utexas.edu

**Abstract**—When APIs evolve, clients make corresponding changes to their applications to utilize new or updated APIs. Despite the benefits of new or updated APIs, developers are often slow to adopt the new APIs. As a first step toward understanding the impact of API evolution on software ecosystems, we conduct an in-depth case study of the co-evolution behavior of Android API and dependent applications using the version history data found in github.

Our study confirms that Android is evolving fast at a rate of 115 API updates per month on average. Client adoption, however, is not catching up with the pace of API evolution. About 28% of API references in client applications are outdated with a median lagging time of 16 months. 22% of outdated API usages eventually upgrade to use newer API versions, but the propagation time is about 14 months, much slower than the average API release interval (3 months). Fast evolving APIs are used more by clients than slow evolving APIs but the average time taken to adopt new versions is longer for fast evolving APIs. Further, API usage adaptation code is more defect prone than the one without API usage adaptation. This may indicate that developers avoid API instability.

## I. INTRODUCTION

Over the course of the past few years, the mobile application arena has exploded with the dissemination of affordable and powerful smart phones. As of 2012, the Apple App Store and Google Play Store boast a combined 1.5 million available apps and 55 billion app downloads worldwide [27]. Google and Apple support mobile app development with their own operating systems and associated Application Programming Interfaces (APIs). These APIs give developers access to features like location services, wi-fi connections, bluetooth functionality, and graphics.

When APIs evolve to accommodate new feature requests, to fix bugs, to meet new standards, and to provide higher performance, client applications often need to make corresponding changes to use new or updated APIs. Despite the benefits of new or improved APIs, developer adoption is often slow among client applications. For example, the Android Operating System is evolving fast, yet API adoption is slow and the consumer pool is fragmented by the Android version numbers [5].

Though many techniques have been proposed to ease library migration and to address API version incompatibilities, API evolution and its associated ripple effect throughout software ecosystems are still under-studied. For example,

Robbes et al. [21] studied how client applications react to API evolution in libraries or frameworks, but the study was confined to the issue of API deprecation in Smalltalk.

As a first step towards understanding the impact of API evolution on developer adoption, we conduct an in-depth case study of the Android API and dependent applications. Using the version history data of Android applications found in github and the API evolution data derived from Android OS documentation pages, we quantify their co-evolution behavior. We analyze the average time between API updates and record the number of method and field changes in each Android version. We also track changes in each major feature of Android. On the side of client applications, we calculate the percentage of Android API method calls and field references and categorize them by the API version number. By comparing the commit time of an API reference in client code against the release time of newer APIs, we identify outdated API usages and measure the *lagging time*—how far existing API references are lagging behind newly released APIs. We also measure the *propagation time*—the time taken for the client code to adopt new API usages. Then we correlate these adoption statistics with the frequency and location of evolving APIs in Android OS.

By characterizing the co-evolution behavior of APIs and dependent applications, we address the following research questions. Our findings are summarized as follows:

- **How fast does the Android API change, and which parts change the most?** Android APIs are evolving at the rate of 115 API updates per month on average. APIs related to hardware, user interface, and web are evolving much faster than others.
- **How dependent is client code on Android APIs, and how long does it take to adopt new APIs?** Around 25% of all method and field references in the client code use the Android APIs. However, application developers are hesitant to adopt new APIs. On average, 28% of Android API calls are lagging behind the latest released version. 22% of outdated APIs eventually upgrade to use newer APIs; nevertheless it takes a considerable amount of time, 14 months, on average.
- **What is the relationship between API stability, usage, adoption, and bugs?** Fast evolving APIs are used more by clients than slow evolving APIs. However, the

pace of client update is slower for fast evolving APIs. Files which are changed to use new APIs are more defect prone than files without API usage adaptation. This may imply that developers avoid frequent upgrades to unstable or rapidly evolving APIs.

To the best of our knowledge, we are the first to quantify the co-evolution behavior of Android and mobile applications and to confirm that client adoption is not keeping pace with API evolution. We are the first to find that API updates are more defect prone than other types of changes by investigating the relationship between API instability and bugs in client code as opposed to a library. Our study shows that fast-evolving APIs are used more and adopted more, but the time taken for API adoption is longer. Though many tools exist to automate API usage updates in client code, these tools are inadequate for promoting adoption alone as various stakeholders affect the process of API adoption. We call for further studies on how to promote API adoption and ultimately facilitate the growth of software ecosystems.

## II. RELATED WORK

**Empirical Studies of API Evolution.** In this paper, we seek to understand developer response to evolving APIs. Several studies analyze different software ecosystems and attempt to assess the *ripple effect* that API changes may have on client applications. Dig and Johnson found that 80% of the code changes that break client-side code are API refactorings [10]. Similarly, Xing and Stroulia studied Eclipse evolution history and found that 70% of structural changes are due to refactorings and existing IDEs lack support for complex refactorings [28]. In our study of API evolution, we examine the relationship between API stability and the degree of adoption measured in propagation and lagging time, which have not been investigated before in the above studies. Hou and Yao study the Java API documentation and find that a stable architecture played an important role in supporting the smooth evolution of the AWT/Swing API [15].

In a large scale study of the Smalltalk development communities, Robbes et al. found that only 14% of deprecated methods produce non-trivial API change effects in at least one client-side project; however, these effects vary greatly in magnitude. On average, a single API deprecation resulted in 5 broken projects, while the largest caused 79 projects and 132 packages to break [21]. In contrast to Robbes et al., our study is not limited to API deprecation and we focus on applications written in Java as opposed to Smalltalk. The mobile software arena may also differ from other applications by placing the burden on developers to support users running a wide variety of devices and different OS versions.

Kim et al. investigate the relationship between API refactorings and bugs and find that the number of bug fixes increases after API refactorings [16]. Weißgerber and Diehl also find that API refactorings often occur together with

other types of changes and that API refactorings are followed by an increasing number of bugs [26]. These studies investigate the relationship between API refactorings and bugs in libraries only, as opposed to bugs in clients.

Yau et al. [29] and Black [7] investigate the ripple effects of evolving software, but these studies focus on a single system, as opposed to the impact of evolving APIs on clients. In this paper, we investigate how mobile applications react to API changes in the Android *ecosystem*, following Lungu et al.'s definition—*a collection of software projects which are developed and co-evolve in the same environment* [18]. **Techniques for Easing API Migration.** Several techniques can help programmers deal with broken code as a result of API evolution. Henkel and Diwan and Ekman and Asklund record API refactorings performed in an IDE in order to replay them in the client applications [13], [14]. Dig et al. present a refactoring-aware version control system to account for refactoring edits during the version merging process [11]. Chow and Notkin suggest how to adapt client applications using API usage adaptation rules written by developers [8]. Dig et al. adopt a proactive approach and create a layer between clients and each updated library version [12]. This approach has the advantage of preventing broken code with no client-side effort, but it can discourage the use of new functionality of upgraded APIs.

Other techniques recommend API replacement using various types of underlying analyses: lexical comparison of method signatures syntactic and semantic similarity of APIs, shingles analysis, analysis of how a library's internal API usage changes between versions, analysis of code comments and release documents, source code implementation details, analysis of how other developers adapted their code, and combination of method signatures and call usages. A survey of techniques for easing API migration is found elsewhere [9], [17]. Cossette and Walker found that, while most broken code may be mended using one or more of these techniques, each is ineffective when used in isolation [9].

**Studies on Android Applications.** Shabtai et al. is the first to conduct a formal study of Android Packages files (APK) [24]. They apply machine learning techniques to classify applications into two categories: tools vs. games. Syer et al. [25] compare the source code size, churn, and dependency characteristics of mobile applications for the Android platform against those for the Blackberry platform. Ruiz et al. investigate the extent of reuse in the Android Market using Software Bertillonage techniques to track code across mobile applications [22]. Sanz et al. [23] detect malicious Android applications in the Android market. Our study differs from these studies by investigating the impact of evolving APIs on adoption. By analyzing bug reports and developers' discussion, Pathak et al. [20] find that 20% of overall energy related bugs in Android occur after an OS update. This may explain our study finding that the number of bugs increases in Android applications following an API

update.

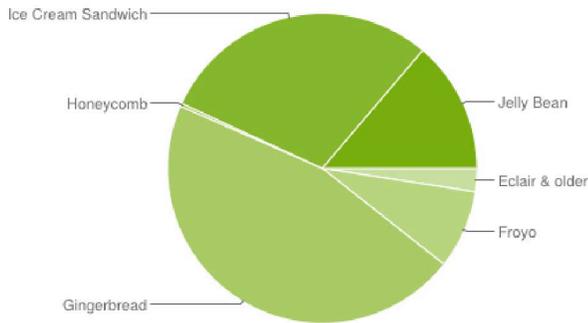
### III. STUDY METHOD

Section III-A describes the Google Android API, its evolution history, and developer pool. Section III-B describes the client applications we selected as subjects.

#### A. Android API

Android is an open source Linux-based operating system owned by Google and designed for touchscreen mobile devices such as smart-phones and tablet computers.

The rapid commercial growth of the mobile sector is coupled with similarly fast-paced hardware and software evolution. Google released the first Android version (Android 1.0, API level 1) on September 23rd 2008. Since then a new version is released approximately in every 3 months, till the release of the current version, (Android 4.2, API level 17) on November 13th 2012. Figure 1 shows the API version history, including release dates, version numbers, associated API levels, and codenames [4].



Version	Codename	API	Dist.	Release Date
1.0	None	1	*	Sep 23, 2008
1.1	None	2	*	Feb 9, 2009
1.5	Cupcake	3	*	Apr 30, 2009
1.6	Donut	4	0.2%	Sep 15, 2009
2.0	Eclair	5	*	Oct 26, 2009
2.0.1	Eclair	6	*	Dec 3, 2009
2.1	Eclair	7	8.1%	Jan 12, 2010
2.2	Froyo	8	8.1%	May 20, 2010
2.3-2.3.2	Gingerbread	9	0.2%	Dec 6, 2010
2.3.3-2.3.7	Gingerbread	10	45.4%	Feb 9, 2011
3.0	Honeycomb	11	*	Feb 22, 2011
3.1	Honeycomb	12	0.3%	May 10, 2011
3.2	Honeycomb	13	1.0%	Jul 15, 2011
4.0-4.0.2	Honeycomb	14	*	Oct 19, 2011
4.0.3-4.0.4	Ice Cream Sandwich	15	29.0%	Dec 16, 2011
4.1	Jelly Bean	16	12.2%	Jul 9, 2012
4.2	Jelly Bean	17	1.4%	Nov 13, 2012

data provided by Google  
 \* indicates that no distribution data is available

Figure 1. Client API Usage

Because the Android operating system runs on a wide variety of different devices, the consumer pool is fragmented by Android version numbers. Older mobile devices often ship with earlier versions of the Android OS and may be incapable of updating to the most recent Android API due to

hardware limitations. Figure 1 shows the current breakdown of Android version usage based on devices that accessed Google Play within a 14-day period in January 2013 [4]. Notably, this distribution shows either a hesitancy towards newer versions or at least a slow adoption trend, with 50% of users running a version released 20 or more months prior to the current version. Many analysts note how little change they have seen in the market share of the dominant Android version in the past 12 months and how this fragmentation differs from Apple’s iOS [6].

For this study, we correlate changes in client applications with changes in Android OS. We build a data structure to store the Android API version history. We analyze the `api-versions.xml` file and the `apidiff` directory that Google provides to developers along with the Android Software Development Kit (SDK). The `api-versions.xml` file provides a complete listing of all Android classes, methods, and fields available in API version 1. The `apidiff` directory is a set of `html` files, cataloging changes to any classes, methods, or fields from the previous API version.

#### B. Study Subjects

We use *active* open source Android applications with the following traits in `github`: over 1000 commits, 5 or more authors, 100 or more line changes per commit, and at least one commit made in 2012.

**CP Congress Tracker** is an app that allows users to manage a personal schedule, locate opinion leaders, and provide general coverage of related congressional events. **Apollo Music Player** is a customizable lightweight Android music app. **Cyanogen** is a set of multimedia and user interface suites. It allows users to place widgets like analog clocks on the home screen of their phone. **Google Play Analytics** allows users access real-time Google Analytics profiles. **LastFM** is a music listening and sharing application. **mp3Tunes** allows users to access and listen to the songs in iTunes. **OneBusAway** is a mobile app that provides real-time arrival information for Seattle area buses. **ownCloud** allows users to access and share files stored in the cloud. **RedPhone** is an app for making secure calls by providing end-to-end encryption. **XMBCremote** is a full featured remote control software for the XMBC media center. Table I shows the last updated time, the number of revisions, change rate, the number of authors, and code size.

### IV. STUDY RESULTS

Section IV-A presents the extent and characteristics of Android API evolution. Section IV-B investigates how fast client code is adapting the updated APIs. Section IV-C analyzes the impact of API evolution on client code.

#### A. Characteristics of Android API Evolution

**RQ1. How fast do Android APIs evolve?** We first identify added, changed, and removed APIs for each Android version. When a new API version is released, Google provides

an html file documenting all API changes in each release [4]. We built a tool that extracts API change information from the html file and stores the API Version history data. We define a changed class as one that is either new in the particular version or has at least one changed, added, or removed API method or field. Similarly, changed methods and fields are the ones with a modified signature since the previous version. Removed methods and fields are those that existed in the previous version but no longer do in the current version. Table II shows the extent of API evolution at the class, method, and field granularity.

To measure the rate of Android API evolution, we compute how many APIs are updated in each month on average.

$$avg. \text{ update rate} = \frac{\sum_{releases} \# \text{ API updates}}{\# \text{ months}}$$

In each month, 44 methods are changed, 11 methods are added, 51 fields are changed, 9 fields are added, and less than one method or field is removed on average. Android is constantly evolving to include more method and field variables, with existing methods and fields frequently changed to add new functionality. However, removal of existing functionalities is rare.

## RQ2. What functions of Android API are updated

Table I  
CHARACTERISTICS OF CLIENT MOBILE APPS

Apps	Updated	Rev	Rev/mo	Author	LOC
Congress Tracker	04-15-2013	1359	25.6	7	13349
Apollo M	03-24-2013	9	0.4	1	15783
Cyanogen	01-10-2011	109	2.3	20	28972
Google A	03-12-2013	926	77.1	23	52932
LastFM	03-03-2013	212	8.2	7	9771
mp3Tunes	02-17-2013	104	2.2	1	9608
OneBusAway	03-09-2013	497	33.1	5	51784
ownCloud	04-12-2013	665	55.4	12	25109
RedPhone	03-23-2013	116	4.8	5	21315
XMBcremote	04-05-2013	928	19.3	24	92893

Table II  
API CHANGES IN ANDROID PER VERSION AND EVOLUTION RATES

API Version	Release Date	Class		Methods				Fields	
		Δ	Δ	+	-	Δ	+	-	
3	Apr 30, 2009	246	368	60	0	296	68	0	
4	Sep 15, 2009	128	70	41	1	208	27	0	
5	Oct 26, 2009	187	199	64	0	234	205	0	
6	Dec 3, 2009	37	0	2	0	7	1	0	
7	Jan 12, 2010	61	52	2	0	22	3	0	
8	May 20, 2010	191	200	38	1	195	23	0	
9	Dec 6, 2010	244	348	42	9	141	11	0	
10	Feb 9, 2011	46	7	0	0	10	0	0	
11	Feb 22, 2011	263	416	95	7	619	36	0	
12	May 10, 2011	118	73	27	1	87	9	0	
13	Jul 15, 2011	69	22	11	0	68	1	0	
14	Oct 19, 2011	269	271	98	8	405	34	0	
15	Dec 16, 2011	84	25	3	0	38	2	0	
Min		37	0	0	0	7	0	0	
Max		269	416	98	9	619	205	0	
Mean		149	158	37	2	179	32	0	
Rate (Total update/month)		42	44	11	<1	51	9	0	

**most?** We investigate the areas of the Android API that are updated most frequently. From Android packages, we select certain *keywords* to characterize API features. For example, the taxon *text* is drawn from all packages relevant to rendering or tracking text on an Android device: `android.text.format`, `android.text.method`, `android.text.style`, and `android.text.util`. We create 25 taxa using various keywords such as: animation, bluetooth, database, graphics, io, os, security, and text. We then categorize each new, changed, or removed API method and field for each taxon. Table III shows which taxa are updated most frequently.

Table III  
API CHANGE DISTRIBUTION PER TAXON (FEATURE)

Taxon	Total Updated Versions	Total Updated APIs	Avg. Changes Per API Release	Avg Update Interval (Month)
animation	7	37	5	5.4
appwidget	3	12	4	12.7
bluetooth	5	9	2	7.6
content	10	179	18	3.8
database	6	100	17	6.3
gest	1	3	3	38.0
graphics	10	84	8	3.8
hardware	10	121	12	3.8
io	2	18	9	19.0
location	4	38	10	9.5
media	8	93	12	4.8
net	8	87	11	4.8
opengl	5	10	2	7.6
os	11	94	9	3.5
rtp	0	0	0	
security	2	25	13	19.0
sip	1	2	2	38.0
support	0	0	0	
telephony	5	49	10	7.6
test	8	70	9	4.8
text	9	147	16	4.2
util	6	180	30	6.3
view	12	546	46	3.2
webkit	10	172	17	3.8
wifi	4	14	4	9.5

We find that the most frequently evolving packages (the lowest average time between API changes) include content, graphics, hardware, os, view, and webkit—each updated in 10 or more of the 14 API releases under consideration. We also find that content, hardware, text, util, view, and webkit have more than 100 API changes since the original Android release.

The high frequency of API updates related to hardware, graphics, and views may be due to the Android hardware fragmentation. In contrast to iOS, which supports five unique devices, there are at least 170 Android devices. Google may rapidly update the hardware and graphics APIs to support widely-varying hardware features.

### B. Characteristics of API adoption by the client programs

This section investigates how client programs respond to API evolution. We study the degree of dependence that client

applications have on Android APIs and how fast clients are adopting new or updated APIs.

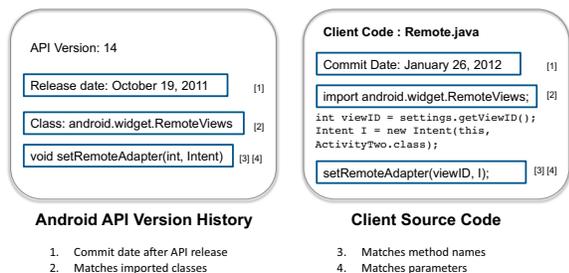


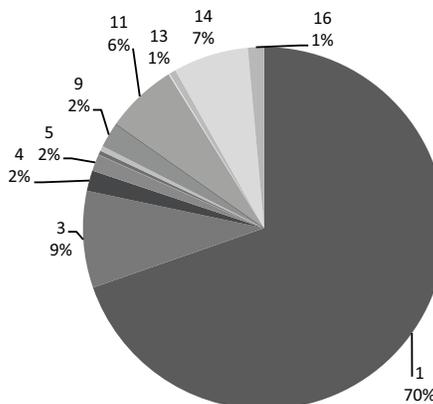
Figure 2. Identifying Android References in Client Source Code

### RQ3. How dependent is client code on Android APIs?

We analyze client application source code to measure the degree of dependence on Android APIs. We identify all Android methods and fields references using a syntax-based lexical search on Java source files. By analyzing the *import* statements, we detect the Android classes referenced in each Java file. For each referenced API method call or field access, we search through our Android API Version History data structure (see Section IV-A) to find a corresponding API invocation with a corresponding API declaration based on a method name and the number of parameters. We detect API usage updates by monitoring changes to the used method name, the number of arguments, and argument names. We also use the commit date of a source file to determine the most recent available API version. For example, Figure 2 shows an Android API method invocation `setRemoteAdapter(int, Intent)` in client code. When scanning the client source file, we find an entry in the Android API Version History data structure of method `setRemoteAdapter`. By matching the number of parameters, the release date of the API entry, the commit date of client code, and the list of imported classes in the source file, we infer that the client code is using the API version 14 for method `setRemoteAdapter(int, Intent)`.

By measuring the proportion of Android API method calls and field references out of all references, we investigate how dependent client apps are on Android APIs (see Figure 3). Approximately 25% of all method and field references in client code are about Android APIs. Around 80% of the references in the most recent version of client code refer to Android API Version 1 or Version 3, released in September 2008 and April 2009 respectively. These results show that though mobile apps are heavily dependent on Android OS and its functionality, developers are hesitant to embrace or fully utilize more recent API features.

**RQ4. What is the lag time between client code and the most recent Android API?** We detect the *lag time*



Client Applications	Android API	Total API	% Android API	Unique Android API
Congress Tracker	1007	3396	30%	82
Apollo Music	1332	3820	35%	155
Cyanogen	1439	5992	24%	144
Google A	3164	12145	26%	336
LastFM	371	2122	16%	64
mp3Tunes	510	2275	22%	101
OneBusAway	2416	10932	22%	297
ownCloud	1838	6132	30%	194
RedPhone	830	4303	19%	160
XMBCremote	3209	14626	22%	275

Figure 3. Degree of Android API dependence of client code

between a client API reference (i.e., API method calls) and its most recent available version. An API method invocation in client code is considered to be *lagging* if a more recent version of the method is available at the time of its commit. We define the *lag time* of outdated API usage as the number of months elapsed between the release of the new version and the commit time of the outdated API usage code. For example, Figure 4 shows `setbutton2(charSequence)` is deprecated between API version 4 and 7. In client code, developers use the deprecated method at a later date, on December 20th 2009. We consider this method reference is *lagging* because the method was deprecated prior to the client code commit. The **lag time** in this case is approximately two months, the time difference between the client code commit on December 20th 2009 and the deprecation in API Version 7 on October 26th 2009.

We measure the number of outdated API calls and their lagging time. This analysis is done on a `git` commit granularity. For each API invocation in each commit, we first identify the used API version by comparing the method signature of the API call in client code with our Android API Data Structure. We then retrieve the most recent API version of that method available at the time of commit. Finally, by comparing the commit date and the release date of its

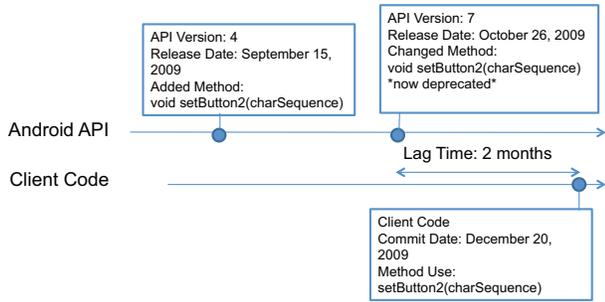


Figure 4. Lag Time Example

updated version, we compute the lagging time. Table IV summarizes the results. At any point in time, on average, 28% of Android method calls are out-of-date and lagging behind the most recent available Android API version. The percentage of outdated API usage varies from a minimum value of 11% to a maximum value of 43% on average.

Table IV  
LAG TIME STATISTICS

Apps	Lag (# Methods)		Min	# Affected Files		
	Max	Avg		Max	Avg	Min
<b>Congress T</b>	516 (50%)	216 (18%)	0 (0%)	81	64	0
<b>Apollo M</b>	968 (72%)	964 (72%)	961(72%)	64	64	64
<b>Cyanogen</b>	256 (17%)	171 (12%)	0 (0%)	35	20	0
<b>Google A</b>	1784 (46%)	1409 (37%)	0 (0%)	134	86	0
<b>LastFM</b>	291 (70%)	181 (43%)	0 (0%)	47	28	0
<b>mp3Tunes</b>	47 (8%)	26 (5%)	4 (1%)	13	8	4
<b>OneBusAway</b>	19 (4%)	14 (3%)	0 (0%)	4	2	0
<b>ownCloud</b>	1488 (52%)	489 (18%)	4 (<1%)	171	121	2
<b>RedPhone</b>	547 (48%)	498 (43%)	414 (35%)	82	72	69
<b>XMBRemote</b>	1421 (41%)	537 (15%)	0 (0%)	238	123	0
<b>Mean</b>	777 (43%)	451 (28%)	138 (11%)	87	60	14

We combine the lagging time results across all subject apps to produce a cumulative distribution of lag time in Figure 5. 50% of all outdated API references are lagging behind the most recent available API by 16 or more months.

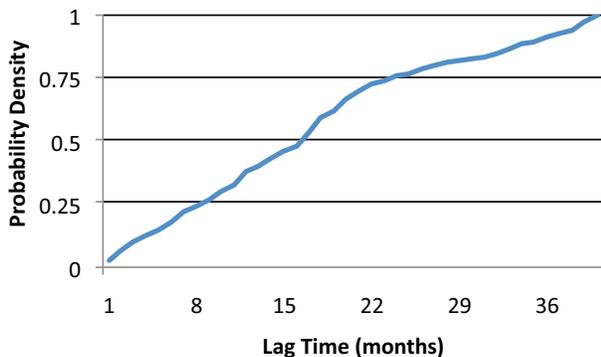


Figure 5. Cumulative Distribution of Lag Time (CDF)

The results suggest that developers do not quickly adopt new APIs. They keep the outdated API references, avoiding the instability of newer APIs and the work that comes with API upgrade.

**RQ5. How long does it take for API changes to propagate throughout the Android ecosystem?** We measure how long it takes for clients to adopt new API usages once a new or updated API becomes available. When a method is eventually updated to a newer API version in client code, we measure its *propagation time*—time difference in months between the API release and the client adaptation timing when the updated usage is committed in the client repository. Figure 6 illustrates this concept by comparing the parallel evolution of Android and a client project. In the Android development time line, the signature of `getMethod` is altered in API version 9 on December 6th 2010 to include an additional `Class` parameter. Client code committed on March 8th 2011 changes the usage of `getMethod` to match the updated API version 9 signature. In this example, the *propagation time* is three months, the time difference between the updated API usage on March 8th 2011 and the release of API version 9 on December 6th 2010.

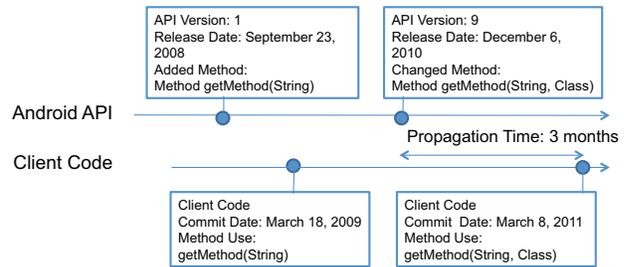


Figure 6. Propagation Time Example

Figure 7 shows the results of propagation time analysis. We inspect each commit patch of the client projects to look for method calls updated to new APIs. For each updated method, we record the propagation time in months. Figure 7 represents the distribution of propagation times across all subject applications in the form of a cumulative distribution plot. The mean propagation time is 14 months (or almost five Android releases) and 50% of all API usage updates occur within approximately 14 months of the associated API release. Outdated API usages eventually upgrade to use newer APIs, but at a much slower pace than the rate of API evolution.

*C. Interplay between Android API evolution and client adoption.*

This section investigates the relationship between API stability, usage, adoption, and bugs.

**RQ6. What is the relationship between API stability and adoption?** To understand how API stability affects

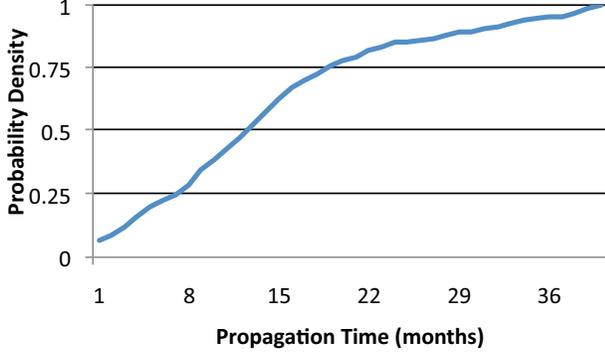


Figure 7. Propagation Time of Methods in Client Code (CDF)

adoption, we measure the Spearman rank correlation between the API evolution rate and adoption measures. For each taxon in Table III, we use the average API update interval (Column Avg Update Interval in Table III), the percentage of total API usages that taxon accounts for (API usage), and the number of API references that were upgraded to newer API versions in client code (propagation count). Table VI summarizes the results. The Spearman correlation between API update interval and API usage is  $-0.47$  with a p-value of  $0.01757$  (See Table VI). A negative correlation value indicates that fast evolving APIs are used more by clients and this trend is statistically significant. The left graph of Figure 8 represents the average API update interval and the API usage percentage for each taxon.

The Spearman correlation between the average API update interval and the propagation count is  $-0.707$  with a p-value  $0.0001113$ . A negative correlation suggests that clients upgrade to faster evolving APIs more frequently. The right side of Figure 8 shows the average API update interval and the number of API usage propagations. These results indicate that faster evolving APIs are used and adopted more by clients.

**RQ7. What is the relationship between API usage and adoption?** To understand the relationship between API usage and adoption, we measure the Spearman correlation between API usage and the average propagation time per taxon (see Table V). When computing the correlation, we remove all taxa whose propagation count is zero. The correlation is  $0.6966$  with p-value  $2.72E-03$ , indicating that APIs that are used more often have higher propagation times. In conjunction with RQ6’s results, this implies that the pace of client updates is slower for widely used, faster evolving APIs and that developers avoid frequent updates to unstable APIs.

We found a positive correlation of  $0.844$  between API usage and propagation count with p-value  $1.93E-05$  (see also Figure 9 for the graph on API usage % and propagation count). The more an API is used, the higher its number of

Table VII  
SPEARMAN RANK CORRELATION BETWEEN BUG FIXES AND API UPDATES

Client Application		Correlation with bugs	p-value
Congress T	Total CLOC	0.39	1.33E-13
	API Update CLOC	0.56	2.20E-16
	Non API Update CLOC	0.39	1.96E-13
OneBusAway	Total CLOC	0.26	1.06E-07
	API Update CLOC	0.46	2.20E-16
	Non API Update CLOC	0.25	2.28E-07
RedPhone	Total CLOC	0.23	1.44E-03
	API Update CLOC	0.24	1.13E-03
	Non API Update CLOC	0.23	1.48E-03
XMBCCremote	Total CLOC	0.34	2.20E-16
	API Update CLOC	0.62	2.20E-16
	Non API Update CLOC	0.33	2.20E-16
Google Analytic	Total CLOC	0.36	1.92E-11
	API Update CLOC	0.54	2.20E-16
	Non API Update CLOC	0.31	5.83E-09
OwnCloud	Total CLOC	0.43	6.113E-16
	API Update CLOC	0.55	2.2E-16
	Non API Update CLOC	0.42	2.81E-15
Cyanogen	Total CLOC	0.58	8.53E-08
	API Update CLOC	0.63	3.749E-09
	Non API Update CLOC	0.58	1.035E-07
LastFM	Total CLOC	0.42	1.10E-07
	API Update CLOC	0.37	5.18E-06
	Non API Update CLOC	0.43	1.04E-07

client code updates. In other words, highly used taxa are adopted more frequently.

**RQ8. What is the relationship between API updates and bugs in client code?** To investigate how API updates affect the likelihood of defects in client code, we analyze the correlation between the number of bugs and the amount of lines changed for the purpose of upgrading to newer APIs. By analyzing version history, we identify java files changed in each commit. For those files, we measure the number of added lines, the number of changed lines for the purpose of upgrading to newer APIs, and the number of changed lines not related to API updates. Next, using a heuristic similar to Mockus and Votta [19], we identify the number of bug fixes by searching for commit messages with the keywords: bug, error, fix, and solve.

We then calculate the Spearman correlation between bug fix CLOC and API upgrade CLOC in client code at the file granularity [30]. We also measure how bugs are correlated with total CLOC and non-API upgrade CLOC respectively. Table VII shows the results. The correlation between bug fixes and API updates is stronger than the correlation between bug fixes and non-API updates in all applications except LastFM.

These results show that, in general, the files with more API updates are more prone to bugs. The stronger correlation

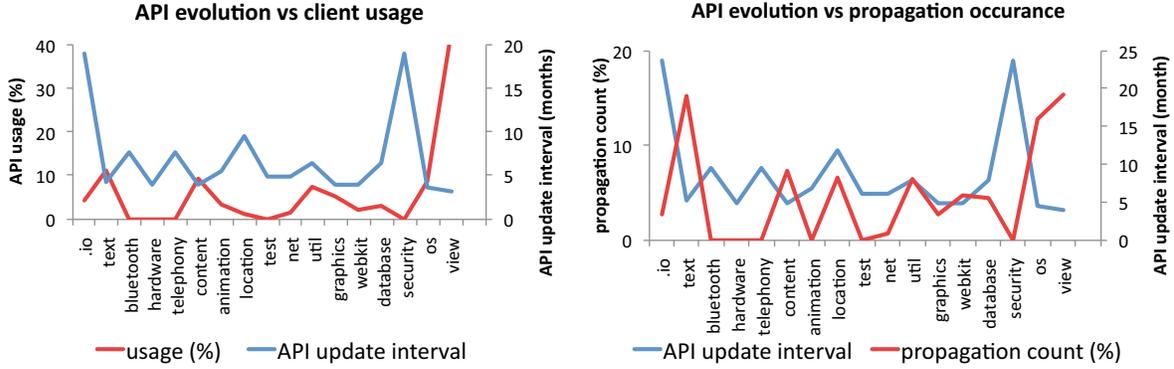


Figure 8. Taxa API update vs. client adoptions

Table V  
CORRELATION BETWEEN API PROPAGATION AND API USAGE

		correlation	p-value
# API usage (%)	propagation time	0.6966134	2.72E-03
# API usage (%)	propagation count	0.8441176	1.93E-05

Table VI  
CORRELATION BETWEEN API EVOLUTION AND CLIENT ADOPTION

		correlation	p-value
avg API update interval	API usage (%)	-0.4706808	0.01757
avg API update interval	propagation count	-0.7072448	0.0001113

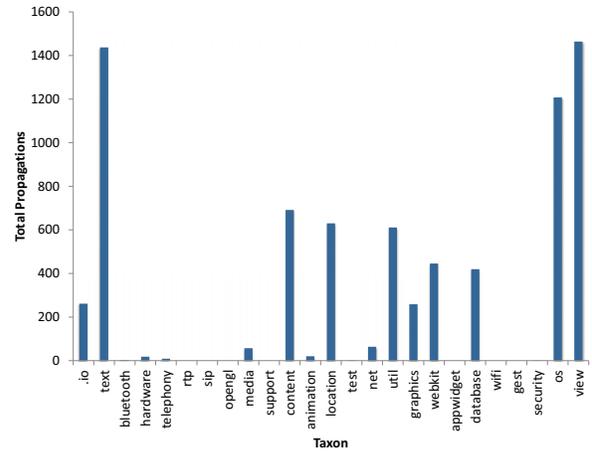
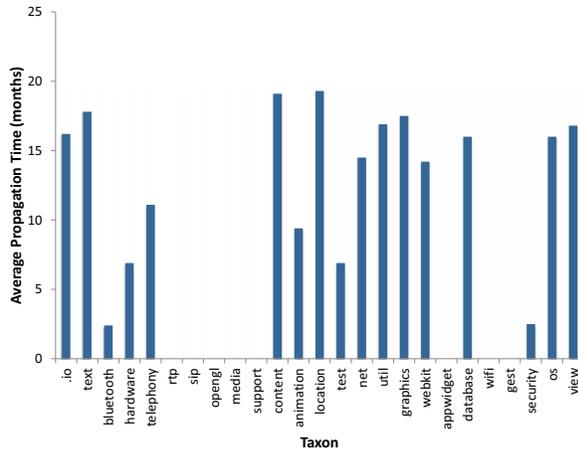


Figure 9. Average Propagation Time and Total Propagation Count for Each Taxon

between API update and bug-fix may explain the slower adoption of new APIs—developers may be skeptic about API adoption as it may introduce bugs. In fact, many developers notice that the API implementation on the Android OS side is often buggy when released. For example, several major bugs related to random rebooting and excessive battery drainage were reported regarding Android 4.2 (Jelly Bean) release [1]. Because of these bugs, developers were hesitant to adopt new APIs or frustrated with their attempts at adoption, some claiming that the release was *the most buggy update since Honeycomb* and they were *definitely expecting an update pretty soon* [1].

## V. THREATS TO VALIDITY

Regarding threats to *construct validity*, we use a syntax-based lexical search to identify Android API references in client code, and we match an API reference with a corresponding API declaration based on the API method name and the number of arguments without considering argument types. For example, consider an API method declaration is updated from `void foo(char, int)` to `void foo(char, char)` on a library side. If nothing is altered at a client call site except the type of the second argument, we cannot detect the change.

Because our method detects API usage change in client code by keeping track of the used API method name, the number of arguments, and argument names, our method could accidentally detect an API usage update when a method invocation is changed from `foo(varA, varB)` to `foo(varA, varC)` even though `varC` is a simply re-named variable of `varB`. While calculating the propagation time and lag time of API references, our method considers API method invocations only, not changes to how API fields were read or used.

Additionally, it is possible for an application to support many different API versions simultaneously [2]. Developers can use version specific APIs inside *if* and *switch* blocks. At runtime, using a Android Build class the API version of a device can be retrieved and appropriate conditional blocks can be executed. Our method of detecting and logging lagging methods does not take into account such multi-version API support.

In terms of threats to *internal validity*, our study presents the correlation between API usage, adoption, and bugs, but not causation. Furthermore, regarding outdated API usages, it is possible that clients are purposely leaving outdated API usages in the codebase to account for the distribution of Android user base. In fact, we find that the most number of propagations are about upgrading to API version 10 (Gingerbread), which currently has the highest market share. The average propagation time to versions up to Gingerbread is higher than that of later versions. This may suggest that developers may eventually gravitate to the versions with a large number of users. Similarly, a high correlation between API updates and bug fixes may be caused by factors beyond our study scope such as test coverage, expertise, etc.

Regarding *external validity*, we investigate Android and ten open source mobile applications found in github, and the results may not generalize to a broader set of mobile applications. Because we chose projects with high activity statistics from different application categories, we believe that our results provide valuable insights on the co-evolution behavior of API and dependent applications.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we perform an empirical study on the co-evolution of Android OS and its clients. Android is evolving fast with an average of 115 API updates per month. Although client applications are heavily dependent on Android, developers seem hesitant to embrace unstable, fast-evolving APIs quickly: 28% of Android references in client code are out-of-date with a median lag time of 16 months. Approximately 22% of these outdated references are eventually adapted to use newer APIs, but the average propagation time is 14 months, or almost 5 Android API version updates.

Furthermore, we study correlations between API usage, API evolution rate, the time taken for API adoption, and the

number of bugs in client code. The APIs that clients use most are the ones Google update most frequently. These same APIs have the higher number of propagations, but with the greater hesitancy (i.e., longer propagation time). Connecting these results with our finding on defect-proneness of API usage adaptation, we believe that developers are hesitant to quickly adopt new, unstable APIs, but eventually tend to migrate their code to keep up with the mass of the Android user base.

To the best of our knowledge, we are the first to find that API updates are more defect prone than other types of changes in client code. Fast-evolving APIs are used more, but the time taken for API adoption is longer. This slow adoption trend may pose various types of risks for client applications such as security vulnerability or poor performance. According to the American Civil Liberties Union (ACLU), “the lag in software updates leaves smartphone users with out-of-date and dangerous systems” [3]. ACLU filed complaints on such spotty Android updates, stating they could potentially harm users by letting hackers steal user data by utilizing security holes. As a part of future work, we would like to understand how the speed of API adoption affects software reliability.

Various stakeholders affect the process of API adoption in the software ecosystem, and further studies are needed to identify factors affecting API adoption. We believe that our findings are a crucial first step and inform future studies on how to promote API adoption and ultimately facilitate the growth of software ecosystems.

## ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants CCF-1149391 and CCF-1117902.

## REFERENCES

- [1] <http://androidheadlines.com/2012/11/2/android/>, 2012. [Online; accessed 23-June-2013].
- [2] <http://stackoverflow.com/questions/3779/>, 2012. [Online; accessed 23-June-2013].
- [3] Aclu: Android fragmentation creates privacy risk. <http://appleinsider.com/articles/13/04/20/aclu-android-fragmentation-creates-privacy-risk>, 2012. [Online; accessed 24-April-2013].
- [4] Android platform versions. <http://developer.android.com/about/dashboards/index.html>, 2012. [Online; accessed 8-April-2013].
- [5] International Data Corporation Worldwide Quarterly Mobile Phone Tracker. [http://www.idc.com/tracker/showproductinfo.jsp?prod\\_id=37.UWMCM5OR98E](http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.UWMCM5OR98E), 2012. [Online; accessed 26-March-2013].
- [6] The many faces of a little green robot. <http://opensignal.com/reports/fragmentation.php>, 2012. [Online; accessed 8-April-2013].

- [7] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263–, Sept. 2001.
- [8] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.
- [9] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *FSE '12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012. ACM.
- [10] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.
- [11] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.
- [12] D. Dig, S. Negara, V. Mohindra, and R. Johnson. Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the 30th International Conference on Software Engineering*, pages 441–450, 2008.
- [13] T. Ekman and U. Askund. Refactoring-aware versioning in eclipse. *Electron. Notes Theor. Comput. Sci.*, 107:57–69, Dec. 2004.
- [14] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.
- [15] D. Hou and X. Yao. Exploring the intent behind api evolution: A case study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 131–140, Washington, DC, USA, 2011. IEEE Computer Society.
- [16] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of refactorings during software evolution. In *ICSE '11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.
- [17] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.
- [18] M. Lungu. Towards reverse engineering software ecosystems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 428–431, 2008.
- [19] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.
- [20] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.
- [21] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, New York, NY, USA, 2012. ACM.
- [22] I. Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122, 2012.
- [23] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. Bringas. On the automatic categorisation of android applications. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 149–153, 2012.
- [24] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333, 2010.
- [25] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:55–64, 2011.
- [26] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, New York, NY, USA, 2006. ACM.
- [27] B. Womack. Google Says 700,000 Applications Available for Android. <http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices>, 2012. [Online; accessed 1-April-2013].
- [28] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.
- [29] S. Yau, J. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 60–65, 1978.
- [30] J. H. Zar. Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.

# How We Design Interfaces, and How To Assess It

Hani Abdeen\*, Houari Sahraoui† and Osama Shata\*

\* Department of Computer Science Engineering, Qatar University, Qatar

hani.abdeen@qu.edu.qa – sosama@qu.edu.qa

† DIRO, Université de Montréal, Montréal(QC), Canada

sahraoui@iro.umontreal.ca

**Abstract**—Interfaces are widely used in Java applications as central design elements for modular programming to increase program reusability and to ease maintainability of software systems. Despite the importance of interfaces and a considerable research effort that has investigated code quality and concrete classes’ design, few works have investigated interfaces’ design. In this paper, we empirically study interfaces’ design and its impact on the design quality of implementing classes (i.e., class cohesion) analyzing twelve Java object-oriented applications. In this study we propose the “Interface-Implementations Model” that we use to adapt class cohesion metrics to assess the cohesion of interfaces based on their implementations. Moreover, we use other metrics that evaluate the conformance of interfaces to the well-known design principles “Program to an Interface, not an implementation” and “Interface Segregation Principle”. The results show that software developers abide well by the interface design principles cited above, but they neglect the cohesion property. The results also show that such design practices of interfaces lead to a degraded cohesion of implementing classes, where these latter would be characterized by a worse cohesion than other classes.

**Keywords**-Object-Oriented; Interfaces; Interface Design Principles; Program to an Interface Principle; Interface Segregation Principle; Cohesion; Empirical Study

## I. INTRODUCTION

Nowadays, interfaces are frequently used as central design elements in object-oriented applications [1], [2], [3]. Interfaces are meant to govern interactions between semi-independent software classes and to encode shared similarities between different class-types [4], [1]. Good design and use of interfaces can significantly ease program comprehension and software maintainability by fostering modularization and minimizing the impact caused by changes in the software implementation [5], [1], [3]. However, designing good interfaces is a sensitive task with a large influence on the rest of the system [5], [2].

Most existing research efforts largely investigated the design quality and detection of design anomalies at the class level without focusing on the specifics of interfaces [6], [7], [8]. This paper aims to address this gap as it constitutes a potential barrier towards fully realizing the benefits of using interfaces as a central element for modular design.

There are two important principles for interface design, which are: “Program to an interface, not an implementation” principle [5, GoF], and we refer to it by PTIP (“Program to Interface Principle”), and the “Interface Segregation Principle (ISP) [2]. The rationale behind the PTIP is to increase the reusability and flexibility of programs by reducing dependencies upon implementations. The rationale behind the ISP is to avoid ‘fat’ interfaces serving diverse clients, so that clients should not depend upon interface’s methods that they do not actually use.

From another perspective, an interface implies the definition and implementation of all its declared methods in their

implementing classes. Naturally, those implementing classes would have different semantics and responsibilities, and may be implementing, at the same time, a variety of interfaces [9], [5], [10]. Still, those classes should be cohesive as much as possible [6], [11], [12], [13]. Studying empirically the design quality of interfaces, assessing their compliance to well-known interface design principles, and/or examining the relations between the design quality of interfaces and that of their implementing classes remain important research challenges.

Recent studies showed that interfaces which do not respect the ISP (as measured by the Service Interface Usage Cohesion (SIUC) metric [14]) are more likely to be change-prone interfaces than those respecting ISP [3]. Yet, we still do not know if those change-prone interfaces do not abide the ISP because they adhere to another design principle, or just because they are badly designed as ‘fat’ interfaces. In other words, it is important to determine if ISP, PTIP, and (interface/implementing class) cohesion are conflicting design properties, or if they can be achieved jointly. Finally, despite many claims about those cited design principles, to the best of the authors’ knowledge, there is no reported study about *how do software developers design and use interfaces?*: e.g., do really developers try to abide by those principles?

We believe addressing those questions can give important support to assist maintainers and designers in their decisions about interface design.

## Contributions

To understand how programmers design and use interfaces, we conduct an empirical study, covering 12 Java real-world applications, that aims at assessing interface design with regard to different properties, namely PTIP, ISP and Cohesion. To assess the compliance of interfaces to the ISP and PTIP principles, we reuse the Service Interface Usage Cohesion (SIUC) metric [14] for measuring ISP, and we develop a new metric, namely Loose Program to Interface (LPTI) for measuring PTIP. The impact of interface design on the cohesion of implementing classes is studied based on the “Interface-Implementations Model” (IIM). This model allows us to adapt existing class cohesion metrics for assessing the internal cohesiveness of interfaces based on their implementing classes. The adapted metrics are the Tight (and Loose) Class Cohesion metrics TCC (and LCC) [11], [12], and the Sensitive Class Cohesion Metric (SCOM) [15], [13]. Using the above-mentioned metrics and 12 open-source Java applications, we conduct statistical analyses to address the following research questions:

**Q1 Do software developers abide by the design properties PTIP, ISP and Cohesion?** we examine empirically whether interfaces in real-world applications abide by the design principles, PTIP, ISP and Cohesion.

**Q2 Is there any conflict between the interface design properties?** we examine the correlations between the ISP and PTIP, and also between those design principles and the internal cohesiveness of interfaces.

**Q3 Do interfaces threaten implementations' cohesion?** we examine the correlation between the cohesion at the interface level and that at the level of implementing classes. Then we compare between the cohesion of classes implementing interfaces and the cohesion of classes which do not implement interfaces.

The remainder of the paper is organized as follows. Section II presents the background and vocabularies we use in this paper. Section III discusses relevant existing work related to interface design. We detail our metrics and the empirical study, with the research hypotheses, respectively in Section IV and Section V. Section VI presents and analyses the results of the empirical study, and describes the findings. Finally, we present threats to their validity in Section VII before concluding.

## II. BACKGROUND AND VOCABULARIES

Before detailing our study for analyzing design quality attributes of interfaces, we introduce the vocabularies and notations we use in the rest of this paper.

**Interface:** we consider an interface as a set of method declarations (i.e., a set of signatures). In this paper, we do not take into account “abstract classes”, “marker interfaces”, interfaces declaring only constants, nor interfaces used only by unit tests (“moke interfaces”). We define the size of an interface  $i$ ,  $size(i)$ , by the number of methods that  $i$  declares.

**Interface Implementations:** a class  $c$  is defined as an implementing class to an interface  $i$  if it defines some *not-abstract* (concrete) method(s) overriding some method(s) of the  $i$ 's methods. For example, if a not-abstract class  $c$  implements explicitly an interface  $i_1$  (i.e., class  $c$  implements  $i_1$ ), and  $i_1$  inherits from another interface  $i_2$  (i.e., interface  $i_1$  extends  $i_2$ ), then  $c$  is an implementing class of both interfaces  $i_1$  and  $i_2$ —since  $c$  must override and implement all the methods declared in  $i_1$  and  $i_2$ . If another class  $c\_sub$  extends  $c$  and defines not-abstract methods overriding some methods of  $i_1$  and  $i_2$ , but not necessarily all their methods, then  $c\_sub$  is also an implementing class of  $i_1$  and  $i_2$ —even though  $c\_sub$  does not explicitly implement  $i_1$  and  $i_2$ .

However, a special subset of the implementing classes of an interface is the *Direct-implementations*, which are the classes that *explicitly* implement the considered interface. In the example above, the class  $c$  is a direct implementation of  $i_1$ , but it is not a direct implementation of  $i_2$ .

We use  $Imp\_C(i)$  to denote the set of all implementing classes of an interface  $i$ . We also use  $Imp\_M(i)$  to denote the set of all the concrete methods that override  $i$ 's methods, which we denote by  $M(i)$ . We denote the subset of direct implementing classes of  $i$  by  $DImp\_C(i)$ .

**Program to Interface Principle (PTIP):** the idea behind the PTIP, as mentioned by GoF [5], is to reduce coupling upon implementations and to achieve flexibility in swapping used objects with different implementations. To this end, interfaces (or abstract base classes) should be used as reference types instead of their implementing classes whenever appropriate [1]. Venners [1] explains in detail and by example how programming to interfaces can add more extensibility and flexibility to OO programs than programming to abstract classes. Nevertheless, Gamma outlines the fact: “As always

there is a trade-off, an interface gives you freedom with regard to the base class, an abstract class gives you the freedom to add new methods later. ...”<sup>1</sup> However, it is worth noting that this paper is limited to the investigation of interface design, not abstract classes. Whether it is better to use abstract classes rather than interfaces is out of the scope of this paper.

With regard to the PTIP, wherever an interface, or a class,  $x$  is used as a type (e.g., a type for an instance variable) we say that there is a *reference* pointing to  $x$ . We denote the set of all references pointing to an interface, or a class,  $x$  by  $Ref(x)$ .

**Interface Segregation Principle (ISP):** interface changes may break the interface clients [3]. Hence, clients should not be forced to depend upon interfaces' methods that they do not actually use. This observation leads to the ISP principle which states that an interface should be designed ‘as small as possible’, so as it declares only methods that are all used by the interface's clients [2].

With regard to the ISP, we use the same definition of interface clients as [3]: we define a class  $c$  as a client to an interface  $i$  if some method in  $c$  calls *any* of the methods overriding the  $i$ 's methods, and/or calls *directly* any of the interface methods—using the polymorphism mechanism. In this definition, we do not care which concrete method will be invoked at runtime, whilst we know that the invoked method will be surely an implementation method of the concerned interface. Hence, the class stating the invocation is a client to that interface.

## III. RELATED WORK

Existing research efforts that are related to object-oriented software design quality mainly revolve around code quality, code smells, and refactoring support for classes (implementations). Few recent articles attempt to address the particularities of interface design.

### A. Source Code Metrics

In literature, there is a large list of source code metrics aiming at assessing the design of OO programs, particularly at class level [6], [16], [11], [12]. The most known ones are the object-oriented metrics by Chidamber and Kemerer (CK) [6]. Chidamber and Kemerer proposed the first class cohesion metric, Lack of Class Cohesion Metric (LCOM), for measuring the lack of class cohesion. This metric has been widely reinterpreted and revised: LCOM2 [6], LCOM3 [17], LCOM4 [18] and LCOM5 (or LCOM\*) [16]. Studies report two problems about those metrics [12], [13], [15]: (1) all of LCOM1 ... LCOM4 do not have an upper limit, hence it is hard to interpret their computed values; (2) all those metrics evaluate the lack of class cohesion instead of measuring the class cohesion itself.

To evaluate existing metrics for class cohesion, on the one hand, Etzkorn et al [12] compares various OO class cohesion metrics (including the LCOMs) with ratings of experts to determine which of these metrics best match human-oriented views of cohesion. The study concludes that the LCC metric (Loose Class Cohesion), as defined by Bieman *et al.* [11], is the metric which best matches the expert opinion of class cohesion. The study also shows that LCC has a strong correlation with the Tight Class Cohesion (TCC), also proposed by Bieman *et al.* in [11]. On the other hand, recently, Al-Dallal

<sup>1</sup>A conversation between Bill Venners and Erich Gamma [5] (June 6, 2005): [www.artima.com/lejava/articles/designprinciples.html](http://www.artima.com/lejava/articles/designprinciples.html)

[13] performed a study investigating the discriminative power of 17 class cohesion metrics, including the LCOMs, LCC and TCC metrics. The discriminative power of a cohesion metric is defined as the probability to produce distinct cohesion values for classes with the same number of attributes and methods but different connectivity pattern of cohesive interactions (CPCI) [13]. This is important since a highly discriminating cohesion metric exhibits a lower chance of incorrectly considering classes to be cohesively equal when they have different CPCIs. The study results show that the Sensitive Class Cohesion Metric (SCOM), as defined by Fernández and Peña in [15], has the best discriminative power.

Although those cohesion metrics are valuable, they unfortunately do not address the particularities of interfaces [3] –since interfaces do not contain any logic, such as method implementations, invocations, or attributes. To the best of our knowledge, there is no reported study mapping those metrics for evaluating the cohesion at the interface level, and/or investigating the relation between the cohesion of implementing classes and interfaces’ design.

### B. Refactoring Support

In recent years, there has been considerable interest in automatic detection and correction of design defects in object oriented software [19], [20], [21]. Mens and Tourwé [8] survey shows that existing approaches are mainly based on source code metrics and predefined bad smells in source code, except for a very few contributions such as [22].

Fowler and Beck [7] propose a set of bad smells in OO class design: e.g., Data class, Blob class, Spaghetti code. Based on Fowler and Beck’s definitions of class smells, several approaches to automatically improving code quality have been developed. Abbes *et al.* [23] performed an empirical study on the impact of Blob and Spaghetti antipatterns on program comprehension. The results conclude that the combination of those antipatterns have a significant negative impact on system comprehensibility. Liu *et al.* [19] provide a deep analysis of the relationships among different kinds of bad smells and their influence on resolution sequences.

Unfortunately, none of those code smells and OO metrics are applicable to Software interfaces. In the literature, few recent articles attempt to address design defects of interfaces.

The authors in [24] study, through 9 open-source OO applications, the correlation between the presence of method clones in different interfaces and the presence of code clones in implementation methods. The results show that there is always a positive correlation, even though moderate, between the presence of method clones in interfaces and the presence of code clones in implementation methods.

Recently, Mihancea and Marinescu [9] studied several recurrent patterns of using inheritance and polymorphism that can impact program comprehensibility (comprehension pitfalls). One of their comprehension pitfalls is the “Partial Typing”, which outlines that interfaces cannot be always assumed as complete types of all their implementing classes. Precisely, it outlines that the variability in the semantics and behaviors of implementing classes of an interface can be very strong in some situations. However, despite the importance of this work for detecting and avoiding the comprehension pitfalls, this work does not investigate interface design and/or its impact on the internal quality of classes (i.e., cohesion).

### C. Interface Metrics

Boxall and Araban define a set of counter metrics to measure the complexity and usage of interfaces [25]. Their metrics return the number of interface methods, all arguments of the interface methods, the interface client classes, etc. The authors in [26] define more complex metrics that assess the interface design quality with regard to existing similarities among interfaces, and with regard to the redundancy in interface sub-class hierarchies.

As outlined in the introduction and Section II, with regard to the ISP, Romano and Pinzger [3] used the Service Interface Usage Cohesion (SIUC), as defined by Perepletchikov [14], to measure the violation of the ISP. Romano and Pinzger refer to SIUC by Interface Usage Cohesion metric (IUC). Their work concludes that in order to limit the impact of interface changes and facilitate software maintenance, interfaces should respect the ISP: i.e., interfaces should be characterized by high values of SIUC metric.

However, despite the success of this cited body of research efforts on interface design metrics, to the best of the authors’ knowledge, up to date, there is no reported study about the compliance of interface design in real-world OO applications to the ISP (and/or PTIP and Cohesion).

### D. Interface Design

Romano and Pinzger [3] investigated the suitability of existing source code metrics (CK metrics, interface complexity and usage metrics, and the SIUC metric) to classify interfaces into change-prone and not change-prone. The paper concluded that most of the CK metrics are not sound for interfaces. Therefore this confirms the claim that interfaces need to be treated separately. The SIUC metric exhibits the strongest correlation with the number of interface changes. Hence, they conclude that the SIUC metric can improve the performance of prediction models for classifying interfaces into change-prone and not change-prone.

The authors in [27] report a technique and tool support to automatically detect patterns of poor API usage in software projects by identifying client code needlessly re-implementing the behavior of an API method. Their approach uses code similarity detection techniques to detect cases of API method imitations then suggest fixes to remove code duplication.

The work in [28] examined the relative impact of interface complexity (e.g. interface size and operation argument complexity) on the failure proneness of the implementation using data from two large-scale systems. This work provides empirical evidence that the increased complexity of interfaces is associated with the increased failure proneness of the implementation (e.g., likelihood of source code files being associated with a defect) and higher maintenance time.

This present paper goes further by studying whether software developers abide by the design principles of interfaces and/or the cohesion quality of implementing classes. Moreover, we empirically study the correlations among those different design properties of interfaces, and study the impact of interface design on the cohesion of implementing classes.

## IV. ASSESSING INTERFACE DESIGN

We are interested in assessing interface design from two perspectives, conformance to design principles and cohesion. To this end, we define, reuse, and adapt a set of metrics to capture these properties.

### A. Metrics For Assessing Interface Design Quality

This section describes the interface metrics that we use to assess interface design with regard to the interface design principles PTIP and ISP.

Similarly to the work by Romano and Pinzger in [3] (see Section III-C), we also use the Service Interface Usage Cohesion (SIUC) to measure the compliance of an interface to the ISP. SIUC is defined by Pereplechikov [14] as follows:

$$SIUC(i) = \frac{\sum_c \frac{num\_used\_mtds(i,c)}{size(i)}}{|Clients(i)|} \quad \forall c \in Clients(i) \quad (1)$$

Where  $num\_used\_mtds(i,c)$  is the number of  $i$ 's methods that are used by the client class  $c$ ; and  $Clients(i)$  is the set of all client classes to  $i$ . SIUC states that an interface has a strong cohesion if each of its client classes actually uses all the methods declared in the interface. SIUC takes its value in the interval [0..1]. The larger the value of SIUC, the better the interface usage cohesion is.

As for the PTIP, to the best of our knowledge, up to date, there is no reported study proposing a measurement or a tool for assessing the adherence of interface design to the PTIP. Following the description of the PTIP in Section II, the PTIP can be interpreted as follows: instead of using a specific concrete class as a reference type, it would achieve more flexibility to use the interface declaring the methods required to perform the desired service [5], [1], [4].

Thus, to assess the design property PTIP, we need to check the dependencies pointing to specific implementing classes rather than to their interfaces. Let  $Ref(i)$  denotes the set of all references pointing to the interface  $i$ , and let  $Ref\_DImp(i)$  denotes the set of all references pointing to the 'direct' implementations of  $i$  (see Section II). We define the Loose Program To Interface metric (LPTI)<sup>2</sup> as the normalized ratio  $Ref(i)$  by  $Ref\_DImp(i)$ :

$$LPTI(i) = \frac{|Ref(i)|}{|Ref(i) \cup Ref\_DImp(i)|} \quad (2)$$

LPTI takes its values in [0..1], where the larger the LPTI value, the better is the interface adherence to the PTIP.

### B. Selecting Cohesion Metrics

As mentioned earlier, cohesion is one of the properties we consider when assessing the interface design. Rather than defining new interface cohesion metrics, we decided to select class cohesion ones and adapt them. This eases the study of the relationship between interface and class cohesion. We set the following criteria for selecting the cohesion metrics for our study: (1) selected cohesion metrics should correlate very well with the human-oriented view of class cohesion [12]; (2) they also should have a good discriminative power [13]; (3) finally, to easily interpret metrics values, it is preferable to select metrics that measure class cohesion instead of lack of cohesion, and take their values in a specific range (e.g., [0..1]).

Following the discussion presented in Section III-A about class cohesion metrics, none of the LCOMs metrics satisfies our 3<sup>rd</sup> criterion. According to the study by Etzkorn *et al.* [12], the LCC metric best matches the human-oriented view of class cohesion. LCC metric, as defined by Bieman and Kang [11], measures class cohesion and takes its values in

<sup>2</sup>The name #“Loose” Program To Interface# outlines that this metric does not consider references pointing to indirect implementations (Section II).

Table I  
DESCRIPTIONS FOR SELECTED CLASS COHESION METRICS.

<b>DEF1.</b> Attributes	The attributes of a class $c$ , $A(c)$ , are all attributes that are defined in $c$ in addition to the not-private attributes defined in the super classes of $c$ (i.e., inherited ones).
<b>DEF2.</b> Local Invocation	We say that there is a local invocation going from a method $m_1$ to another one $m_2$ if in the $m_1$ 's body there is a call to $m_2$ via <i>this/self</i> or <i>super</i> keywords (i.e., an invocation within the scope of the same object).
<b>DEF3.</b> Attribute Access	By definition, a method $m$ accesses an attribute $a$ of its class, if $a$ appears in the $m$ 's body, or within the body of another method that is invoked by $m$ , directly or transitively via local invocations.
<b>DEF4.</b> Methods Connectivity	Two methods belonging to the same class $c$ are defined as “directly” connected if they access at least one shared attribute of $c$ . In the same vein, $m_1$ and $m_n$ , are said “indirectly” connected if there is a sequence of methods $m_2, m_3, \dots, m_{n-1}$ , such that: $m_1 \delta m_2, \dots, m_{n-1} \delta m_n$ ; where $m_x \delta m_y$ represents a direct connection.
Tight Class Cohesion (TCC)*[11]	TCC( $c$ ) = the ratio of directly connected pairs of visible methods within $c$ , $DCpM(c)$ , by the number of all possible pairs of visible methods in $c$ : $NP(c)$ . $TCC(c) = \frac{ DCpM(c) }{NP(c)} \quad (3)$
Loose Class Cohesion (LCC)*[11]	LCC( $c$ ) = the ratio of directly or indirectly connected pairs of visible methods within $c$ , $CpM(c)$ , by $NP(c)$ . $LCC(c) = \frac{ CpM(c) }{NP(c)} \quad (4)$
Sensitive Class Cohesion Metric (SCOM)*[15]	SCOM( $c$ ) = the normalized ratio of the summation of the weighted connection intensity between all pairs of methods in $c$ . The weighted connection intensity between two methods $x$ and $y$ is defined as follows: $Con_{x,y} = a_{x,y} \times ( A(x) \cap A(y)  / \min( A(x) ,  A(y) ))$ . Where $a_{x,y} =  A(x) \cup A(y)  /  A(c) $ , and $A(x)$ and $A(y)$ are respectively the sets of $c$ 's attributes that are accessed by $x$ and $y$ , and $A(c)$ is the set of all $c$ 's attributes. $SCOM(c) = \frac{\sum_q \sum_p Con_{q,p}}{NP(c)} \quad \forall m_q, m_p \in M(c) \quad (5)$

\* All these metrics are applicable only if  $|M(c)| > 1$ . They take their values in [0..1], where 0 is the worst value and 1 is the ideal one.

[0..1]. Thus, LCC satisfies our 1<sup>st</sup> and 3<sup>rd</sup> criterion. However, according to the study by Al-Dallal [13], the metric which has the best discrimination power is the SCOM metric as defined in [15]. SCOM metric measures class cohesion and takes its values in [0..1]. Thus, SCOM satisfies our 2<sup>nd</sup> and 3<sup>rd</sup> criteria. Nevertheless, on the one hand, there is no reported study about the correlation between SCOM and human-oriented view about class cohesion. On the other hand, the study of Al-Dallal shows that LCC does not have a good discriminative power; but the TCC metric, which is also defined by Bieman and Kang [11] and has a strong correlation with LCC [12], has a better discriminative power. According to all this above, we select all of SCOM, LCC and TCC metrics for assessing class cohesion in our experiments. Table I describes these selected metrics.

### C. Adapting Cohesion Metrics

Recall that it is not possible to evaluate interface cohesion without additional heuristics –since interfaces do not contain any logic such as attributes and/or method implementations. To evaluate interface cohesion, our approach retrieves information at the interface implementations to capture coupling between the interface methods, and assess the cohesion at the interface level. Therefore, we propose and use the Interface-Implementations Model (IIM).

In the IIM of a software project  $p$ ,  $IIM(p)$ , every Interface  $i$  is mapped to an entity namely *Integral-Interface*, that we refer to by  $f_i$ . An  $f_i$  consists of a list of methods,  $M(f_i)$ , and a list of attributes,  $A(f_i)$ , so that  $f_i$  can be thought as a class with methods, attributes and accesses. Each method in  $f_i$ ,  $m_{f_i}$ , represents its corresponding method in  $i$ ,  $m_i$ , with all its associated implementation methods  $Imp\_M(m_i)$ . Thus, a  $m_{f_i}$  can be thought as a composition of all the implementation methods of  $m_i$ . Hence, we say that a  $m_{f_i}$  accesses an attribute  $a$  if any of the implementation methods of  $m_i$  accesses  $a$ . Using  $A(m_x)$  to denote the set of attributes accessed by  $m_x$ , the set of attributes accessed by a method  $m_{f_i}$ , and the set of all attributes that are associated to  $f_i$ ,  $A(f_i)$ , are defined as follows:  $A(m_{f_i}) = \cup A(m) \quad \forall m \in Imp\_M(m_i)$ ; and  $A(f_i) = \cup A(m_{f_i}) \quad \forall m_{f_i} \in M(f_i)$ .

Figure 1 shows an example describing the mapping of an interface ( $I1$ ) to its integral-interface presentation ( $fI1$ ). Note that some methods, such as  $C3.bar()$ , access some attributes because of local invocations:  $C3.bar()$  accesses, in addition to  $a3$ , the attribute  $a1$  because it locally invokes  $C1.bar()$  (via `super`), and this latter accesses  $a1$  in its body. Similarly, we say that  $C2.foo()$  accesses, in addition to the attribute  $d2$ , the attributes  $a2$  and  $b2$  (see Table I).

To compute cohesion metrics for interfaces, we use the IIM in which each interface is represented by an integral-interface (see Figure 1). For example, the values of TCC and LCC for the interface  $I1$  in Figure 1 are maximal:  $TCC = LCC = 1$  –since  $I1$  consists of only one pair of methods ( $bar()$ ,  $foo()$ ),  $NP(I1) = 1$ , and these methods are connected:  $A(bar()) \cap A(foo()) = \{b1, a2, b2\}$ . The SCOM value for  $I1$  is:  $\frac{(6/6) \times (3/4)}{1} = \frac{3}{4} = 0.75$  –Equation (5).

**Note.** An integral-interface  $f_i$  does not replicate all the information in its implementing classes –since these latter can completely differ in their semantics and behaviors [9], [5], [10], and also define additional methods to those declared in  $i$ . Taking for example the implementing classes  $C1$  and  $C2$  of the interface  $I1$  (Figure 1). On the one hand, the values of TCC, LCC for  $C2$  are maximal:  $TCC(C2) = LCC(C2) = 1$  –since all the methods in  $C2$  are connected. Similarly,  $SCOM(C2) = \frac{(26/10) \times 3}{13/15} \approx 0.87$ . On the other hand, the values of TCC, LCC and SCOM for  $C1$  are minimal:  $TCC(C1) = LCC(C1) = SCOM(C1) = 0$  –since there is no pair of connected methods in  $C1$ . Recall that the values of TCC, LCC and SCOM for the interface  $I1$  are respectively 1, 1 and 0.75. Hence, we need to empirically test the correlation between the cohesion at interfaces’ level and the cohesion at the level of implementing classes.

For the implementation, we use the Moose toolkit [29] because its meta-model language, FAMIX, includes descriptions of all software entities (e.g., classes, interfaces, methods, attributes etc.) and their associated relationships (e.g., invocations, attributes accesses, references etc.) that are necessary to build the IIMs and compute the metrics. We implemented all the metrics with this platform, utilizing exactly their formulations presented in previous sections.

## V. EMPIRICAL STUDY

In this section, we detail the empirical study that we conduct to answer the research questions outlined in the introduction (Q1, Q2 and Q3).

```

public class C1 implements I1 {
    int a1;
    String b1, x1;

    public void bar() {
        .... a1;
    }

    public void foo() {
        ....
        b1;
    }

    public void something() {
        x1 ...;
    }
}

public interface I1 {
    public void bar();
    public void foo();
}

... C3 extends C1 {
    Double a3, x3;

    public void bar() {
        super.bar();
        a3 ...;
        b1 ...;
    }
}

public class C2 implements I1 {
    Double a2, b2;
    String d2, x2, y2;

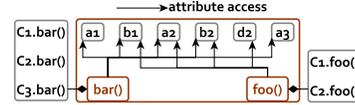
    public void bar() {
        .... a2;
        b2 ...;
    }

    public void foo() {
        .... d2;
        bar();
    }

    public void anotherthing() {
        .... x2;
        y2 ...;
        foo();
    }
}

```

The interface  $I1$ , which declares  $bar()$  and  $foo()$ , is implemented by 3 classes  $C1$ ,  $C2$  and  $C3$ .  $C3$  extends  $C1$  and re-overrides only  $bar()$ .



The representation of the interface  $I1$  in an IIM ( $fI1$ ).  $fI1$  consists of  $bar()$  and  $foo()$ , which access 6 attributes:  $A(fI1) = \{a1, b1, a2, b2, d2, a3\}$ .

Figure 1. Example about IIM.

### A. Goal, Perspective and Target

The goal of the study is to analyze, from the perspective of researchers, the design of interfaces in order to: - evaluate the adherence of real-world interfaces to the design properties PTIP, ISP and Cohesion; - investigate the correlation between those different design properties; - and finally, evaluate the impact of interface design on the cohesion of implementing classes. We believe the results of our study are interesting for researchers and also for software maintainers and engineers who want to fully release the benefits of interfaces for modular design of OO systems.

### B. Context

The context of the empirical study consists of twelve open-source object-oriented software applications that are all written in Java. Table II shows information about the size of those applications: (1) ArgoUML<sub>v0.28.1</sub>, (2) Hibernate<sub>v4.1.4</sub>, (3) Jboss<sub>v6.0.0</sub>, (4) Jfreechart<sub>v1.0.14</sub>, (5) JHotDraw<sub>v7.1</sub>, (6) Opentaps<sub>v1.5.0</sub>, (7) Plantuml<sub>v7935</sub>, (8) Proguard<sub>v4.8</sub>, (9) Rapidminer<sub>v5.0</sub>, (10) Spring-RCP<sub>v1.1.0</sub>, (11) SweetHome<sub>v3.4</sub>, and (12) Vuze<sub>v4700</sub>. We selected open-source systems so that researchers can replicate our experiments. However, many of those systems are widely used in both academic and industry (e.g., Hibernate, Jboss, Rapidminer and Spring-RCP). We carefully selected those systems to cover a large variety of systems that differ in utility, size, users and developers community. Finally, each of those systems contains a considerable variety of interfaces: small, medium and large interfaces – except Proguard and SweetHome which contain small sets of relatively small interfaces.

For each application, we collected the values of SIUC and LPTI for each interface, and the values of TCC, LCC and SCOM for each interface and class. Using the Anderson-Darling normality test, we found that the values collected for all the metrics are not normally distributed, with  $p$ -values  $< 0.05$ . This means that parametric tests cannot be used in our study.

### C. Hypotheses

The empirical study aims at assessing the following hypotheses with regard to interface design.

Table II  
INFORMATION ABOUT CASE-STUDY SOFTWARE PROJECTS.

App	<sup>1</sup> Classes	<sup>2</sup> Interfaces	$\sum size(i)$	Imp	DImp
ArgoUML	1793	88	833	358	177
Hibernate	2658	319	2352	745	563
Jboss	3573	458	3350	648	523
Jfreechart	374	45	231	104	83
JHotDraw	396	42	523	159	58
Opentaps	1257	113	1386	214	167
Plantuml	722	77	311	328	205
Proguard	388	18	164	238	240
Rapidminer	1796	99	628	780	228
Spring-RCP	566	95	558	159	100
SweetHome	355	13	131	30	28
Vuze	2539	683	5724	1493	1270
<b>TOTAL</b>	<b>16417</b>	<b>2050</b>	<b>16191</b>	<b>5256</b>	<b>3642</b>

$\sum size(i)$  represents the number of all methods declared in interfaces; *Imp* and *DImp* denote respectively implementing classes and direct implementation ones; Only classes and interfaces defining/declaring more than 1 visible method are considered.

<sup>1</sup> Concrete (not-abstract) classes, without unit test classes. <sup>2</sup> Excluding interfaces outside the system boundary (e.g., Java library interfaces, such as Serializable interface).

**H<sub>0PTIP</sub>** There is no significant evidence showing that interfaces' design respects the PTIP. The alternative hypothesis is:  
**H<sub>1PTIP</sub>** There is a statistical evidence showing that interfaces adhere to the PTIP.

Identically to the above hypotheses, we define the other hypotheses with regard to the other design properties SIP and Cohesion: **H<sub>0SIP</sub>** and **H<sub>1SIP</sub>**; and **H<sub>0Coh</sub>** and **H<sub>1Coh</sub>**. We assess each of the previous null hypotheses by comparing the distance between the average value of the corresponding metric(s) (e.g., SIUC for ISP) to the ideal and worst values of that (those) metric(s). Recall that all used metrics take their values in [0..1], where 0 is the worst value and 1 is the ideal one. For this purpose, we divide the interval [0..1] as follows: **[0 BAD values 0.4 INDECISIVE values 0.6 GOOD values 1]**.

**H<sub>0ρ(P TIP,ISP,Coh)</sub>** There is no significant correlation between the interface properties PTIP, ISP and Cohesion. The alternative hypothesis is:  
**H<sub>1ρ(P TIP,ISP,Coh)</sub>** There is a significant *positive* (or *negative*) correlation between the interface properties. Thus, designing interfaces with regard to one of those properties can anticipate enhancing (or degrading) the quality of interface design with regard to the other correlated property/ies.

We assess the previous null hypothesis for each couple of interface properties (i.e., **H<sub>0ρ(P TIP,ISP)</sub>**, **H<sub>0ρ(P TIP,Coh)</sub>** and **H<sub>0ρ(ISP,Coh)</sub>**) by studying the Spearman correlations between the values of corresponding metrics.

**H<sub>0Coh(Int,Imp)</sub>** There is no significant evidence showing that interfaces' design can impact the cohesion of implementing classes. The alternative hypothesis is:  
**H<sub>1Coh(Int,Imp)</sub>** Interfaces impact the cohesion of classes.

To assess the **H<sub>0Coh(Int,Imp)</sub>**, first, we study the Spearman correlation between the cohesion of interfaces and that of their implementing classes; second, we compare the cohesion of implementing classes and not-implementing classes using a two-tailed Wilcoxon test.

## VI. RESULTS ANALYSIS

This section presents and analyses the results of our study aiming at assessing the research hypotheses.

### A. Interface Design with-respect-to ISP, PTIP and Cohesion

Figure 2 shows that interfaces have a strong tendency to be designed according to the PTIP and ISP properties, but also with neglecting the cohesion property. On the one hand, Figure 2(a) shows that for all studied applications, bar only two applications (JFreeChart and Proguard), the median values of LPTI metric are very high and close (or even equal) to the ideal value 1. The figure shows also that for all studied applications, bar only two applications (JHotDraw and Sweet-Home), the median values of SIUC metric are very high. On the other hand, the Figure 2(a) shows that the median values of cohesion metrics (TCC, LCC and SCOM) are very close to the worst value in 7 applications (ArgoUML, Hibernate, JBoss, Rapidminer, Spring-RCP, SweetHome and Vuze). Only in 3 applications (JFreeChart, Opentaps and Proguard), interfaces tend to have acceptable cohesion values. These results are evidenced in Figure 2(b), where all interfaces are studied together, and in Figures 2(c), 2(d) and 2(e), where random samples of interfaces are studied.

Table III shows, on the first hand, that the mean values of LPTI and SIUC metrics are decisively high ( $Mean > 0.6$ ), but the mean values of cohesion metrics (TCC, LCC and SCOM) are not decisive in many cases ( $0.4 \leq Mean \leq 0.6$ ). Except for the cases of ArgoUML, Hibernate, JBoss, Rapidminer, SweetHome, and Vuze, mean cohesion values are decisively small:  $Mean < 0.4$ . This is also true when considering all the interfaces together. On the second hand, Table III shows that the standard deviations of metric values are high in almost all cases, thus we can't draw our conclusions by looking only to the mean values. Still, considering the size of our sample (2050 interfaces from 12 applications), and the mean and median values, we are confident that interfaces generally conform to design principles PTIP and ISP, but are not sufficiently cohesive.

This leads us to the following findings:

**F1** Software developers definitely abide by the "Program to Interface" and "Interface Segregation" principles.

**F2** Developers of interfaces do not abide the "Cohesion" property in interfaces' design.

These findings lead us to conjecture that PTIP and ISP, on the one hand, and the Cohesion, on the other hand, could be conflicting interface-design properties.

### B. Correlations between ISP, PTIP and Cohesion Properties

To assess the **H<sub>0ρ(P TIP,ISP)</sub>**, **H<sub>0ρ(P TIP,Coh)</sub>** and **H<sub>0ρ(ISP,Coh)</sub>** hypotheses, we study the correlations between the involved properties. The results are presented in Table IV. Although the correlations between the interface properties PTIP, ISP, and Cohesion are statistically significant when considering all the applications together, they are not strong enough ( $-0.2 \leq \rho \leq 0.11$ ) to be practically significant. More specifically, correlations between PTIP and ISP are, for all the applications, close to zero except for Rapidminer ( $-0.45$ ). Besides, there is a weak (to very weak) negative correlation between the PTIP and Cohesion properties in many cases,

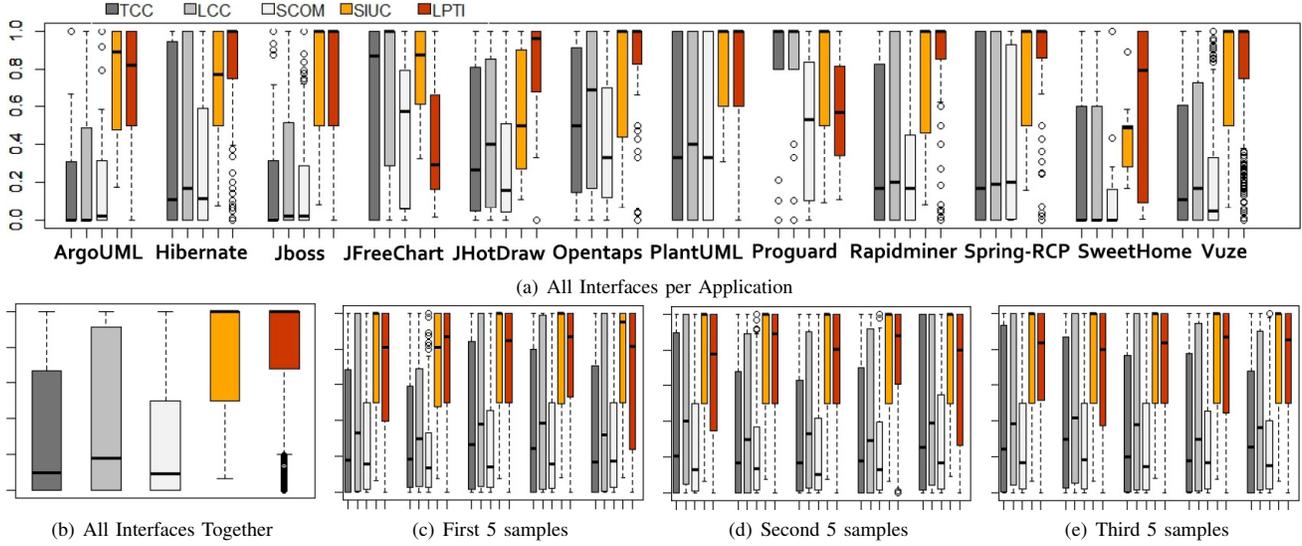


Figure 2. Box plots of #TCC, #LCC, #SCOM, #SIUC and #LPTI metrics for interfaces.

Each of 2(c), 2(d) and 2(e), shows box blots of interface metrics for 5 random samples of 100 interfaces taken from all applications and with possible overlaps.

Table III

THE MEAN VALUES AND STANDARD DEVIATIONS ( $\sigma$ ) FOR INTERFACE METRICS, BY APPLICATION, AND FOR ALL INTERFACES TOGETHER.

	PTIP		ISP		Cohesion					
	LPTI	$\sigma$	SIUC	$\sigma$	TCC	$\sigma$	LCC	$\sigma$	SCOM	$\sigma$
ArgoUML	<b>0.70</b> ↑	0.31	<b>0.74</b> ↑	0.29	<i>0.20</i> ↓	0.34	<i>0.26</i> ↓	0.39	<i>0.25</i> ↓	0.38
Hibernate	<b>0.83</b> ↑	0.29	<b>0.71</b> ↑	0.30	<i>0.36</i> ↓	0.42	0.40	0.44	<i>0.30</i> ↓	0.38
Jboss	<b>0.73</b> ↑	0.41	<b>0.76</b> ↑	0.30	<i>0.20</i> ↓	0.33	<i>0.28</i> ↓	0.37	<i>0.21</i> ↓	0.33
Jfreechart	0.41	0.33	<b>0.79</b> ↑	0.23	0.58	0.45	<b>0.65</b> ↑	0.42	0.47	0.39
JHotDraw	<b>0.82</b> ↑	0.25	0.56	0.33	0.41	0.38	0.46	0.39	<i>0.31</i> ↓	0.35
Opentaps	<b>0.85</b> ↑	0.27	<b>0.74</b> ↑	0.33	0.50	0.38	0.57	0.40	0.42	0.35
PlantUML	<b>0.76</b> ↑	0.32	<b>0.83</b> ↑	0.22	0.47	0.44	0.51	0.44	0.56	0.42
Proguard	0.59	0.30	<b>0.77</b> ↑	0.33	<b>0.78</b> ↑	0.39	<b>0.81</b> ↑	0.34	0.50	0.37
Rapidminer	<b>0.84</b> ↑	0.30	<b>0.74</b> ↑	0.33	<i>0.36</i> ↓	0.41	0.40	0.43	<i>0.29</i> ↓	<i>0.34</i>
Spring-RCP	<b>0.85</b> ↑	0.29	<b>0.78</b> ↑	0.28	0.40	0.45	0.44	0.46	0.40	0.41
SweetHome	0.60	0.44	0.44	0.22	<i>0.27</i> ↓	0.39	<i>0.28</i> ↓	0.39	<i>0.16</i> ↓	0.29
Vuze	<b>0.84</b> ↑	0.24	<b>0.76</b> ↑	0.30	<i>0.32</i> ↓	0.38	<i>0.36</i> ↓	0.40	<i>0.24</i> ↓	0.34
ALL	<b>0.80</b> ↑	0.31	<b>0.75</b> ↑	0.30	<i>0.33</i> ↓	0.40	<i>0.38</i> ↓	0.41	<i>0.28</i> ↓	0.36

High Means ( $> 0.6$ ) and low Means ( $< 0.4$ ) are respectively in boldface (annotated with ↑) and italic-face (annotated with ↓).

Table IV  
SPEARMAN CORRELATION AMONG INTERFACE PROPERTIES ( $\alpha = 0.05$ ).

	PTIP vs. ISP		PTIP(LPTI) vs. Coh.			ISP(SIUC) vs. Coh.		
	LPTI,SIUC	-TCC	-LCC	-SCOM	-TCC	-LCC	-SCOM	
ArgoUML	0.09	0.05	0.07	0.11	-0.30	-0.34	-0.26	
Hibernate	-0.19	-0.06	-0.10	-0.02	0.04	0.07	0.17	
Jboss	-0.01	-0.22**	-0.25**	-0.22**	0.19	0.16	0.21*	
Jfreechart	-0.06	-0.30*	-0.20	-0.16	-0.04	-0.02	0.03	
JHotDraw	-0.27	-0.05	-0.07	0.00	0.23	0.09	0.33	
Opentaps	0.11	-0.30**	-0.35**	-0.24*	-0.02	0.02	0.17	
PlantUML	0.00	0.02	0.00	0.08	0.07	-0.02	-0.06	
Proguard	-0.27	<b>-0.50</b> *	-0.47*	-0.34	<b>0.76</b> **	<b>0.76</b> **	0.55	
Rapidminer	-0.45**	-0.25*	-0.33**	-0.19	0.16	0.20	0.14	
Spring-RCP	-0.09	-0.25*	-0.30**	-0.22**	0.07	0.05	0.20	
SweetHome	-0.02	<b>-0.57</b> *	<b>-0.59</b> *	-0.59**	-0.14	-0.14	-0.11	
Vuze	-0.09	-0.23**	-0.26**	-0.19**	0.07	0.10	0.07	
ALL	-0.13**	-0.17**	-0.20**	-0.13**	0.07*	0.08*	0.11**	

Significant results obtained with  $p - value < 0.01$  or  $0.01 \leq p - value \leq 0.05$  are respectively annotated with \*\* or \*. Significant correlations ( $\rho \geq 0.5$  or  $\rho \leq -0.5$ , and  $p - value \leq 0.05$ ) in bold face.

except in SweetHome. In this unique case, the negative correlation is strong enough ( $\approx 0.6$ ). Finally, we observed a strong positive correlation between ISP and Cohesion (TCC/LCC) only in one application (Proguard). In both cases of strong correlations, the concerned applications have the smallest sets of interfaces, 13 for SweetHome and 18 for Proguard. As a consequence, and with regard to our findings F1 and F2 explained in the previous section, the low cohesion of software interfaces cannot be explained by conflicting forces with the other design properties of interfaces. These results lead us to the following finding:

**F3** There is no evidence showing that the PTIP, ISP and Cohesion are conflicting properties of interface design, so that these properties can be achieved jointly. However, developers of interfaces abide by the PTIP and ISP properties, but neglect the Cohesion property.

This finding brings us to our final question, whether interfaces impact the cohesion of implementing classes.

Table V  
SPEARMAN CORRELATION BETWEEN THE COHESION OF INTERFACES ( $i$ ) AND THE COHESION OF THEIR IMPLEMENTING CLASSES,  $imp$ , AND DIRECT ONES,  $dimp$  ( $\alpha = 0.05$ ).

	Coh( $i$ )vs.Coh( $imp$ )			Coh( $i$ )vs.Coh( $dimp$ )		
	TCC	LCC	SCOM	TCC	LCC	SCOM
ArgoUML	<b>0.60**</b>	<b>0.61**</b>	<b>0.56**</b>	<b>0.60**</b>	<b>0.61**</b>	0.48**
Hibernate	0.44**	0.45**	0.40**	<b>0.55**</b>	<b>0.56**</b>	<b>0.58**</b>
Jboss	<b>0.60**</b>	<b>0.62**</b>	<b>0.59**</b>	<b>0.71**</b>	<b>0.74**</b>	<b>0.69**</b>
Jfreechart	0.30**	0.35**	0.13	<b>0.51**</b>	<b>0.52**</b>	0.33**
JHotDraw	0.26**	0.24**	0.32**	<b>0.57**</b>	<b>0.60**</b>	<b>0.53**</b>
Opentaps	0.14*	0.17**	0.22**	0.19*	0.22**	0.24**
PlantUML	0.44**	0.42**	0.37**	0.44**	0.40**	0.44**
Proguard	0.04	0.03	0.19**	0.02	0.01	0.17**
Rapidminer	0.17**	0.20**	0.14**	0.32**	0.38**	0.37**
Spring-RCP	0.36**	0.30**	0.38**	<b>0.58**</b>	<b>0.52**</b>	<b>0.57**</b>
SweetHome	<b>0.63**</b>	<b>0.75**</b>	<b>0.52**</b>	<b>0.68**</b>	<b>0.76**</b>	<b>0.57**</b>
Vuze	0.46**	0.49**	0.47**	<b>0.52**</b>	<b>0.55**</b>	<b>0.52**</b>
ALL	0.44**	0.45**	0.44**	<b>0.50**</b>	<b>0.51**</b>	<b>0.51**</b>

Significant results obtained with  $p - value < 0.01$  or  $0.01 \leq p - value \leq 0.05$  are respectively annotated with \*\* or \*. Significant correlations ( $\rho \geq 0.5$  or  $\rho \leq -0.5$  and  $p - value \leq 0.05$ ) are in boldface.

### C. Interface Design vs. Implementations Cohesion

Up to now, we found that interfaces are in general characterized by a low cohesion that is weakly correlated with the other design properties PTIP and ISP. In this section, we study whether this fact impacts on the cohesion of implementing classes, or not, by assessing the null hypothesis  $H_{0Coh(int,imp)}$  presented in Section V-C.

To assess this hypothesis, we first study the correlation between the cohesion of interfaces and the cohesion of implementing classes. We perform this study with regard to all implementing classes, and for all ‘Direct’ implementing classes (see Section II). Then, we perform a Wilcoxon test comparing the cohesion of implementing classes to the cohesion of the other *concrete* (not-implementing) classes.

**Note.** Recall that an interface  $i$  can have several implementing classes. Thus, to study the correlation between interface cohesion and implementation cohesion, with high precision, we consider for each interface  $i$  all the pairs of cohesion values with regard to  $i$ ’s implementations:  $(Coh(i), Coh(imp_1))$ ,  $(Coh(i), Coh(imp_2))$  ...  $(Coh(i), Coh(imp_n))$ , where ‘Coh’ refers to a cohesion metric (TCC, LCC or SCOM), and where an  $imp_j$  denotes an implementing class to  $i$ . In other words, we do not simply use the average/median cohesion value of implementations to perform the correlation analysis.

The results in Table V show the following observations. The correlation between the cohesion of interfaces and the cohesion of all their implementing classes (first three columns) is in general statistically significant, but generally below 0.5. When considering all the applications, the correlation is average ( $\approx 0.45$ ). However, by considering only direct implementing classes (last three columns), the correlation becomes stronger in almost all cases. This shows that there is considerable association between interfaces’ design and the cohesion of their implementing classes, particularly those classes which directly implement interfaces. This is true despite the variability in the semantics and behaviors of those classes, their number and size, and their freedom to implement different interfaces at the same time and/or to belong to different class-type hierarchies (recall our discussion in Section IV-C).

From another perspective, Table VI shows the difference between cohesion mean values for implementing classes and not-implementing classes. At first glance, the delta values are not that big, and we cannot assume that using interfaces always impacts (degrades) the cohesion of classes. Actually, in many cases, the mean cohesion values of classes implementing interfaces is somewhat larger than the cohesion of those which do not implement interfaces. Notable exceptions are ArgoUML, Hibernate, JBoss, PlantUML, Rapidminer, and Vuze. In these cases, the results show that the cohesion of implementing classes is worse than the cohesion of the other classes. Surprisingly, these cases match exactly the cases in which interfaces are characterized with low cohesion: see Table III and our discussion in Section VI-A. This confirms that our design of interfaces impacts the cohesion of classes which implement them. This is also supported by the results of the Wilcoxon test comparing the cohesion of all implementing classes against that of all not-implementing classes. The results show a statistically significant degradation of the cohesion of implementing classes compared the cohesion of not-implementing ones. In addition to support our finding about interface cohesion F2, these results allows us to reject the null hypothesis  $H_{0Coh(int,imp)}$  and accept the alternative one  $H_{1Coh(int,imp)}$ :

**F4** Interfaces negatively impact the cohesion of classes.

However, considering that the deltas between cohesion mean values are large enough only for applications with low mean cohesion for interfaces, we claim that interfaces impact negatively the cohesion of implementing classes if interfaces do not adhere to the Cohesion property.

### Results Summary

Our study shows that there is empirical evidence that software developers definitely abide by the ‘‘Program to Interface’’ and ‘‘Interface Segregation’’ principles of interface design, but they neglect the ‘‘Cohesion’’ property. However, there is no significant evidence to explain the low cohesion at interfaces by an intrinsic conflict with the well-known principles of interface design (aforementioned). This empirical finding is of high importance since the results show that there is considerable correlation between the cohesion of interfaces and the cohesion of their implementing classes, particularly, direct implementation ones. Note that this result was obtained while considering all varieties of interfaces, including those declaring just a couple of methods; and regardless of the size and number of implementing classes, or other factors that can cause a strong variability in the semantics and behaviors of (variability in the cohesion of) implementing classes. Finally, and most important, there is empirical evidence that such design practices of interfaces lead to a degraded cohesion of implementing classes, where the cohesion of these latter (as measured by the TCC, LCC and SCOM metrics) is worse than the cohesion of not-implementing classes.

### Results Implications

We believe our findings are interesting for both researchers and software developers (and maintainers). Although the ‘‘Program to Interface’’ and ‘‘Interface Segregation’’ principles (PTIP and ISP) of interface design are widely known, this is, to the best of our knowledge, the first time in software engineering research that the results show strong evidence on

Table VI  
COMPARISON BETWEEN THE COHESION OF IMPLEMENTING CLASSES (IMP: 5256 CLASSES IN TOTAL) AND THAT OF NOT-IMPLEMENTING CLASSES (NOT\_IMP: 11161 CLASSES IN TOTAL), WITH TWO-TAILED WILCOXON TEST ( $\alpha = 0.05$ ).

	TCC			LCC			SCOM		
	Imp	NOT_Imp	$\Delta$	Imp	NOT_Imp	$\Delta$	Imp	NOT_Imp	$\Delta$
ArgoUML	0.19	0.32	<b>-0.13</b>	0.21	0.34	<b>-0.13</b>	0.13	0.29	<b>-0.16</b>
Hibernate	0.36	0.46	<b>-0.10</b>	0.37	0.48	<b>-0.11</b>	0.30	0.45	<b>-0.15</b>
Jboss	0.30	0.34	<b>-0.04</b>	0.36	0.45	<b>-0.09</b>	0.20	0.27	<b>-0.07</b>
Jfreechart	0.55	0.48	<i>0.07</i>	0.56	0.47	<i>0.09</i>	0.47	0.46	<i>0.01</i>
JHotDraw	0.38	0.44	<b>-0.06</b>	0.42	0.35	<i>0.07</i>	0.23	0.37	<b>-0.14</b>
Opentaps	0.51	0.47	<i>0.04</i>	0.50	0.48	<i>0.02</i>	0.43	0.35	<i>0.08</i>
PlantUML	0.40	0.48	<b>-0.08</b>	0.40	0.45	<b>-0.05</b>	0.42	0.45	<b>-0.03</b>
Proguard	0.33	0.33	0.00	0.33	0.35	<b>-0.02</b>	0.29	0.29	0.00
Rapidminer	0.32	0.36	<b>-0.04</b>	0.32	0.35	<b>-0.03</b>	0.29	0.29	0.00
Spring-RCP	0.45	0.45	0.00	0.47	0.41	<i>0.06</i>	0.37	0.37	0.00
SweetHome	0.42	0.41	<i>0.01</i>	0.53	0.42	<i>0.11</i>	0.18	0.32	<b>-0.14</b>
Vuze	0.31	0.39	<b>-0.08</b>	0.23	0.36	<b>-0.13</b>	0.22	0.33	<b>-0.11</b>
ALL	0.34	0.41	W.diff. <b>-0.07**</b>	0.36	0.41	W.diff. <b>-0.05**</b>	0.27	0.35	W.diff. <b>-0.07**</b>

Delta values in **boldface** (*italic-face*) denote that implementing classes have **worse** (*better*) cohesion than not-implementing classes. For ALL applications, the *W.diff.* denotes the *Difference-in-Location* obtained from a two-tailed Wilcoxon test. \*\* and \* denote respectively results obtained with  $p - value < 0.01$  or  $0.01 \leq p - value \leq 0.05$ .

the adherence of interfaces to these principles in real and large long-living software systems. Hence, this paper provides a strong foundation for the importance of these design principles for further investigation of interface design and OO program quality. The results imply that researchers and software maintainers should consider the particularities of interfaces by using appropriate metrics for assessing interfaces' design (e.g., SIUC metric and our proposed metric LPTI).

Furthermore, the results of our study imply that researchers and software maintainers should consider the impact of interface design on implementing classes while investigating their quality. They should consider the Cohesion property, in addition to other design principles, while designing and/or maintaining interfaces. That is by using appropriate models, such as our proposed model IIM, for mapping class metrics (e.g., cohesion and coupling metrics) for assessing program quality at the interface level. This would assist maintainers in identifying interfaces that cause a poor design quality. Additionally, our findings suggest that researchers in software refactoring field should consider the refactoring of interfaces (along with their implementing classes) to improve programs' design. Finally, due to the important cost that may be caused by changes in interfaces, our findings strongly suggest that researchers should investigate new methodologies (e.g., metrics) that can estimate the internal cohesion of interfaces in early design stages (i.e., before realizing implementing classes).

## VII. THREATS TO VALIDITY

This section considers different threats to validity of our findings.

The threats to internal validity of our study concern the used variables, such as the values of the metrics (LPTI, SIUC, TCC, LCC and SCOM). In our study, the values of all used metrics are computed by a static analysis tool (Moose [29]) and deterministic functions. We chose Moose because its meta-model, which is FAMIX, includes descriptions of all software entities (e.g., interfaces, classes, methods, attributes, etc.) and their associated relationships that are necessary to compute our metrics. In the presence of programming-language dynamic features such as polymorphism and dynamic class loading, the dependencies (e.g., method calls) could be slightly over-

or underestimated by static analysis [30]. In our work, this threat does not concern the computed values of LPTI since this metric is computed using only static type references Section IV-A. Regarding the SIUC values, this threat is mitigated since we well handled the problem of polymorphic methods calls, that is due to late-binding mechanism, as explained in Section II (see "*Interface Segregation Principle*" paragraph). As for the values of used cohesion metrics, this threat is very mitigated since computing these metrics is based only on local attribute accesses and local method calls –i.e., via *this/self* or *super*, see Table I.

Construct validity threats can come from the fact that the used metrics might not actually measure the studied properties. Regarding the class cohesion metrics, we circumvent these threats by carefully selecting various cohesion metrics. Our selections were based on relevant recent studies about class cohesion metrics [13], [12]. We derived our finding about the Cohesion property by considering all of these cohesion metrics. As for the SIUC metric, we referred to a recent published study using also this metric for assessing the alignment of interface design to the design principle ISP [3]. The only metric we defined in this paper is LPTI, since we did not find in literature any metric for assessing the alignment of interface design to the PTIP. Still, in our definition of LPTI, we carefully referred to the interpretations of the PTIP in most relevant studies about interface design principles (e.g., [5], [1], [2], [4]).

Threats to external validity concern the generalization of our findings. One possible threat in our study is that all the used applications are open-source software systems. We believe, however, that our sample is fairly representative because we carefully selected twelve applications that differ in size and utility, among them some systems are widely used in both academic and industry (e.g., Hibernate, Jboss and Rapidminer). We plan, in the near future, to verify our findings through additional industrial projects.

To prevent the possible threats to conclusion validity, we carefully select the statistical methods that are adequate for both the nature of the data and the hypotheses. Because our data is not normally distributed (Anderson-Darling test), we used the Spearman correlation coefficient to investigate the

relation between different interface properties (PTIP, ISP and Cohesion) and also to investigate the relation between interface cohesion and the cohesion of implementing classes. We used the non-parametric statistical hypothesis test, Wilcoxon signed-rank test to compare the cohesion of implementing classes with that of other classes. Finally, we performed many tests, and obtained  $p$ -values were in general very low. After their correction for multi-tests, they remain statistically significant.

## VIII. CONCLUSION AND FUTURE WORK

In this paper, we investigated the design of software interfaces from the perspective of the well-known design principles “Program to an Interface, not an Implementation” (PTIP) and “Interface Segregation” (ISP), and with regard to the Cohesion property. We conducted an empirical study through a large variety of open-source software systems containing 2050 interfaces and many thousands of classes. For our study, we used the SIUC metric and defined the LPTI metric to assess the conformance of interfaces to the PTIP and ISP principles. Furthermore, we used several class cohesion metrics (TCC, LCC and SCOM) for assessing the cohesion of classes, and we adapted them for the interfaces by means of the Interface-Implementations Model (IIM), that we propose in this paper.

The results of our study showed that software developers abide well by the design principles PTIP and ISP, but they consider less the cohesion property. This empirical finding is of high importance since the results also showed that interfaces with low cohesion tend to degrade the cohesion of all classes implementing them, compared to the classes which do not implement interfaces. The results of our study imply that researchers and software maintainers should consider the impact of interface design on programs while investigating their quality. They should also consider the Cohesion property, in addition to other design principles, while designing and/or maintaining interfaces. Our findings suggest that existing monitoring tools of software quality should be extended to adapt the class cohesion metrics for assessing the cohesion of interfaces. This would assist maintainers in identifying interfaces that cause a poor design quality. In addition, tools can provide information about interface use by using the the LPTI metric for assessing the alignment of interfaces to the PTIP.

Our future work will concentrate on investigating further interface properties, such as size, depth of inheritance and number of implementing classes, and the relations between these properties and the properties that we investigated in this paper. Another direction to investigate in the future is the associations between interface design and external quality attributes, such as program reusability and comprehensibility.

## ACKNOWLEDGMENT

This publication was made possible by NPRP grant #09-1205-2-470 from the Qatar National Research Fund (a member of Qatar Foundation).

## REFERENCES

- [1] B. Venners, “Designing with interfaces,” 1998, <http://www.javaworld.com>.
- [2] R. C. Martin, “Design principles and design patterns,” 2000, [www.objectmentor.com](http://www.objectmentor.com).
- [3] D. Romano and M. Pinzger, “Using source code metrics to predict change-prone java interfaces,” in *Proceeding of ICSM’11*, 2011, pp. 303–312.

- [4] N. Warren and P. Bishop, *Java in Practice*. Addison Wesley, 1999.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [6] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. on Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [8] T. Mens and T. Tourwé, “A survey of software refactoring,” *Trans. on Softw. Eng.*, vol. 30, no. 2, pp. 126–138, 2004.
- [9] P. Mihancea and R. Marinescu, “Discovering comprehension pitfalls in class hierarchies,” in *Proceedings of CSMR ’09*, 2009, pp. 7–16.
- [10] B. Liskov, “Data abstraction and hierarchy,” in *Proceeding of the OOPSLA’87*, ser. OOPSLA ’87, 1987, pp. 17–34.
- [11] J. M. Bieman and B.-K. Kang, “Cohesion and reuse in an object-oriented system,” in *Proceedings of the 1995 Symposium on Softw. reusability*, ser. SSR ’95. ACM, 1995, pp. 259–262.
- [12] L. H. Etzkorn, S. Gholston, J. Fortune, C. Stein, D. R. Utley, P. A. Farrington, and G. W. Cox, “A comparison of cohesion metrics for object-oriented systems,” *Inf. & Softw. Tech.*, vol. 46, no. 10, pp. 677–687, 2004.
- [13] J. Al Dallal, “Measuring the discriminative power of object-oriented class cohesion metrics,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 788–804, Nov. 2011.
- [14] M. Perepletchikov, C. Ryan, and K. Frampton, “Cohesion metrics for predicting maintainability of service-oriented software,” in *Proceedings of QJIC ’07*, ser. QJIC ’07. IEEE Computer Society, 2007, pp. 328–335.
- [15] L. Fernández and R. P. na, “A sensitive metric of class cohesion,” *Inf. Theories and Applications*, vol. 13, pp. 82–91, 2006.
- [16] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
- [17] W. Li and S. Henry, “Maintenance metrics for the object oriented paradigm,” in *Proceedings of International Softw. Metrics Symposium*, 1993, pp. 52–60.
- [18] M. Hitz and B. Montazeri, “Measuring coupling and cohesion in object-oriented systems,” in *Proc. Intl. Sym. on Applied Corporate Computing*, 1995.
- [19] H. Liu, Z. Ma, W. Shao, and Z. Niu, “Schedule of bad smell detection and resolution: A new way to save effort,” *IEEE Trans. on Softw. Eng.*, vol. 38, no. 1, pp. 220–235, Jan. 2012.
- [20] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Trans. on Softw. Engineering*, vol. 38, pp. 5–18, 2012.
- [21] A. Oumi, M. Kessentini, H. A. Sahraoui, and M. Boukadoum, “Maintainability defects detection and correction: a multi-objective approach,” *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.
- [22] M. Kessentini, S. Vaucher, and H. A. Sahraoui, “Deviance from perfection is a better criterion than closeness to evil when identifying risky code,” in *25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 113–122.
- [23] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension,” in *Proceedings of CSMR’11*, ser. CSMR ’11. IEEE Computer Society, 2011, pp. 181–190.
- [24] H. Abdeen and O. Shata, “Characterizing and evaluating the impact of software interface clones,” *International Journal of Software Engineering & Applications (IJSEA)*, vol. 4, no. 1, pp. 67–77, Jan 2013.
- [25] M. A. S. Boxall and S. Araban, “Interface metrics for reusability analysis of components,” in *Proceedings of ASWEC ’04*, ser. ASWEC ’04. IEEE Computer Society, 2004, pp. 40–51.
- [26] H. Abdeen and O. Shata, “Metrics for assessing the design of software interfaces,” *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 1, no. 10, pp. 737–745, dec 2012.
- [27] D. Kawrykow and M. Robillard, “Improving api usage through automatic detection of redundant code,” in *Proceedings of ASE’09*. IEEE, 2009, pp. 111–122.
- [28] M. Cataldo, C. de Souza, D. Bentolila, T. Miranda, and S. Nambiar, “The impact of interface complexity on failures: an empirical analysis and implications for tool design,” School of Computer Science, Carnegie Mellon University, Tech. Rep., 2010.
- [29] S. Ducasse, M. Lanza, and S. Tichelaar, “Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems,” in *Proceedings of CoSET ’00 (2nd International Symposium on Constructing Software Engineering Tools)*, Jun. 2000.
- [30] S. Allier, S. Vaucher, B. Dufour, and H. A. Sahraoui, “Deriving coupling metrics from call graphs,” in *10th Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 43–52.

# An Accurate Stack Memory Abstraction and Symbolic Analysis Framework for Executables

Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua  
*University of Maryland*  
*College Park*  
 {kapil,wazeer,akotha,msmithso,barua}@umd.edu

Angelos Keromytis  
*Columbia University*  
*New York*  
 angelos@cs.columbia.edu

**Abstract**—This paper makes two contributions regarding reverse engineering of executables. First, techniques are presented for recovering a precise and correct stack memory model in executables in presence of executable-specific artifacts such as indirect control transfers. Next, the enhanced memory model is employed to define a novel symbolic analysis framework for executables that can perform the same types of program analysis as source-level tools. Frameworks hitherto fail to simultaneously maintain the properties of correct representation and precise memory model and ignore memory-allocated variables while defining symbolic analysis mechanisms. Our methods do not use symbolic, relocation, or debug information, which are usually absent in deployed binaries. We describe our framework, highlighting the novel intellectual contributions of our approach, and demonstrate its efficacy and robustness by applying it to various traditional analyses, including identifying information flow vulnerabilities in five real-world programs.

## I. INTRODUCTION

Reverse engineering executable code has received a lot of attention recently in the research community. The demand for advanced executable-level tools is primarily fueled by a rapid rise in zero-day attacks on several popular applications. It is a well-known fact that most of the applications used on a daily basis are IP-protected software that are available only in the form of executables. Robust reverse-engineering tools are required to completely analyze the impact of latest cyberattacks on such applications, to define efficient counter strategies and to certify their robustness against such attacks.

Reverse engineering tools are also essential for continuous software maintenance. Various organizations such as US Department of Defense [1] have critical applications that have been developed for older systems and need to be ported to future secure versions in light of exposed vulnerabilities. In many cases, the application source code is no longer accessible requiring these applications to continue to run on outdated configurations. This engenders a need for advanced tools which enable identification and extraction of functional components for reuse in new applications.

The applicability of a reverse-engineering framework in the above scenarios of vulnerability detection, software certification and software maintenance entails three desired features: 1) The recovered intermediate representation (IR)

should be *functional* such that it can be employed to recover a functional source code or recompiled to obtain a working rewritten binary. A non-functional code also fails to capture the complete application behavior, resulting in inaccurate analysis results. 2) Since executables mainly contain memory locations instead of explicit program variables, the IR should have a *precise memory abstraction* to effectively reason about memory operations in presence of executable-specific features such as indirect control transfer instructions (CTI) and lack of procedure prototypes. The paucity of registers in x86 ISA further underscores this requirement by allocating most of the variables to memory locations. 3) The framework must support advanced analyses mechanisms on the recovered IR, enabling the *same kind of analysis that can be done on the original source code*. Unfortunately, obtaining all three features in a framework is very challenging when dealing with stripped binaries which do not contain symbolic or debugging information.

Executable specific artifacts such as indirect CTIs complicate the task of recovering a *precise memory abstraction* while maintaining the *functionality* in IR. A memory abstraction involves associating each stack memory reference to a set of variables on the memory stack. In order to recover such an abstraction, we need to determine the value of stack pointer at each program point in a procedure relative to its value at the entry point. This is usually accomplished by analyzing each stack modification instruction, including CTIs which can possibly modify the stack pointer due to several reasons such as cleanup of arguments.

However, the modification in the value of stack pointer cannot be easily determined in all scenarios. For example, in case of an indirect CTI, the stack modification is deterministic only if all its statically determined possible targets modify the stack pointer by the same value. However, such targets might modify the stack pointer by different values, or a call to an external function with an unknown prototype might have a statically indeterminable impact on the value of stack pointer. Existing frameworks [2], [3] require that the return from a CTI should always modify the stack pointer by a deterministic constant value.

We present techniques for recovering a precise memory

model and functional IR in such scenarios. Our mechanism formulates a set of constraints using control flow constructs in the caller procedure to compute the value of stack modification at a call-site. The constraints are solvable in most scenarios. When the constraints cannot be solved, it embeds run-time checks to maintain the functionality of IR.

This enhanced memory model improves the precision of several analysis techniques for executables. In our second contribution, we employ this memory model and present a novel symbolic analysis for executables, *Symbolic Value Analysis*, which enables *analysis similar to source code*. Symbolic analysis [4], [5] is employed for a variety of applications such as alias analysis and security analysis. In source code, only pointer and array accesses are considered memory accesses, hence such symbolic analysis methods [4], [5] only focus on instructions involving program variables. Due to a large percentage of memory operations in executables, a symbolic analysis framework for executables must employ a precise memory model.

However, existing frameworks either ignore memory models while recovering a symbolic abstraction or do not recover a symbolic abstraction. Several executable techniques [6], [7], [8] restrict their analysis to only the registers and handle memory locations in a very conservative manner. Consequently, these methods lose a great deal of precision at each memory access. On the other hand, several popular analysis frameworks for executables, notably Value Set Analysis (VSA) [3] and related methods [9], analyze memory accesses but represent values of program variables as a set of integral values and memory locations, which do not represent the symbolic relations between these variables.

The primary contributions of our work are the following:

- **Precise and correct stack memory abstraction:** We present a hybrid static-dynamic mechanism to determine the impact of executable specific artifacts such as indirect CTIs on the value of stack pointer, resulting in a functional representation and a precise memory model. This mechanism can be employed to improve the precision of any existing memory analysis framework such as VSA [3] and others [9].
- **Novel Symbolic Analysis:** Based on the improved memory model, we formulate a novel Symbolic Value Analysis which computes symbolic abstraction for variables as well as for memory locations.
- **Applications:** We extend our analysis for several applications such as security analysis, redundancy removal and demonstrate that client applications become less effective when memory tracking is not enabled.

We evaluated our techniques with SPEC2006 benchmark suite as well as several real world programs such as Apache server. Our techniques improve the precision of memory models by 25% in programs containing significant number of indirect CTIs. This improved memory model enhances the precision of Symbolic Value Analysis by 20% on average.

```
main:
1  sub 24, $esp      //Local Allocation
2  mov $10, 8(%esp) //Access (%esp+8)
3  call *%eax        // An Indirect call
4  mov $20, 12(%esp) //Access
                        //(%esp+12+UNKNOWN)
.....
```

Figure 1: An example demonstrating the imprecision in the presence of indirect calls, second operand in the instruction is the destination

Our techniques are scalable and analyze large programs such as gcc in less than 5 minutes.

## II. MOTIVATION

In this section, we demonstrate the limitation of existing frameworks in obtaining a functional IR with a precise memory model and the relative importance of considering the underlying memory model for symbolic abstraction.

**Precise and correct stack memory abstraction :** A source program has an abstract stack representation where the local variables are assumed to be present on the stack but their precise layout is not specified. In contrast, an executable has a fixed physical stack layout.

To recreate an IR, the physical stack must be deconstructed to individual abstract frames per procedure. Since, each such frame comprises variables from the source code, a memory model is defined as precise if each frame can be divided into abstract locations analogous to the original variables.

Previous methods [3] have approached this problem in two steps. First, all the instructions in a procedure which can modify the stack pointer are analyzed to compute the maximum size to which the stack can grow in a single invocation of the procedure. Next, each such abstract frame is further abstracted through a set of `a-locs`. An `a-loc` is characterized by two attributes: its relative offset in the region with respect to other `a-locs` and its size. The `a-loc` representation requires the determination of the value of the stack pointer at each program point in a procedure relative to its value at the entry point.

As highlighted in Section I, this is usually accomplished by tracking each update to the stack pointer. However, several artifacts might result in a non-deterministic stack modification, invalidating the inherent assumption in previous frameworks [3]. We characterize the impact of a CTI `I` on the value of stack pointer using the following definition:

$$\text{StackDiff}(I) = \text{Stack Pointer after } I - \text{Stack Pointer before } I.$$

The term `StackDiff` can be applied to either the CTI or a corresponding called procedure, and represents the stack modification amount in either case. `StackDiff` of a CTI can be positive if the called procedure cleans up

its arguments, or zero if it does not. In theory, it can be negative if the procedure leaves some local allocations on the stack, although we have not observed this in compiled code. Several approaches have been suggested to calculate the value of `StackDiff` by symbolically evaluating all the stack modification instructions in a procedure [3]. As per these methods, `StackDiff` at an indirect CTI is deterministic if all possible targets have the same value of `StackDiff`. Thereafter, the stack pointer in the caller procedure is adjusted by `StackDiff` amount. This adjustment is imperative for maintaining the correctness of data-flow.

However, `StackDiff` cannot be determined statically in all scenarios. For example, possible targets of an indirect CTI might have different `StackDiff`, or an external function with an unknown prototype might have a statically unknown `StackDiff`. In such scenarios, existing frameworks either result in an imprecise memory abstraction or fail to maintain the correctness. As per CodeSurfer/X86, “if it cannot determine that the change is a constant, it issues an error report” (Section 4.2) [3]. Hence, the corresponding frame cannot be represented through `a-locs`, resulting in an imprecise memory model. IDAPro applies a constraint-based mechanism to compute the values of `StackDiff` independent of the called procedures. However, when the underlying method fails to determine a unique solution, it compromises the correctness by accepting one feasible solution (which could be wrong) out of an infinite number of possible outcomes [10].

Fig 1 illustrates an example of such a scenario. In Fig 1, a local region of size 24 is allocated in a procedure, consequently, the memory access at Line 2 results in the discovery of an `a-loc` at offset 16. Suppose the possible targets of the indirect CTI at line 3 have different `StackDiff` values. Consequently, `esp` after Line 3 has an unknown offset relative to its value at the entry point of the procedure. Hence, no `a-loc` can be identified at Line 4. On the other hand, if `StackDiff` value is calculated wrongly, it results in an incorrect data-flow at Line 4.

Our hybrid mechanism maintains the precision as well as functionality. Our static mechanism enables abstraction through a set of `a-locs` and dynamic mechanism guarantees the correctness when `StackDiff` cannot be computed.

**Symbolic abstraction:** Since executables extensively employ memory locations, not analyzing them for symbolic analysis in executables results in imprecise symbolic relations. Fig 2(a) shows a source code example and the relations between various computations determined through symbolic analysis. Fig 2(b) shows a sample code which might arise when the example in Fig 2(a) is converted to an executable. Here, variables `a`, `b`, `c` and `d` are allocated to memory locations. *Since existing symbolic analyses for source code [4] as well as for executables [6] do not propagate symbolic expressions across memory locations, a new symbol is defined at every memory reference instruction.*

	Allocations: a: -4(%ebp) b: -8(%ebp) c: -12(%ebp) d: -16(%ebp)	No Memory abstraction	With Memory abstraction
int main(){ int a,b,d; scanf("%d",&a); if(a>0) return; b=a+2; ..... c=a+12; d=b+10; }	main: 1 mov \$esp,\$ebp 2 sub 24,\$esp //Local Allocation 3 lea -4(%ebp),4(%esp) //mov &a for arg 4 mov ptr,(%esp) //mov "%d" for arg 5 call scanf 6 mov -4(%ebp),%eax //Load a 7 jg L1: //Return if a>0 8 add \$2,%eax //Compute a+2 9 mov %eax,-8(%ebp) //Store b 10 ..... 11 mov -4(%ebp),%eax //Load a 12 add \$12,%eax //Compute a+12 13 mov %eax,-12(%ebp) //Store c 14 ..... 15 mov -8(%ebp),%eax //Load b 16 add \$10,%eax //Compute b+10 17 mov %eax,-16(%ebp) //Store d L1: ret	x1 x1+2	x1 x1+2
Symbolic Relations:		x2 x2+12	x1 x1+12
b=a+2 c=a+12 d=b+10		x3 x3+10	x1+2 x1+12

Figure 2: (a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination (c) Symbolic relations on the assembly code with no memory abstraction (d) Symbolic relations on the assembly code with memory abstraction

As evident from Fig 2(c), the resulting symbolic relations are conservative and yield imprecise program information.

We observe that representing a symbolic abstraction for memory locations can eliminate this limitation. Fig 2(d) shows the symbolic relations when a symbolic abstraction is maintained for memory locations as well. Suppose, the variable `a` ( $-4(\%ebp)$ ) has value `x1` in the environment of symbolic abstraction. Hence, the representation of symbolic abstraction for memory locations implies that the variable `%eax` at Line 6 and Line 10 is assigned value `x1`. Similarly, the memory location  $-8(\%ebp)$  at Line 9 and the variable `%eax` at Line 13 are assigned value `x1+2`. Propagation of these values results in symbolic relations that are similar to those obtained through the source code.

### III. OVERVIEW

Fig 3 presents an overview of our binary analysis framework. Our framework is built over existing SecondWrite framework as presented in [11]. SecondWrite translates the input x86 binary code to a functional program represented in the intermediate representation (IR) of the LLVM Compiler [12]. SecondWrite implements various mechanisms to obtain an IR which contains features like procedures, procedure arguments and return values. *This conversion back to a compiler IR is not a necessity for the work we present; any binary system [2], [3] can use our analysis.* LLVM IR obtained above is passed through our analysis system.

A key challenge in binary analysis is discovering which portions of the code section in an input executable are definitely code. SecondWrite implements *speculative disassembly* and *binary characterization*, proposed by Smithson and Barua [13], to efficiently address this problem. The indirect CTIs are handled by translating the original target to the corresponding location in IR through a *call translator*

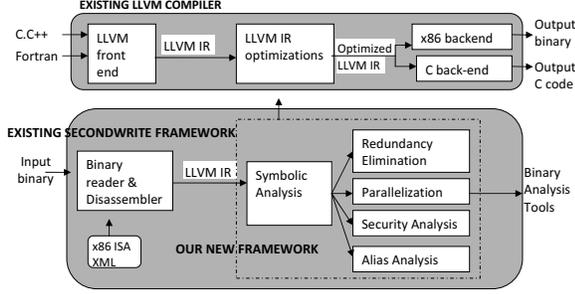


Figure 3: Organization of the system

procedure [13]. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis.

**Memory assumptions:** Similar to most executable analysis frameworks [3], [14], [15], our techniques assume that executables follow the *standard compilation model* where each procedure may allocate an optional stack frame in one direction only and each variable resides at a fixed offset in its corresponding region. Like most static binary tools, we do not handle self modifying or obfuscated code.

#### IV. RECOVERING PRECISE MEMORY MODEL

In this section, we discuss our hybrid static-dynamic solution to obtain a functional representation with a precise memory model. We first present a symbolic constraint mechanism to determine the value of  $StackDiff$  for each CTI where it is unknown. Next, we discuss our solution for maintaining the functionality even when  $StackDiff$  at some CTIs cannot be solved. Our analysis employs the prototypes of well-known library functions, similar to the IDAPro’s FLIRT database [2], for determining their  $StackDiff$  value. We assume that existing methods [3] are able to determine the value of  $StackDiff$  for each procedure, which holds true under the assumptions of *standard compilation model*.

##### A. Static Computation

A CTI  $I$  can result in an unknown  $StackDiff$  in three cases, which we collectively refer to as *Unknown CTIs*.

**Case 1:**  $I$  is a direct CTI to an external procedure with unknown prototype.

**Case 2:**  $I$  is an indirect CTI with unresolved targets.

**Case 3:**  $I$  is an indirect CTI and its targets have different  $StackDiff$ .

In such scenarios, our mechanism employs several boundary conditions imposed by the control flow inside the corresponding caller procedure to determine  $StackDiff$ . The proposed constraint formulation does not require us to determine the precise set of targets of an indirect CTI, which itself is an extremely challenging problem.

We define symbolic values  $X_I$  and  $S_I$  for representing  $StackDiff$  and local stack height at a CTI  $I$ . Every stack

**Unknown Symbolic Values :**  $X_I$ , where  $X_I = StackDiff$  of procedure call  $I$

**Initial/Helper Variables :**

$Targ(T)$ : Set of procedures targeted by call target address  $T$

$StackDiff(f)$ :  $StackDiff$  of procedure  $f$

$Y\_SET(F) = \cup_{f \in F} StackDiff(f)$

$BeginP$  = Entry point of procedure  $P$ ;  $Pred_{BB}$  = Predecessors of basic block  $BB$ ;

$BeginBB, EndBB$  = Entry point, terminator of basic block  $BB$

$S_I$  = Stack height after instruction  $I$ ;

$S_{BB}$  = Stack height at beginning of basic block  $BB$ ;

$PrevI$  = the previous instruction to  $I$  ( $I \neq BeginBB$ )

$S_{I'}$  = if ( $I \neq BeginBB$ ) then  $S_{PrevI}$  else  $S_{BB}$

$R$ : A register,  $Size(R)$ : Size of register  $R$ ,  $N$ : A constant

**Initial Conditions :**  $S_{BeginP} = 0$

**Data flow rules :**

For every instruction  $I$ :

$I = push R \Rightarrow S_I = S_{I'} + size(R)$

$I = pop R \Rightarrow S_I = S_{I'} - size(R)$

$I = add esp, N \Rightarrow S_I = S_{I'} - N$

$I = sub esp, N \Rightarrow S_I = S_{I'} + N$

$I = jmp L \Rightarrow S_{BeginL} = S_{I'}$

$I = call Y \Rightarrow$

if ( $Y\_SET(Targ(Y))$  contains a single constant  $C$ )

$S_I = S_{I'} + C$

else

$S_I = S_{I'} + X_I$

default (if not an invalidation condition)  $\Rightarrow S_I = S_{I'}$

**Boundary Conditions :**

1.  $\forall BB: \forall Pred \in Pred_{BB}, S_{BeginBB} = S_{EndPred}$

2.  $I = ret$ : Constraint  $S_{I'} = 0$

**Invalidation Conditions :**

1.  $I = esp \leftarrow \dots$  /\* Any assignment except in data-flow rules\*/

2.  $I$  accesses return address

Figure 4: Data flow rules used to determine stack modifications in a procedure  $P$

modification instruction in a procedure is analyzed to derive an expression of  $S_I$  in terms of the  $X_{IS}$ . The resulting expressions are transformed into a linear system of equations that can be solved to calculate the value of  $X_{IS}$ .

Fig 4 presents the rules for generating symbolic constraints and equations in a particular procedure  $P$ . It presents rules for analyzing each stack modification instruction, a set of initialization and boundary conditions for solving the symbolic equations and a set of conditions which invalidate our symbolic constraints for the current procedure.

In an x86 program, several instructions can modify the value of stack pointer. The local frame in a procedure is usually allocated by subtracting a constant value from  $esp$ . Similarly, the local frame is deallocated by adding a constant amount to  $esp$ . Push and pop instructions implicitly modify the stack pointer by the size of amount pushed onto the stack. The rules in Fig 4 incorporate the deterministic modification at each CTI. An indeterministic modification is modeled symbolically as  $X_I$ . The dataflow rules in Fig 4 obtain an expression for  $S_I$  considering each such stack modification instruction.

In order to solve the above symbolic equations, Fig 4 generates two constraints based on the control flow in procedure  $P$ . These conditions hold true for every executable following the standard compilation model [3]:

$\rightarrow \forall Pred \in Pred_{BB}, S_{BeginBB} = S_{EndPred}$ : This condition implies that at a merge point in the control flow of a procedure, the stack height at the end of every

predecessor basic block must be equal. Otherwise, any subsequent stack access might access different stack locations depending on the path taken at run time, resulting in an indeterminate behavior.

→  $S_{I'} = 0 \forall \text{ret} \in \mathbb{P}$ : In an x86 program, a return instruction loads an address from the location pointed by `esp` and sets the program counter to the loaded value. Since the return address is pushed by the caller procedure and a compiled program usually does not access the return address directly, `esp` can refer to the return address only if stack height  $S_{I'}$  is zero. Thereafter the return instruction may optionally specify an operand to clean up some incoming arguments, so `StackDiff` could be positive or zero.

Fig 4 also formulates the following conditions which invalidate the assumptions behind our boundary conditions. In such situations, we discontinue our static mechanism and rely on our dynamic mechanism to maintain the correctness.

→  $I = \text{esp} \leftarrow \dots$ : Any assignment to `esp` other than those in data-flow rules implies a local frame allocation of variable size. In such a scenario, the boundary conditions fail to obtain a solution for  $X_I$ . However, this condition arises in extremely rare circumstances of variable size arrays on stack frame.<sup>1</sup>

→  $I$  accesses return address: In a usual compiled code, `StackDiff` is either zero or positive. In theory, procedures could have a negative `StackDiff`, implying that the procedure leaves some local allocations on the stack. In such scenarios, `esp` would not point to the return address at the point of return. Hence, a return must be implemented by explicitly accessing the return address from the middle of the stack. This invalidates the assumption behind our boundary condition 2 and we resort to run-time checks.

The resulting symbolic equations are solved by employing a custom linear solver that categorizes the equations into disjoint groups based on the variables used in every equation. A group is solved only if the number of equations is equal to the number of unknowns. We keep propagating calculated values to other groups until no more calculated values are present. Once we obtain a solution of  $X_I$  for each  $I$  in a procedure, we can obtain a safe abstraction of memory regions into a set of *a-locs* using the methods in [3].

### B. Dynamic Mechanism

As mentioned above, the above method does not guarantee a solution for all the scenarios. For example, it fails to determine the value of `StackDiff` in basic blocks containing multiple CTIs each with an unknown  $X_I$  value. Below, we discuss our dynamic mechanism to handle all the three cases of *Unknown CTIs* presented in Section IV-A.

<sup>1</sup>Code produced by popular compilers contains x86 idioms like `leave` instruction which implicitly assign a previously stored value to `esp`. Such idioms are currently handled explicitly in our framework.

```

Sym := Sym+T|T
T := T*F|F
F := I|n
I := [IR Variables]
n := [Int]

```

Figure 5: Grammar for symbolic expressions.  $+$  and  $*$  are standard arithmetic operators, *Int* is the set of all integers, *IR Variables* are symbols in the obtained intermediate representation

**Case 1:** Since this case represents control transfer to an external procedure, the body of the called procedure cannot be modified. Such scenarios are handled by calling the external procedure using a trampoline. The trampoline dynamically computes the shift in stack pointer value before and after the call using inline assembly instructions.

**Case 2 and Case 3:** Recall from Section III, an indirect CTI is translated to the corresponding location in IR using a switch statement inside a *call translator* procedure. In such scenarios, `StackDiff` is declared as an explicit return variable in the call translator procedure. The definition of the call translator is modified to return the value of `StackDiff` for the called procedure in each switch statement.

## V. SYMBOLIC VALUE ANALYSIS

Our technique, Symbolic Value Analysis, is a flow-sensitive, context insensitive analysis which computes a conservative over-approximation of a set of symbolic values that each data object (variables and `a-locs`) can hold at each program point. Symbolic Value Analysis is based on memory model obtained in Section IV.

Sec V-A first presents the abstraction for representing the symbolic values in our analysis and subsequent sections discuss the intraprocedural and interprocedural versions of the analysis.

### A. Symbolic Abstraction

Fig 5 presents the grammar for representing the symbolic expressions in our abstraction. As evident from Fig 5, symbolic expressions are numeric algebraic polynomials containing sums of product terms of variables.

**Symbolic Value Set:** A symbolic value set is a finite set of symbolic expressions defined by the Grammar in Fig 5. It constitutes a conservative over-approximation of the set of symbolic values that each data object holds.

The abstraction supports standard arithmetic set operators such as Addition ( $\oplus$ ) and Multiplication ( $\otimes$ ). The abstraction also supports a Widen ( $\nabla$ ) operator. This operator implements the inherent widening operation in our environment. If the required cardinality increases beyond a limit, we invalidate the current symbolic value set. This operation prevents the exponential blowup of symbolic expressions.

$$\nabla \text{SymValSet}_1 = \{\text{if } |\text{SymValSet}_1| > \text{LIMIT}, \text{then } \top \text{ else } \text{SymValSet}_1\} \quad (1)$$

## B. Intraprocedural Analysis

Our analysis defines three kind of memory regions, associated with procedures (Stack), global data (Global) and heaps (HeapRgn). The method presented in Section IV enables us to precisely abstract the above memory regions through a set of `a-locs` in the presence of indirect CTIs.

Our method assumes that the symbols corresponding to the binary code’s registers have been converted to single-static assignment (SSA) form before running our analysis. Since in SSA form each variable is assigned exactly once, a single symbolic map is sufficient to maintain flow-sensitive symbolic value sets for variables. However, memory locations are usually not implemented in SSA format in IR. Consequently, a symbolic map is maintained at each program point to represent flow-sensitive symbolic value sets for memory locations. Hence, symbolic value analysis effectively computes the following maps:

SR: Map between `Vars` and their corresponding symbolic value sets.

SM<sub>e</sub>: Map between `a-locs` and their corresponding symbolic value sets before a program point `e`

Executables regularly employ the *indirect-addressing* mode for accessing memory locations. After obtaining `a-locs` using the framework in Section IV, VSA [3] is employed to determine the set of memory addresses which each direct or indirect memory access instruction can access. Given a set of `a-locs`, VSA can compute an over-approximation of the set of `a-locs` that each register and each `a-loc` holds at a particular program point.

The algorithm is implemented on the IR, but we present our algorithm on C-like pseudo instructions for ease of understanding. Each instruction in the IR implements a transfer function which translates the symbolic maps defined at its input to the symbolic maps at its output. The following definitions are introduced to ease the presentation.

---

$R_i$ : IR (SSA) variables; $e$ : A program point; $r$ : Data object (Var or a-loc)
$SM'_e$ : Map between <code>a-locs</code> and their symbolic value sets after program point <code>e</code>
$SR(r)$ : Mapping of Var $r$ in map SR; $SM_e(r)$ : Mapping of <code>a-loc</code> $r$ in map $SM_e$
$Mem_e(r)$ : Set of memory addresses that $r$ can hold at point <code>e</code> (obtained by VSA)
$(r, SV)$ : Pairing between a data object $r$ and a symbolic value set SV

---

The memory abstraction includes a concept of *fully accessed* and *partially accessed* `a-locs`. In order to understand partial `a-locs`, consider that  $Mem_e(r)$  contains a list of memory addresses that the data object  $r$  can hold at current program point `e`. If this object is dereferenced in a memory access instruction of size  $s$ , the `a-locs`, that are of size  $s$  and whose starting addresses are in set  $Mem_e(r)$ , represents the fully accessed `a-locs`. The partially accessed `a-locs` consists (i) `a-locs` whose starting addresses are in  $Mem_e(r)$  but are not of size  $s$  and

Name	Operation	Transfer Function
1. Assignment	$R1 := R2$	$SR = \{SR - SR(R1)\} \cup \{(R1, SR(R2))\}$
2. Arithmetic	$R3 := R2 \text{ OP } R1$	$if \text{ OP} = +$ $tmp = \nabla(SR(R2) \oplus SR(R1))$ $if \text{ OP} = *$ $tmp = \nabla(SR(R2) \otimes SR(R1))$ $else // Create a new symbolic expression$ $tmp = R3$ $SR = \{SR - SR(R3)\} \cup \{(R3, tmp)\}$
3. Load	$R1 := *(R2)$	$\{F, P\} = *(Mem_e(R2), s)$ $if  P  = 0$ $tmp = \nabla(\bigcup_{v \in F} SM_e(v))$ $else$ $tmp = \top$ $SR = \{SR - SR(R1)\} \cup \{(R1, tmp)\}$
4. Store	$*(R2) := R1$	$\{F, P\} = *(Mem_e(R2), s)$ $if  F  = 1 \ \& \  P  = 0 \ \& \ Func \text{ is not recursive \ \&}$ $F \text{ has no heap a-locs} // Strong Update$ $SM'_e = \{\{SM_e - SM_e(v)\} \cup \{(v, SR(R1))\} \mid v \in F\}$ $else // Weak Update$ $SM'_e = \{\{SM_e - SM_e(y) \mid y \in \{F \cup P\}\} \cup \{(v, \nabla(SR(R1) \cup SM_e(v))) \mid v \in F\} \cup \{(p, \top) \mid p \in P\}\}$
5. SSA Phi	$R_{n+1} = \phi(R_1, R_2, \dots, R_n)$	$SR = \{SR - SR(R_{n+1})\} \cup \{R1, \nabla(\bigcup_{i \in \{1, n\}} SR(R_i))\}$

Table I: Transfer functions for each instruction in a procedure *Func*. Here,  $s$  denotes the size of dereference in a memory access instruction.

(ii) `a-locs` whose addresses are in  $Mem_e(r)$  but whose starting addresses and size do not meet the condition to be fully accessed `a-locs`. Using the notation from [3], this operation is mathematically represented as:

$$\{F, P\} = *(Mem_e(r), s)$$

Here,  $F$  represents the fully accessed and  $P$  represent the partially accessed `a-locs`. As the name suggests, only some portion of a partial `a-loc` is updated or referenced in a memory access instruction. Hence, they are treated conservatively in our analysis, as will be explained below.

Table I shows the mathematical forms of transfer functions for each instruction. Below, each of these transfer functions is discussed in detail.

*1. Assignment:*  $e: R1 := R2$

This is the basic operation where symbolic analysis behaves similarly to the concrete evaluation. Any existing entry in the symbolic map SR corresponding to the variable R1 (computed in an earlier iteration) is removed from the map and the symbolic value set of variable R2 is assigned to variable R1.

*2. Arithmetic Operation:*  $e: R3 := R2 \text{ OP } R1$

In such scenarios, the analysis evaluates the symbolic values according to the underlying mathematical operator. The evaluation is defined for addition, subtraction and multiplication operators. Addition and multiplication are handled by em-

plying the underlying ( $\oplus$ ) and ( $\otimes$ ) operators respectively. Subtraction operation is handled analogous to the addition by reversing the sign of each coefficient in the symbolic expressions of second operand, R1. Since the remaining operations are not represented, a new symbolic expression is introduced to represent the result of the computation.

### 3. Memory Load $e: R1 := *(R2)$

The analysis relies on obtaining the *a-locs* accessed by this instruction. If the current instruction does not access any partial *a-loc*, the symbolic value of variable R1 is computed by unioning the symbolic values corresponding to each of the possible *a-loc*. Otherwise, it is assigned  $\top$ .

### 4. Memory store $e: *(R2) := R1$

The propagation of symbolic values is governed by current memory store accessing a single *a-loc* or multiple *a-locs*. If the current memory store only updates a single fully accessed *a-loc* (strong update), the existing symbolic values of the destination memory location is replaced by the symbolic set. Otherwise, the new symbolic values are unioned with the existing ones to obtain the updated symbolic value set of fully accessed *a-locs* (weak update). The partially accessed *a-locs* are assigned symbolic  $\top$ .

Memory regions corresponding to the stack frame of a recursive procedure or to heap allocations potentially represent more than one concrete *a-loc*. Hence, the assignments to their *a-locs* are also modeled by weak updates.

### 5. SSA Phi Function: $e: R_{n+1} = \phi(R_1, R_2, \dots, R_n)$

At join points in the control flow of a procedure, the symbolic value sets from all the predecessors are unioned to obtain a new symbolic value set.

## C. Interprocedural propagation

Interprocedural analysis requires the correct handling of symbolic values at callsites and return points.

Several binary analysis frameworks [16], [3], including SecondWrite [11], implement various analyses to recognize the arguments. Once the arguments are recognized, formal arguments and returns are represented as a part of procedure definition and actual arguments and returns are explicitly represented as a part of a call instruction in the IR.

The symbolic value set of a formal argument for a procedure  $P$  is computed by unioning the symbolic value sets of corresponding actual arguments across all the call-sites for procedure  $P$ . Mathematically, the initialization of formal  $f_i$  of procedure  $P$ , where  $a_{ci}$  represents the corresponding actual argument at a callsite  $c$ , is represented as

$$SR = \{SR - SR(f_i)\} \cup \{(f_i, \nabla(\bigcup_{\forall c \in CallSites(P)} SR(a_{ci})))\} \quad (2)$$

The return variables are also handled in a similar manner. In order to propagate the symbolic values of memory locations, the memory symbolic maps from each call site need to be unioned to determine the symbolic map at entry point  $P_{entry}$

of a procedure  $P$ .

$$SM_{P_{entry}} = \bigcup_{\forall c \in CallSites(P)} SM_c \quad (3)$$

Similarly, symbolic map just after a call instruction  $C$ , is computed by unioning the symbolic maps at all the return points in the called procedure  $P$ .

The externally called procedures are handled in one of the following three ways. First, procedures which are known not to affect the memory regions (e.g. `puts`, `sin`) are modeled as identity transformers (a NOP). External procedures like `malloc`, which create a memory region, are also modeled as identity transformers since these procedures are already handled by defining a memory abstraction `HeapRgn` corresponding to each allocation site. External procedures like `free`, which destroy a memory region, are conservatively modeled as NOP. Next, unsafe but known external procedures (e.g. `memcpy`) are handled by widening the symbolic value set of all *a-locs* in the memory regions possibly accessed by the procedure. Unknown external procedures (which include user defined libraries) are handled by widening the symbolic value set of registers and all *a-locs* in all the memory regions.

## VI. RESULTS

Our techniques are implemented as part of the Second-Write framework presented in Section III. The evaluation is performed on several benchmarks from the SPEC2006 and OMP2001 suites and some real world programs, as listed in Table II. Benchmarks are compiled with gcc v4.3.1 with O3 flags (Full optimization) and results are obtained on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu.

### A. Functional Representation and Precise Memory Model

Fig 6 and Fig 7 present the statistics regarding our hybrid mechanism for obtaining precise memory model and functional IR. We only present statistics for benchmarks containing non-negligible *Unknown CTIs* (negligible defined as  $\leq 10$  or number of procedures containing *Unknown CTI*  $\leq 1\%$ ). Of 33 programs in Table II, 11 had non-negligible unknown CTIs. Fig 6 presents the fraction of procedures containing *Unknown CTI* in each of these benchmarks. It divides this fraction into scenarios where the static mechanism was able to determine the value of `StackDiff` and where the dynamic mechanism was required to maintain the functionality. Case 1 (Section IV-A) does not arise since we employ the prototypes for standard library procedures. We never hit the invalidation conditions stipulated in Fig 4, justifying the assumptions behind our formulation.

Fig 7 illustrates the additional *a-locs* derived as a result of successful constraint solutions, normalized with respect to original *a-locs* of type `Stack` (Section V-B). As evident, we were able to obtain 10% more *a-locs* in C benchmarks and 30% more *a-locs* in C++ benchmarks on average.

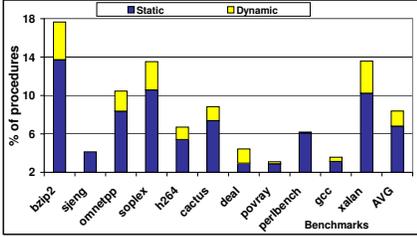


Figure 6: Percentage of procedures with unknown CTIs. The static represents cases when constraint solvers succeeded

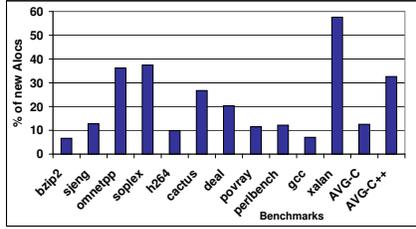


Figure 7: Additional alocs added as a result of constraint solvers, normalized to original number of alocs

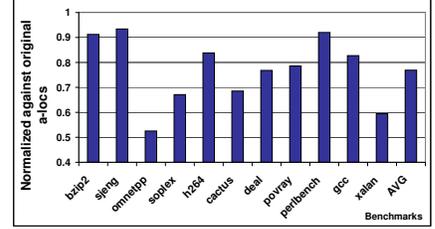


Figure 8: Variables requiring a new symbolic alphabet in presence of additional a-locs

Application	Source	Lang	LOC	# Proc	Time(s)	Mem (MB)
bwaves	Spec2006	F	715	22	4.25	24.47
lbm	Spec2006	C	939	30	0.8	1.03
equake	OMP2001	C	1607	25	0.64	3.62
mcf	Spec2006	C	1695	36	0.31	2.85
art	OMP2001	C	1914	32	0.36	2.74
wupwise	OMP2001	F	2468	43	1.37	5.68
libquantum	Spec2006	C	2743	73	1.30	6.30
leslie3d	Spec2006	F	3024	32	8.24	23.72
namd	Spec2006	C++	4077	193	19.46	111.53
astar	Spec2006	C++	4377	111	1.49	8.39
bzip2	Spec2006	C	5896	51	4.8	90.27
milc	Spec2006	C	9784	172	41.16	19.68
sjeng	Spec2006	C	10628	121	9.93	34.98
sphinx	Spec2006	C	13683	210	7.11	31.19
zeusmp	Spec2006	F	19068	68	37.85	285.48
omnetpp	Spec2006	C++	20393	3980	21.66	58.24
hammer	Spec2006	C	20973	242	12.13	36.52
soplex	Spec2006	C++	28592	1523	21.21	144.14
h264	Spec2006	C	36495	462	29.56	220.53
cactus	Spec2006	C	60452	962	25.65	185.05
gromacs	Spec2006	C/F	65182	674	47.82	252.33
dealII	Spec2006	C++	96382	15619	114.30	240.18
calculix	Spec2006	C/F	105683	771	192.99	404.32
povray	Spec2006	C++	108339	3678	71.01	242.61
perlbench	Spec2006	C	126367	2183	94.18	210.37
gobmk	Spec2006	C	157883	4188	60.66	242.19
gcc	Spec2006	C	236269	6426	280.37	490.68
xalan	Spec2006	C++	267318	30,062	264.97	183.75
gzip	Compress	C	10671	98	1.42	20.06
tar	Compress	C	20518	343	9.58	18.85
ssh	Web client	C	73355	887	40.57	22.55
lynx	Browser	C	135876	2106	140.08	73.01
apache	WebServer	C	232931	2026	37.98	232.12

Table II: Applications Table

This enhanced `a-locs` abstraction is employed in our symbolic value analysis framework.

### B. Symbolic Value Analysis

Table II shows the analysis time and storage requirements of our Symbolic Value Analysis on various applications. The numerical value of `Limit`, the maximum size of a symbolic value set, was kept to 5. The analysis time and the required storage is largely a function of number of procedures in the benchmark. The analysis time is typically low, within 1 minute, for most of the benchmarks except for some intensive benchmarks such as `gcc` and `dealII`.

In order to understand the importance of tracking memory locations, we obtain the percentage of symbolic expressions that containing at least one symbolic alphabet propagated

through a memory location, as a percentage of symbolic expressions for all IR variables. We observe that 35% of symbolic expressions contain alphabets propagated through memory locations. An extended version of the paper [17] presents more detailed results. In absence of an abstraction for memory locations, the analysis would have introduced a new alphabet in all these expressions. This validates our central contribution that tracking memory locations is essential for effective symbolic analysis on executables.

In order to understand our symbolic abstraction, we divided the objects into various categories according to the size of their symbolic value set. On average, 64% of objects can be abstracted with a single symbolic expression in our symbolic domain, 16% of objects need multiple expressions and 20% of objects cannot be represented with finite symbolic abstraction ( $\top$ ) [17]. Maintaining a symbolic value set instead of a single symbolic expression allows us to maintain this extra precision for 16% of data objects.

Fig 8 captures the enhancement in the precision of Symbolic Value Analysis with the presence of additional `a-locs` derived by the constraint mechanism. According to Table I, a load instruction accessing an unknown memory location is represented by a new symbolic alphabet. Fig 8 demonstrates the decrease in the number of load instructions requiring a new alphabet while employing additional `a-locs`. The presence of additional `a-locs` enhances the precision of symbolic value analysis by 10% to 50% in several programs.

### C. Applications

As mentioned before, symbolic analysis is employed in multiple source-level analysis. Here, we demonstrate that our Symbolic Value Analysis enables us to extend several source-level analyses to executables. An extended version [17] explores more applications such as parallelization and alias analysis.

**Value numbering:** It has been shown that even highly optimized executables contain large amount of redundant instructions. For example, Fernandez et al. [18] observed that around 30% of memory references in an optimized program are redundant. Redundancy elimination simplifies the intermediate representation, thereby aiding other optimizations

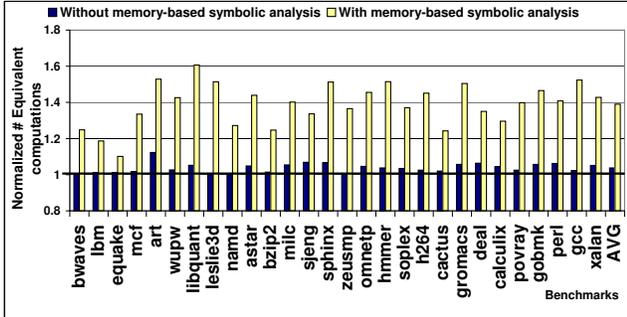


Figure 9: Normalized improvement in detection of equivalent computations (No Symbolic analysis = 1.0)

and speeding up subsequent binary analysis. For example, time taken by bug testing tools for solving a query can be cut in half by simplifying the query first [19].

As explained in Section II, memory based symbolic analysis frameworks obtain more complete symbolic relations between computations in an executable, exposing more equivalences than defined by traditional Value numbering.

We define an abstract interpretation based algorithm on the lattice of symbolic expressions. Fig 9 compares the number of equivalent computations determined in three cases: one when no symbolic analysis is performed, second when symbolic analysis is employed only for variables (obtained by neglecting the transfer functions for memory load and memory store in Table I) and third, when memory based symbolic analysis is employed to determine equivalence. Hence, the second case is similar to existing source-level methods of symbolic analysis since it tracks only variables. As evident, numbering employing memory-based symbolic analysis is able to expose around 40% more equivalent computations in executables than base value numbering (when no symbolic analysis is applied). This figure also shows that symbolic analysis based only on variables is not sufficient in exposing more equivalences in executables and exposes only 3% more equivalences than discoverable when no symbolic analysis is applied. This underscores the importance of maintaining symbolic abstraction for memory locations in improving the efficacy of the applications.

**Security analysis** : Information flow violations represent one of the most serious security challenges in modern software systems. Source-level information-flow frameworks [20] employ scalable thin slicing to accurately reason about information propagation in the presence of pointers. However, the lack of a scalable framework to accurately track memory locations in an executable forces the existing executable analyses to ignore memory references [21], resulting in an imprecise detection of violations. Consequently, most of the executable level frameworks resort to dynamic information-flow analysis for detecting the violations. Our framework statically detects information flow violations in executables with a high degree of precision and a small

Program	Exploit Ref	Type	False Alarms	False Alarms (Source Tools)
muh	CAN-2000-0857	Format String	0	0
pfingerd	NISR16122002B	Format String	2	0
wu-ftpd	CVE-2000-0573	Format String	0	6
gzip	CVE-2005-1228	Dir Traversal	1	0
tar	CVE-2001-1267	Dir Traversal	0	0

Figure 10: Security Analysis

number of false alarms.

Here we evaluate our framework for detecting two important security flaws namely *format string vulnerability* and *directory traversal vulnerability* [22]. Format string flaws arise due to an unsafe implementation of variable-argument functions like *printf* in C library. In such functions, a *format string* argument specifies the number and type of other arguments. However, there is no runtime routine to verify that the function was actually called with the arguments specified by the *format string*. Similarly, a directory traversal vulnerability arises when a filename supplied by a user is employed in a file-access procedure without sufficient validation. Intuitively, information flow violations can be detected by checking the presence of insecure values in the symbolic expressions corresponding to a sensitive variable.

In order to detect these two vulnerabilities, we define the user input functions as *corrupting* functions and variable argument functions and file open functions as *sensitive* functions respectively. The presence of a symbolic alphabet, defined at a callsite of any *corrupting* function, in the symbolic value set corresponding to underlying argument of any *sensitive* function signifies a vulnerability.

Fig 10 shows the evaluation of our method on five programs with known vulnerabilities. As evident, we are able to detect these vulnerabilities, reporting fewer false alarms than source-level tools [22]. Our method fails to detect any of the vulnerabilities if the symbolic propagation across memory locations is disabled.

## VII. RELATED WORK

**Binary analysis**: There has been several binary analysis frameworks such as BitBlaze [19], Jakstab [23], IDAPro [2], CodeSurfer/x86 [3] and others. None of these tools obtain a functional IR or perform customized symbolic analysis. Several binary rewriters such as PLTO [15], UQBT [8] obtain a functional IR, but it has very imprecise memory abstraction, which is not suitable for advanced binary analyses. As described in Section II, IDAPro comes the closest in trying to deal with the problem of indirect CTIs, but they do not guarantee a functional IR.

The work that is closely related to symbolic value analysis are frameworks proposed by Debray et al. [6], Amme et al. [7], Balakrishnan et al. [3] and Guo et al. [9]. Debray et al. [6] and Amme et al. [7] present alias-analysis algorithms for executables. However, their biggest limitation is that they do not track memory locations and hence, lose a great deal of

precision at each memory access. Balakrishnan et al. [3] and Guo et al. [9] present memory analysis algorithms that find an over-approximation of the set of constant and memory address ranges that each abstract data object can hold. However, as presented in Section II, such an abstraction is not suitable for symbolic analysis applications. Further, the IR recovered by these frameworks is not functional.

**Symbolic Analysis:** There has been an extensive body of work employing symbolic analysis for analyzing and optimizing programs. Cousot [24] proposed an early method for discovering the linear relationships between variables. Rugina et al [25] employ symbolic constraint solvers to determine symbolic bounds of each variable. Symbolic analysis has been used extensively to support the detection of parallelism [4]. However, all these above methods obtain symbolic expressions for only the variables and not memory locations, hence they lose a great deal of precision when applied to executables.

**Value numbering:** Several source code algorithms to discover equivalences are based on an algorithm by Kildall [26]. Later, Bodik et al [5] and others proposed more precise algorithms using backward symbolic propagation and path sensitive analysis. However, all these algorithms are based on variables alone and none of these variables propagate value numbers across memory locations.

## VIII. CONCLUSIONS

In this paper, we have proposed techniques to obtain a functional and precise representation from executables and presented methods to adapt symbolic analysis for executables. The improved memory model considerably enhances the precision of our symbolic analysis framework and novel symbolic analysis framework improves the efficacy of various analyses. In the future, we plan to extend this framework for other purposes such as binary understanding.

## REFERENCES

- [1] Announcement for Binary Executable Transforms, <http://www07.grants.gov/>.
- [2] IDAPro disassembler, <http://www.hex-rays.com/idaipro/>.
- [3] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *In CC'04*, pp. 5–23.
- [4] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 477–518, Jul. 1996.
- [5] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *POPL '98*, pp. 237–251.
- [6] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *POPL '98*, pp. 12–24.
- [7] W. Amme, P. Braun, F. Thomasset, and E. Zehendner, "Data dependence analysis of assembly code," *Int. J. Parallel Program.*, vol. 28, no. 5, pp. 431–467, Oct. 2000.
- [8] C. Cifuentes and M. V. Emmerik, "Uqbt: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, 2000.
- [9] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *CGO '05*, pp. 291–302.
- [10] Simplex method in IDA Pro, <http://www.hexblog.com/?p=42>.
- [11] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *EuroSys '13*, pp. 295–308.
- [12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–87.
- [13] M. Smithson and R. Barua, "Binary Rewriting without Relocation Information," *USPTO patent pending no. 12/785,923*, May 2010.
- [14] G. Balakrishnan and T. Reps, "Divine: discovering variables in executables," in *VMCAI'07*, pp. 1–28.
- [15] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A link-time optimizer for the intel ia-32 architecture," in *In Proc. 2001 Workshop on Binary Translation*, 2001.
- [16] J. Zhang, R. Zhao, and J. Pang, "Parameter and return-value analysis of binary executables," in *COMPSAC '07*, pp. 501–508.
- [17] A Symbolic Analysis Framework for analyzing executables, <http://www.ece.umd.edu/~barua/icsm13-extended.pdf>.
- [18] M. Fernández, R. Espasa, and S. K. Debray, "Load redundancy elimination on executable code," in *Euro-Par '01*, pp. 221–229.
- [19] D. Song and et al., "Bitblaze: A new approach to computer security via binary analysis," in *ICISS '08*, pp. 1–25.
- [20] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *PLDI '09*, pp. 87–97.
- [21] M. Cova, V. Felmetzger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *ACSAC '06*, pp. 269–278.
- [22] S. Z. Guyer and C. Lin, "Client-driven pointer analysis," in *SAS'03*, pp. 214–236.
- [23] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *CAV '08*, pp. 423–427.
- [24] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL'78*, pp. 84–96.
- [25] R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *SIGPLAN Not.*, vol. 35, no. 5, pp. 182–195, May 2000.
- [26] G. A. Kildall, "A unified approach to global program optimization," in *POPL '73*, pp. 194–206.

# An Automation-assisted Empirical Study on Lock Usage for Concurrent Programs

Rui Xin\*, Zhengwei Qi\*, Shiqiu Huang\*, Chengcheng Xiang\*, Yudi Zheng<sup>†</sup>, Yin Wang<sup>‡</sup>, and Haibing Guan\*

\*School of Software, Shanghai Jiao Tong University, China  
Email: {arsread, qizhwei, hsqfire, xiangchengcheng, hbguan}@sjtu.edu.cn

<sup>†</sup>University of Lugano, Switzerland  
Email: yudi.zheng@usi.ch

<sup>‡</sup>Facebook, USA  
Email: yinwang@fb.com

**Abstract**—Nowadays concurrent programs are becoming more and more important with the development of hardware and network technologies. However, it is not easy for programmers to write reliable concurrent programs. Concurrency characteristics such as thread-interleaving make it difficult to debug or maintain concurrent programs. Although there are lots of research work on concurrency such as multi-thread testing tools, concurrent program verification and data race detection, all of them leave open problems. For instance, some are not scalable enough for large real world applications and some may report false warnings. Since locks are widely used to protect shared memory, it is beneficial for both programmers and tool designers in all fields to have a good understanding of common lock usage patterns in real world concurrent programs.

This paper reports an empirical study on lock usage in concurrent programs. It is based on our automatic lock analysis tool called LUPA. The study analyzes how lock is used in concurrent programs and how lock usage changes throughout the product environment. In this study, four representative concurrent programs (Apache httpd, Mysql, Aget, Pbzip2) are selected, of which both lock manifestation and lock usage pattern in different versions are studied. This study reveals some interesting findings including but not limited to: (1) about 80.5% of the lock related functions acquire only one lock; (2) simple lock patterns account for 54.5% of all lock usage in real world applications; (3) only 12 out of 527 detected patterns belong to condition lock pattern which may lead to vulnerabilities easily; (4) only 0.65% of the functions are lock related. Additionally, a potential bug caused by problematic locking pattern is found.

**Keywords**—concurrent program; lock usage; empirical study;

## I. INTRODUCTION

### A. Motivation

Nowadays concurrent programs play a significant role in processing huge amounts of data for high performance. However, for programmers, it is not easy to write a bug-free concurrent program. They have to cope with lots of troubles caused by the inherent characteristics of concurrent programs. First, when writing programs, developers focus on the normal behaviours. It is difficult to consider all possible user inputs, all combinations of parameters and all interleavings of multiple threads. In addition, arbitrary thread interleaving makes it hard to debug concurrent programs. Even reproducing a reported

concurrency bug is extremely challenging. As a result, programmers often omit the reproduction step and simply develop a patch based on the problem reported, or sometimes a guess of what may have happened [1].

Similar challenges arise in many other areas such as testing, security and software verification. Recently there are more and more research work focusing on improving the quality of concurrent programs. Unfortunately, there are still many open issues:

1) *Concurrent program testing and verification*: Testing is one of the practical ways to find real bugs during software development. For testing concurrent programs, it is hard to cover the whole exponential interleaving space of concurrent programs. Thus random test and symbolic execution are widely applied to generate test inputs automatically. ConTest [2] randomly generates inputs and injects delays at the program points related to buggy code patterns. But many of the generated test cases are redundant and hence do not improve the coverage while incurring longer testing time. Furthermore the ability of random test to detect concurrency bugs depends on the buggy code patterns. Symbolic execution used in [3] tries to use fewer test cases to achieve higher thread-interleaving coverage and detect more concurrency bugs. However path explosion makes these kinds of approaches hard to test real world concurrent programs. This situation can be ameliorated with a good understanding of the lock usage characteristic and common locking patterns, which leads to better design of testing tools. For example, we can reduce the number of paths explored in symbolic execution based on common lock usage patterns, e.g., how many locks are acquired in a single function.

Similar problems occur in program verification and model checking. Understanding lock usage manifestation and common lock patterns of real world concurrent programs can make verification and model checking tools scalable to large applications and reduce lots of false positives.

2) *Concurrency bug detection*: In order to improve the quality of concurrent programs, lots of research work focus on concurrency bug detection. Data races and deadlocks are two main topics of concurrency bug detection. Many approaches [4]–[9] use static or dynamic analysis methods to detect potential data races. However static approaches can be

complete but may incur lots of false warnings while dynamic approaches are incomplete (e.g., traditional lockset analysis used in [6]) and incur high overhead. A recent study [10] reveals that user defined synchronizations can lead to a lot of false data race detections. Another main topic of concurrency bug detection is deadlock [11]–[13].

Thus dealing with the challenges in existing approaches will benefit from having a good knowledge of common lock usage patterns in the real world by studying popular concurrent programs. For example, if the statistic of lock usage pattern shows that almost all of the functions has a limited number of lock variables, there is no need to pay lots of efforts to recognize distinct lock variables.

Based on the fact that using lock is one of the basic characteristics of concurrent programs and also the common method to control access to shared memory, it is necessary to study how lock is used in real world concurrent programs, including the evolution of lock usage between different versions. Up to now, there are some empirical studies on bug characteristic including concurrency bugs [1], [14], [15]. A study on bug [14] investigates bug characteristic and new factors of bugs. Another study [15] reveals the radio, impact and common patterns in incorrect fixes. A recent study [1] provides a study on concurrency bugs including bug pattern, manifestation and fix strategy. However, there are few researches on lock usage in concurrent programs. Besides, a majority of the existing studies depends on human effort and cost lots of time.

### B. Contribution

To the best of our knowledge, this work provides the first study on lock usage in real world concurrent programs. In this study, we select 4 representative open source concurrent programs of 3 different versions and implement a static, low analysis overhead and low false negative analysis tool named LUPA to automatically collect data of lock usage in these applications. Overall 46650 functions are analyzed, of which 307 are lock related. In order to ensure the accuracy of the collected data, we make efforts to examine the data by manually checking the source code of the target applications and discussing with authors of these applications. Our study concerns about three aspects of lock usage:

- **Lock manifestation:** Language characteristic of lock usage including the amount of lock related function, the amount of distinct lock variables used in a single function and the scope of each distinct lock variable.
- **Lock usage pattern:** Statistic of how lock acquisition and release are performed in each lock related function, how often these lock patterns are used, and other characteristic such as whether function call is used to acquire or release a lock.
- **Lock usage evolution:** The changes of lock manifestation and lock pattern between different versions of an application.

The content of our study and the corresponding findings are summarized in Table I.

```

1 APR_DECLARE(apr_status_t)
   apr_thread_mutex_lock(
   apr_thread_mutex_t *mutex)
2 {
3     apr_status_t rv;
4     rv = pthread_mutex_lock(&mutex->mutex);
5     #ifdef HAVE_ZOS_PTHREADS
6         if (rv) {
7             rv = errno;
8         }
9     #endif
10    return rv;
11 }

```

Fig. 1: Lock wrapper

## II. PRELIMINARIES

In this section, the formal definitions about Lock Wrapper, Unlock Pattern, Lock Related Function, Unique Lock, Lock Pattern are listed as follows.

*Definition Lock Wrapper* is a function that calls the system API such as `pthread_mutex_lock` to acquire a lock without any unlock operation. In Fig. 1, the function `apr_thread_mutex_lock` in Apache httpd 2.2.11 is an example of lock wrapper, the goal of which is to hold a lock after it returns.

Similarly, the concept **Unlock Wrapper** can be defined as a function which uses system API like `pthread_mutex_unlock` in order to release a lock without any lock operation.

*Definition Lock Related Function* is a function that calls lock (unlock) wrappers or API functions directly at least once.

Lock related functions are what we really pay attention to. Unlike other functions, a lock related function is directly associated with lock operation. On one hand, studying on only lock related functions can draw insight from statistic data. On the other hand, it helps reduce both human effort and analysis overhead of automatic tools.

*Definition Unique Lock* is the representation of a set of alias lock references pointing to the same memory area.

For the reason that it is difficult to know how many locks a function may hold during the execution, especially when recursive calls and loops exist. The unique lock variables are identified path insensitively, i.e., variables at the same location in the program are considered identical. The technical detail of unique identification will be discussed later. It is worthwhile to calculate the number of unique locks statically because it is necessary for programmers or verification tools to know whether a function hold more than one lock.

*Definition Lock Pattern* is a pair of a lock and unlock operation with their pre-conditions. Specially, a lock pattern in lock (or unlock) wrapper contains only one lock (or unlock) operation.

A lock pattern shows when and in what condition a function acquires or releases a lock. In this work, how many different lock patterns and how often does each lock pattern

TABLE I: The content of our study and the corresponding findings

Research questions	Findings
How many lock related functions?	Lock related functions are a very small subset of total functions
How many locks a function may acquire?	Most of the functions acquire only one lock
What are the scopes of the locks (global or local)?	Global locks are widely used in concurrent programs
How many lock patterns in an application and how often the common patterns are used?	Compared to condition lock pattern, Simple lock pattern is preferred in real world applications
Whether function call is used to acquire or release a lock?	i) Inter-procedural lock or unlock is frequent ii) Recursive function calls are rarely used
Does lock usage change in different versions?	lock usage changes little with the expanded scale of an application

```

lock (&S->M) ;
...
unlock (&S->M) ;

if (x)
    lock (L)
...
if (x)
    unlock (L)
    
```

(a) Simple lock pattern      (b) Condition lock pattern

```

if (OK != lock (&M))
    return ERROR;
...
unlock (&M) ;
    
```

(c) Test lock pattern

Fig. 2: Code snippets of three lock patterns

happen in each lock related function is studied. Typically, more attention is paid to the three patterns shown in Fig. 4, 5 and 6 respectively. In our experience, these three patterns have a significant impact on causing static analysis tools unsound or impractical.

In the simple lock pattern, a lock is released sequentially after the code following the lock operation. In the condition lock pattern, two conditions are used to control the lock and unlock operation in the same path. In the test lock pattern, the return value of the lock operation is tested and the lock is released if and only if the test result is true.

### III. METHODOLOGY

In this section, the methodology applied in our study is introduced. Firstly, the whole workflow of the study is present in general. Then a subsection is used to explain how to choose the target application in our study. At last, our analysis tool for lock usage recognition is described in detail.

#### A. Workflow

Fig. 3 shows the whole workflow of the study. In the first step, some representative concurrency applications are selected for study. Multiple versions of the source codes are collected for each selected application. Then the source code is translated into LLVM bitcode [16]. After that the generated bitcode is passed to our static analysis tool to collect data of lock manifestation, and to recognize lock patterns automatically. To ensure accuracy, manual investigation is performed with the help of the generated preliminary results by reading the source code and posts in forums or by discussing with developers of

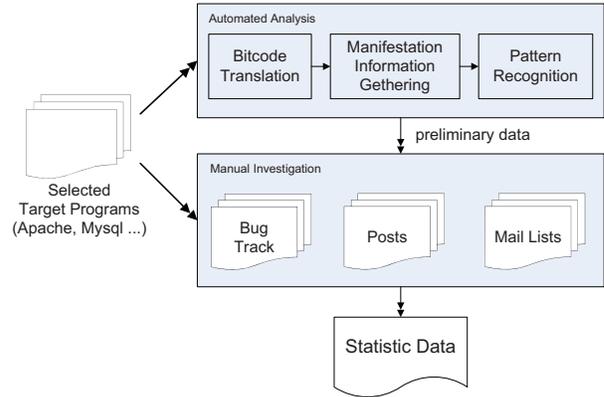


Fig. 3: The work flow of the study

the application. Finally, checked data is formatted and some insight conclusions are drawn from the statistic data.

#### B. Target programs

Four representative applications are chosen as our target programs. In the first place, all of these applications are concurrent programs and use lock to control shared memory access. Then these four applications are real world open source programs whose source codes are available. Also, useful information such as bug tracking, release note and mail list of these applications is easy to find because they have been maturely developed and are well maintained. Their lines of code vary from 364 to 786873 and some of them are under long-term development. In addition, each of them is used in different area. For example, Apache httpd is a web application server that uses concurrency to handle concurrent client requests. Aget<sup>1</sup> is a multi-thread download accelerator that supports HTTP downloads and can be run from the console. Pbzip2<sup>2</sup> is a parallel file compressor that uses pthreads and achieves near-linear speedup on SMP machines. All of above characteristics make these selected applications representative. Then, three available versions of these applications are randomly chosen for analysis and study. The information of the chosen applications is shown in Table II. Table II contains lines of code

<sup>1</sup><http://www.enderunix.org/aget/>

<sup>2</sup><http://compression.ca/pbzip2/>

TABLE II: Descriptive Statistic of Target Programs

Targets	Versions	LOC	NOF	Description
Apache	2.0.46	217477	5220	Web Server
	2.2.11	252993	6472	
	2.2.22	266705	6690	
Aget	0.2	364	8	Http Downloader
	0.2p1	381	8	
	0.4.1	672	18	
Pbzip2	1.1.11	3479	41	File Compressor
	1.1.14	3932	57	
	1.1.16	4167	64	
Mysql	4.0.19	332150	6967	Database Server
	4.1.1-alpha	461798	8664	
	5.0.91	786873	12441	

and numbers of functions in the three versions of the four applications.

### C. Lock pattern recognition

Almost all of the empirical studies on software bug such as [1], [14], [15] significantly rely on human effort, therefore these study consume lots of time and effort. Collecting data of target programs requires good knowledge of them, which is a severe task for non-developer. However, most existing program analysis tools are either impractical for large applications or induce lots of false negative. Besides that, to our knowledge there is few of tools focusing on lock usage analysis. As the result, it is necessary to design a lightweight and conservative analysis tool to assist our study.

In this study, a static analysis tool named LUPA (Lock Usage Pattern Analysis) is implemented to perform inter-procedural analysis for collecting lock manifestation data and detecting all of the lock patterns in target programs. LUPA is efficient and scalable enough to analyze large concurrent programs by using function summary. Its preservation is guaranteed by a subset-based, flow-insensitive, filed-sensitive and context-insensitive alias analysis algorithm discussed later.

In order to facilitate the analysis, firstly the source code is translated to LLVM bitcode, which is in SSA form [17]. Although there already exists a compiler supplied by LLVM, lots of problems still have to be resolved to compile the whole target program into LLVM bit code. In this study, a script wrapper has been written to handle options before being passed into the compiler, for example, removing useless options which may cause compilation failure. Also, official documents of the target applications are referred in order to compile all the modules and make no optimization because some lock or unlock wrapper may be inlined to their callers due to compiler’s optimization.

After the bit code translation, LUPA starts to perform inter-procedural analysis of the generated bit code. It finds and analyzes each lock related function by traversing all functions. Then LUPA builds control flow graph, use-define chains [18] and control dependency graph [19] for each lock related function, and analyzes every path in each lock related function. For each lock or unlock operation including calling wrapper function, the tool will check the lock variable passed into the callee and use alias analysis to distinguish different lock. For

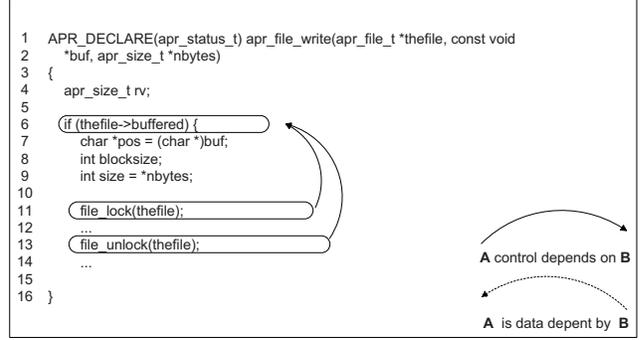


Fig. 4: Simple lock pattern recognition of Apache httpd 2.2.11

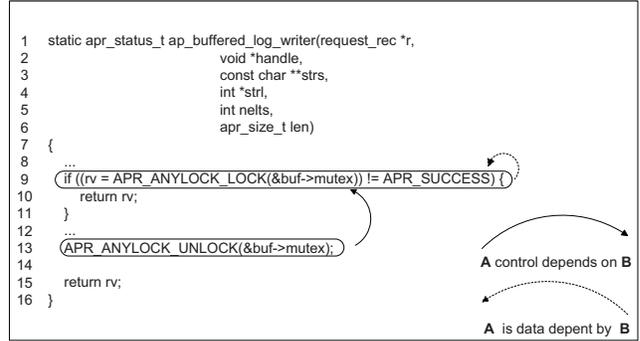


Fig. 5: Test lock pattern recognition of Apache httpd 2.2.11

each distinct lock variable, the tool treats every adjoining lock and unlock operation pair as a lock pattern and recognizes what kind of lock pattern it is.

According to the definitions of different lock patterns, LUPA uses corresponding ways to recognize their types:

- Typically, if both the lock operation and unlock operation control depend on a same condition (e.g., Fig. 4), then it is a simple lock pattern.
- If the unlock control depends on a condition that checks the return value of the lock operation, it is a test lock pattern. As shown in Fig. 5, the unlock operation in line 13 control depends on line 9, and line 9 data depends on the return value of the lock operation.
- Recognizing condition lock pattern is more complex. If the lock operation and the unlock operation have different control dependency nodes while these nodes control depend on the same statement, this pattern is treated as a condition lock pattern. Take the function `allocator_alloc` in Fig. 6 as an example, the lock operation control depends on the statement in line 8 and the unlock operation control depends on line 11. Because line 8 and line 11 control depend on line 7, the usage of lock operation in line 9 and unlock operation in line 12 is a condition lock pattern.

When a function call is encountered, LUPA will check whether the callee function has been analyzed before. If so, the

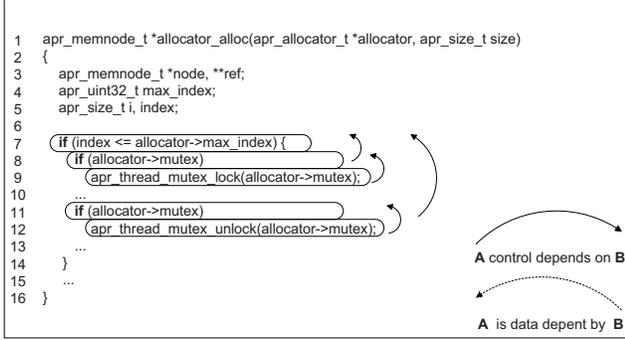


Fig. 6: Condition lock pattern recognition of Apache httpd 2.2.11

function summary of the callee will be used for the context-sensitive inter-procedural analysis at the call site. Otherwise, LUPA firstly analyzes the callee function until all needed data have been collected. Then LUPA creates the function summary of the callee and the analysis returns to the call site. If a lock related function recursively calls itself, it will be analyzed for a bounded time which is set to 2 in this study in order to achieve a conservative result. In fact, recursive self-call rarely appears during the study.

The experimental result of pattern recognition is shown in Table III. The benchmark programs contain one version of each target program. In this experiment, the recognition rate for the 3 special patterns is about 97.2% and there is no false positive.

#### IV. LOCK MANIFESTATION STUDY

In Table IV we show the statistic of lock manifestation from the four target programs. The first column is the name of the target programs and the second column is the selected versions of each program. The third column is the total number of the functions and the fourth column is the number of the lock related functions. GL in Column 5 means global locks' number and LL in Column 6 means local locks' number. Column 7 to 8 show global locks' rate and local locks' rate. The last 3 columns show the number of the function holding 1 lock, 2 locks and more than 2 locks separately. This statistic illustrates: i) how many lock related functions in the programs; ii) how many global and local locks in the programs; iii) how many locks a function may hold. As discussed before, the number of unique locks is calculated statically with the help of alias analysis instead of exact lock variables' number. The calculated locks here include locks acquired by a function and its callees. Distinct from the maximum number of holding locks at the same time, the number of unique locks from the entry of the function to its exit points is paid attention. Global locks refer to global lock variables and local locks are local variables or parameters passed into callee functions. From this statistic we have some findings.

##### A. Lock related functions are a very small subset of total functions

From Table IV we can see that among 46650 function 12 different versions of 4 different programs, only 307 functions

are lock related functions. It is a fact that analyzing all the functions may take a lot of time and resource. Additionally, analyzing the target programs from its entry makes testing tools or verification tools not practical for large-scale concurrent programs. Furthermore, analyzing all the functions may bring little substantive help to concurrency bug detection because all the lock operations are completed inside the lock-related functions. Therefore, paying attention only to lock related function is a practical way to find concurrency bugs. For example, when applying state-of-the-art symbolic execution engines (e.g., KLEE [20]) to large-scale programs, trade off can be made by using hybrid methods. Firstly, fast lockset analysis can be used to identify critical sections where suspected illegal behaviours happen. In those cases, symbolic execution can be applied to inferring the critical section precisely. For the sake that those critical sections only exist in lock related functions, the path explosion problem of classic symbolic execution can be relieved to a great extent.

##### B. Most of the functions acquire only one lock

Calculating the number of unique locks path-insensitively can still reflect the number of locks possibly held by a function. Through learning the lock manifestation statistic, it is obvious that about 80.5% of the observed functions acquire only one lock, 10.7% of the functions acquire 2 locks and only 27 functions acquire more than 2 locks. Acquiring more locks does not mean the function is more reliable and secure. On the contrary, the number of acquired locks refers to the complexity of a function, which implies that there are more potential bugs inside this function. Thus for programmers, it is a good programming practice to keep functional uniqueness and use as less lock as possible in a function. Also, this finding can help reduce the analytical complexity of testing or bug detection tool because under regular circumstance it is not necessary to make every endeavour to distinguish different lock variables. Therefore, functions acquiring only one lock can be treated in a trivial way. Only for a small subset of functions that acquire more locks should we run some nontrivial analysis.

##### C. Global locks are widely used in concurrent programs

It turns out that about 61.6% of the observed locks are global variables. Local locks are heavily used in Apache httpd while in the other three applications most of the locks are global. The reason is that in Apache httpd locks are passed as parameter to lock (unlock) wrapper or other lock relative functions to acquire (release) a lock for better readability and reusability. It means that putting majority of the effort to individual functions can improve the efficiency of the inter-procedural analysis without much accuracy sacrifice.

#### V. LOCK USAGE PATTERN STUDY

Table V presents the statistic of used lock patterns in four selected concurrent programs. The first column is the name of the target programs and the second column is the selected versions of each program. The third is the total number of lock patterns. Column 4 to 7 show the appearance times of simple lock pattern, condition lock pattern, test lock pattern and other patterns separately. The last 3 columns show the number of functions acquiring or releasing locks inter-procedurally, the

TABLE III: Evaluation Result of Pattern Identification

Targets	Version	Patterns	Simple	Identified Simple	Condition	Identified Condition	Test	Identified Test
Apache	2.2.11	28	12	12	7	7	1	1
Aget	0.4.1	2	2	2	0	0	0	0
Pbzip2	1.1.11	31	15	15	0	0	1	1
Mysql	5.0.91	117	65	62	3	3	1	1
Overall	-	178	94	91	10	10	3	3

TABLE IV: The statistic of lock manifestation

Targets	Versions	Functions	LRF	Lock Number	GL	LL	GL%	LL%	1 lock	2 locks	> 2 locks
Apache	2.0.46	5220	13	13	0	13	0	100	13	0	0
	2.2.11	6472	20	20	0	20	0	100	20	0	0
	2.2.22	6690	18	18	0	18	0	100	18	0	0
Aget	0.2	8	1	1	1	0	100	0	1	0	0
	0.2p1	8	1	1	1	0	100	0	1	0	0
	0.4.1	18	1	1	1	0	100	0	1	0	0
Pbzip2	1.1.11	41	16	36	26	10	72.2	27.8	6	4	6
	1.1.14	57	19	45	35	10	77.7	22.3	8	4	7
	1.1.16	64	24	46	36	10	78.2	21.8	13	5	6
Mysql	4.0.19	6967	54	65	42	23	64.4	35.6	45	5	4
	4.1.1-alpha	8664	66	76	40	38	52.6	47.4	57	8	1
	5.0.91	12441	74	86	52	34	60.4	39.6	64	7	3
Overall	-	46650	307	380	234	146	61.6	38.4	247	33	27

number of the functions using recursive calls and the number of lock/unlock wrapper functions.

Firstly, the appearance times of the 3 common lock patterns shown in Fig 4, 5 and 6 are counted. Secondly, the number of lock related functions invoking other lock related functions is recorded as well as the number of lock related functions with recursive lock or unlock operation. At last the number of lock and unlock wrappers is reported. In the lock pattern study, a call to lock (unlock) wrapper is treated as a call to library lock (unlock) function.

#### A. Simple lock pattern is preferred

In Table V we can see that about 54.5% of lock patterns are simple lock pattern. It is used by every version of every selected program. The reason is that using simple lock pattern is easy and safe. Using simple lock pattern ensures that the shared variables are protected and no lock is still held after the function returns if there is no other return statement between these lock and unlock operations. Moreover, writing such kind of lock pattern is easy for programmers. Therefore, programmers prefer to adopt simple lock patterns. This finding can be used to assist verification tools or bug detection tools in reducing analysis space and eliminating false warnings.

#### B. Condition lock pattern is sparingly used in real world applications

Only 25 out of 527 lock patterns belong to condition lock pattern. Condition lock pattern is widely found in Apache httpd but rarely appears in the other three applications. The main reason why condition lock pattern is sparingly used is that errors occur if lock acquiring condition and lock releasing condition are not met. Programmers choose to use condition lock pattern because of requirements of algorithms,

appearance of some new situation during the maintaining or other tough circumstances. Even though programmers try their best to protect that the two flag variables have the same value when tested by the two conditions in a single thread, their values may be modified by multiple thread which may lead to unreleased lock or uninitialized unlock problems. To avoid potential vulnerabilities caused by condition lock pattern, one of the better ways is to refine the original algorithms and redesign the existing data structures. Moreover the heavy use of condition lock pattern makes test case generation tool and bug detection tool difficult to cover all possible paths many of which are infeasible. Therefore, for testing and bug detection tool it had better to spend much effort to other lock patterns.

#### C. Inter-procedural lock or unlock is frequent

In Table V it is clear that acquiring or releasing a lock is a common case and 138 in 307 lock related functions uses callees to perform lock or unlock operation. This finding is not contradictory to the finding that most of the functions acquire only one lock because many of them use lock or unlock wrapper to acquire or release a lock. Take Apache httpd as an example, nearly all the lock related functions call the function `apr_thread_mutex_lock` to acquire a lock and call `apr_thread_mutex_unlock` to release a lock, except these two wrapper functions. Based on this finding, it is necessary for verification tools or bug detection tools to perform inter-procedural analysis and the explored space of inter-procedural analysis can be reduced with the help of the finding that global locks are widely used in concurrent programs. In contrast to inter-procedural lock or unlock operation, recursive lock or unlock appear 3 times in the four selected applications with 12 different versions. Thus, these problems to be solved can be simplified by making the assumption that concurrency bugs should not be related to

TABLE V: The statistic of lock patterns

Targets	Versions	Lock Pattern	Simple	Condition	Test	Other	Inter-procedural	Recursive	Wrapper
Apache	2.0.46	25	4	8	0	13	10	1	3
	2.2.11	28	12	7	1	8	17	1	3
	2.2.22	22	8	7	1	6	14	1	3
Aget	0.2	1	1	0	0	0	0	0	0
	0.2p1	1	1	0	0	0	0	0	0
	0.4.1	2	2	0	0	0	0	0	0
Pbzip2	1.1.11	31	15	0	1	15	11	0	4
	1.1.14	40	16	0	1	23	13	0	3
	1.1.16	56	32	0	1	23	14	0	3
Mysql	4.0.19	91	58	0	0	33	15	0	0
	4.1.1-alpha	113	73	0	0	40	18	0	2
	5.0.91	117	65	3	1	48	26	0	5
Overall	-	527	287	25	6	209	138	3	26

```

1 void* terminatorThreadProc(void* arg)
2 {
3     int ret = pthread_mutex_lock(&
4         TerminateFlagMutex);
5     ...
6     while ((finishedFlag == 0) && (
7         terminateFlag == 0))
8     {
9         ret = pthread_cond_wait(&
10             TerminateCond, &
11             TerminateFlagMutex);
12     }
13     if (finishedFlag != 0)
14     {
15         ret = pthread_mutex_unlock(&
16             TerminateFlagMutex);
17         return NULL;
18     }
19     ret = pthread_mutex_unlock(&
20         TerminateFlagMutex);
21     ...
22     return NULL;
23 }

```

Fig. 7: An example of releasing lock in different branches in Pbzip2 1.1.1

recursive function calls.

Besides the three lock patterns described in Section II, there are other lock patterns. In the followings some other common lock patterns found in the selected applications are described.

Fig. 7 is an example of releasing lock in different branch from Pbzip2 1.1.1. Most of the unclassified patterns are belonging to this type. In this example the function `terminatorThreadProc` acquires a lock in the beginning. Then the function releases the lock in line 13 and returns if the signal `finishedFlag` is non-zero, or the lock is released at the exit point in line 16. All the lock usage in this function is safe because no lock is released without locked before and the function releases all of lock before it returns. However, potential bugs may exist when using this kind of pattern because programmers may forget to release a lock on all branches. In this case, deadlock or other concurrency bugs

```

1 safe_mutex_lock(fifo->mut);
2 while (fifo->full) {
3     ...
4     if (syncGetTerminateFlag() != 0) {
5         pthread_mutex_unlock(fifo->mut);
6         close(hInfile);
7         return -1;
8     }
9 }
10 ...
11 if (queueElement == NULL) {
12     close(hInfile);
13     handle_error(EF_EXIT, -1, "pbzip2 : ...");
14     return -1;
15 }
16 ...
17 safe_mutex_unlock(fifo->mut)

```

Fig. 8: An example of potential concurrency problem using unsafe pattern in Pbzip2 1.1.1

may happen.

Fig. 8 is an example of potential problem using this kind of pattern. In this example, the lock `fifo->mut` is not released after the function returns in line 14. This problem is confirmed after the discussion with the developer. Although the developer claimed that it is not a real bug because the process is terminated and unlikely to cause deadlock, it is easy to cause more other serious errors in the future.

Fig. 9 shows a special lock pattern named inverse lock pattern used in Mysql. In `read_block` a lock passed as parameter is released and it is acquired after that. And this function is called by other functions in while loops to read a page data from file to the block buffer.

## VI. LOCK USAGE EVOLUTION

To understand how lock usage evolves in different versions of an application, the numbers of locks and lock patterns of different versions are calculated and compared. In order to show how lock usage changes with the expanded scale of an application in different versions, two metrics are introduced: the first one is *lock number/lock related function number* and

```

1 static void read_block(KEY_CACHE *keycache,
    BLOCK_LINK *block, uint read_length,
    uint min_length, my_bool primary)
2 {
3     ...
4     if (primary)
5     {
6         keycache_thread_mutex_unlock(&
            keycache->cache_lock);
7         ...
8         keycache_thread_mutex_lock(&
            keycache->cache_lock);
9         ...
10    }
11    ...
12 }

```

Fig. 9: Inverse lock pattern in Mysql 5.0.91

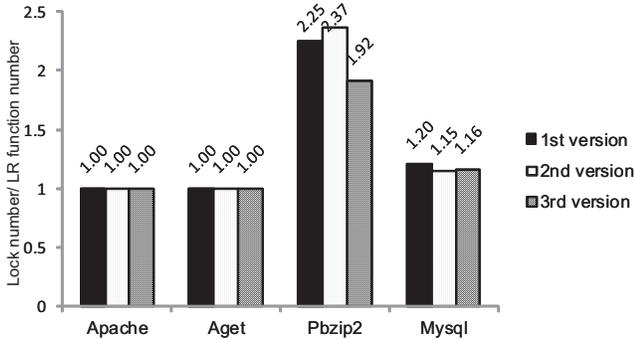


Fig. 10: Trend of lock number in different versions

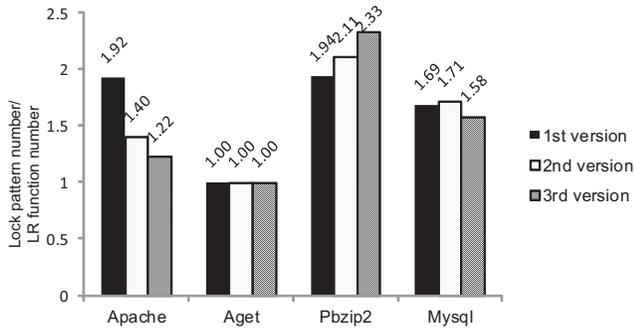


Fig. 11: Trend of lock pattern number in different versions

the second one is *lock pattern number/lock related function number*. Fig. 10 shows the trend of lock number and Fig. 11 presents the evolution of lock pattern. From the two figures it is obvious that although the number of locks and lock patterns varies in different versions, the metrics *lock number/lock related function number* and *lock pattern number/lock related function number* do not change a lot. Another phenomenon is found that lock usage of a function of different version also changes a little. This is because adding or changing locks is not the best way to fix concurrency bugs as described in [1]. Programmers prefer to adopt other ways to fix concurrency

```

(a) version 1.1.11                                (b) version 1.1.14
void *fileWriter(void *outname)
{
...
while(true){
    safe_mutex_lock(OutMutex);
    if (conditionA && (conditionB ||conditionC))
    {
        safe_mutex_unlock(OutMutex);
        usleep(50000);
        continue;
    }
    else
    {
        safe_mutex_unlock(OutMutex);
    }
    ...
}
}

void *fileWriter(void *outname)
{
...
while(true){
    safe_mutex_lock(OutMutex);
    if(newCondition){
        safe_mutex_unlock(OutMutex);
        break;
    }
    if (conditionA && (~conditionB ||conditionC))
    {
        safe_mutex_unlock(OutMutex);
        usleep(50000);
        continue;
    }
    else
    {
        safe_mutex_unlock(OutMutex);
    }
    ...
}
}

```

Fig. 12: Change of lock usage in Pbzip2

bugs such as changing the design of data structure or re-design of algorithms and they have to carefully check if the bug fix introduces no new bugs when adding lock or changing lock usage. Fig. 12 is an example to show how lock usages change in different versions. Fig. 12 (a) comes from the source of Pbzip2 1.1.1 and (b) comes from Pbzip2 1.1.14. In (a) the lock OutMutex is acquired in the beginning of a while loop and is released in each `if...else` branch while in (b) a new condition is added to handle the new situation before the former `if...else` statement. In the new branch the lock OutMutex is released and breaks the while loop.

## VII. RELATED WORK

### A. Bug study

Lots of researches have been done on studying bug characteristics in real world applications. Some of them concern about the common reasons and patterns of bugs. For example, a study [14] uses natural language text classification techniques to study around 29,000 bugs and shows the recent trends of bug characteristics. A study on real world concurrency bug [1] examines bug pattern, manifestation and fix strategy. The conclusions in this empirical study help improve multiple thread testing and concurrency bug detection tools in all directions. Some of them pay attention to some patches with errors that may even introduce new bugs. For example, FIXA-TION [21] introduces a novel method called distance-bounded weakest precondition combined with symbolic execution to check whether a fix cover all the bug-triggering inputs and report whether there is a newly introduced bug. Clone detection is used in [22] to detect bug pattern in concurrent software.

### B. Data race

There are lots of researches on data race detection. They can be generally classified into two categories: static approaches and dynamic approaches. As dynamic approaches, the lockset based method and the happens-before method are diffusely used to detect data race bugs. ERASER [6] is a lockset based dynamic tool that computes the hold lock set of a program and reports a data race bug if a shared memory location is accessed by different threads without being

protected by a lock. It is efficient but imprecise and is inclined to produce false warnings. FASTTRACK [7] reduces most of the VC-based [23] operation to cut down the overhead of happens-before method. It is dynamically sound and precise. There are several implementations of static approaches, such as [4], [8], [24]. RacerX [25] extends the lockset algorithm, and use it to statically analyze data races and deadlocks. It can assure better accuracy, and the results are well ranked for the benefit of users. [26] compares three static analysis tools, whose result is useful for combining them to increase their effectiveness. In addition, there are some approaches that combine lockset and happens-before method to detect potential data race bugs such as [5], [9], [27]. Another extension, MUVI [28], is aimed at multi-variable concurrency bugs and detecting new inconsistent bugs in large applications. Due to the inefficiency of data race detection performed on the software level, hardware-assisted detection is presented. [29] gives a study on the relationship between data dependency and method dependency, which shows that data dependency complements call dependency.

### C. Deadlock

Deadlock is another state-of-art issue of concurrent programs. Up to now there have already been lots of studies on detecting deadlock bugs in concurrent programs, such as [30], [31]. Ahmed K. Elmagarmid made a study on deadlock detection [11] in which most of the common deadlock detection algorithms are discussed. MagicFuzzer [13] is a dynamic deadlock detection method that computes thread-specific lock-dependency relations after the execution of a program and finds deadlock cycles inside the lock-dependency. MagicFuzzer is able to find cycles matched with execution and it is scalable enough for large concurrency applications. A tool [12] on JAVA programs uses static analysis methods to examine the source code and infer lock order graphs to find potential deadlock cycles.

### D. Testing

Owing to the thread-interleaving characteristic of concurrent programs, it is difficult to write test cases that achieve high coverage. Recently there are some effective testing approaches like [2], [20], [32], [33] trying to improve the quality of concurrent programs. Cloud9 [3] copes with the path explosion problem of symbolic execution by parallelizing symbolic execution, and applies it to software testing. Cloud9 facilitates automatic testing of real systems and provides interfaces for writing symbolic test cases. A random testing tool ConTest [2] injects delays at program points related to buggy code patterns to generate random test inputs. The random testing used in ConTest may produce lots of duplicate thread-interleaving test cases causing the low coverage. Unfortunately, the ability of ConTest to detect potential bug relies on the bug patterns passed to it. Compared to random testing, this technique is able to achieve higher synchronization-pair coverage and statement-pair coverage with a lower overhead. BALLERINA [33] generates random multi-thread unit test cases involving only two threads and decreases human effort for examining false alarms by automatically clustering similar reports.

## VIII. CONCLUSION

In this paper, an empirical study on lock usage is presented. To reduce human effort during the study, a static, efficient and conservative analysis tool named LUPA is developed. LUPA is practical to analyze large scale programs and has a high recognition rate of 97.2% for special lock patterns. With the assistant of LUPA, we provide the first empirical study on lock manifestation, lock pattern and lock usage evolution in 3 versions of four concurrent programs (Apache httpd, Aget, Pbzp2 and Mysql). In this study, 46650 functions, 307 lock related functions and 527 lock patterns are investigated. Meanwhile, a potential bug in Pbzp2 1.1.11 is detected during the study. In addition, the useful findings are indicated to guide software testing and the design of bug detection and verification tools. In the future, we will develop some scalable and precise bug detection tools based on the findings in this paper.

## ACKNOWLEDGEMENT

This work was supported by NSFC (No. 61272101), 863 Program (No. 2011AA01A202, 2012AA010905), International Cooperation Program (No. 11530700500, 2011DFA10850), Swiss National Science Foundation (project CRSII2 136225), and Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04C092010).

## REFERENCES

- [1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 329–339. [Online]. Available: <http://doi.acm.org/10.1145/1346281.1346323>
- [2] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, ser. JGI '01. New York, NY, USA: ACM, 2001, p. 181. [Online]. Available: <http://doi.acm.org/10.1145/376656.376848>
- [3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 183–198. [Online]. Available: <http://doi.acm.org/10.1145/1966445.1966463>
- [4] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 320–331. [Online]. Available: <http://doi.acm.org/10.1145/1133981.1134019>
- [5] X. Xie and J. Xue, "Acculock: Accurate and efficient detection of data races," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 201–212. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2190025.2190068>
- [6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: <http://doi.acm.org/10.1145/265924.265927>
- [7] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," *Communications of The ACM*, vol. 53, pp. 93–101, 2010.

- [8] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Proceedings of the 19th international conference on Computer aided verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 226–239. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1770351.1770386>
- [9] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '03. New York, NY, USA: ACM, 2003, pp. 179–190. [Online]. Available: <http://doi.acm.org/10.1145/781498.781529>
- [10] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, "Dynamic recognition of synchronization operations for improved data race detection," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 143–154. [Online]. Available: <http://doi.acm.org/10.1145/1390630.1390649>
- [11] A. K. Elmagarmid, "A survey of distributed deadlock detection algorithms," *SIGMOD Rec.*, vol. 15, no. 3, pp. 37–45, Sep. 1986. [Online]. Available: <http://doi.acm.org/10.1145/15833.15837>
- [12] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for java libraries," in *Proceedings of the 19th European conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 602–629. [Online]. Available: [http://dx.doi.org/10.1007/11531142\\_26](http://dx.doi.org/10.1007/11531142_26)
- [13] Y. Cai and W. K. Chan, "Magicfuzzer: scalable deadlock detection for large-scale applications," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 606–616. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337294>
- [14] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 25–33. [Online]. Available: <http://doi.acm.org/10.1145/1181309.1181314>
- [15] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36. [Online]. Available: <http://doi.acm.org/10.1145/2025113.2025121>
- [16] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–86. [Online]. Available: <http://dl.acm.org/citation.cfm?id=977395.977673>
- [17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: <http://doi.acm.org/10.1145/115372.115320>
- [18] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge University Press, 2002.
- [19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: <http://doi.acm.org/10.1145/24039.24041>
- [20] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1855741.1855756>
- [21] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 55–64. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806812>
- [22] K. Jalbert and J. S. Bradbury, "Using clone detection to identify bugs in concurrent software," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–5.
- [23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: <http://doi.acm.org/10.1145/359545.359563>
- [24] Z. D. Luo, L. Hillis, R. Das, and Y. Qi, "Effective static analysis to find concurrency bugs in java," in *SCAM*, 2010, pp. 135–144.
- [25] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 237–252. [Online]. Available: <http://doi.acm.org/10.1145/945445.945468>
- [26] D. Kester, M. Mwebesa, and J. S. Bradbury, "How good is static analysis at finding concurrency bugs?" in *SCAM*, 2010, pp. 115–124.
- [27] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '03. New York, NY, USA: ACM, 2003, pp. 167–178. [Online]. Available: <http://doi.acm.org/10.1145/781498.781528>
- [28] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 103–116. [Online]. Available: <http://doi.acm.org/10.1145/1294261.1294272>
- [29] H. Kuang, P. Mader, H. Hu, A. Ghabi, L. Huang, L. Jian, and A. Egyed, "Do data dependencies in source code complement call dependencies for understanding requirements traceability?" in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 181–190.
- [30] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *OSDI*, 2008, pp. 295–308.
- [31] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, may 2009, pp. 386–396.
- [32] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 210–220. [Online]. Available: <http://doi.acm.org/10.1145/2338965.2336779>
- [33] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 727–737. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337309>

# Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues

Mark D. Syer<sup>1</sup>, Zhen Ming Jiang<sup>2</sup>, Meiyappan Nagappan<sup>1</sup>, Ahmed E. Hassan<sup>1</sup>, Mohamed Nasser<sup>3</sup> and Parminder Flora<sup>3</sup>  
 Software Analysis and Intelligence Lab<sup>1</sup>, Department of Electrical Engineering & Computer Science<sup>2</sup>, Performance Engineering<sup>3</sup>  
 School of Computing, Queen's University, Canada<sup>1</sup>, York University, Canada<sup>2</sup>, BlackBerry, Canada<sup>3</sup>  
 mdsyer@cs.queensu.ca, zmjiang@cse.yorku.ca, {mei, ahmed}@cs.queensu.ca

**Abstract**—Load tests ensure that software systems are able to perform under the expected workloads. The current state of load test analysis requires significant manual review of performance counters and execution logs, and a high degree of system-specific expertise. In particular, memory-related issues (e.g., memory leaks or spikes), which may degrade performance and cause crashes, are difficult to diagnose. Performance analysts must correlate hundreds of megabytes or gigabytes of performance counters (to understand resource usage) with execution logs (to understand system behaviour). However, little work has been done to combine these two types of information to assist performance analysts in their diagnosis. We propose an automated approach that combines performance counters and execution logs to diagnose memory-related issues in load tests. We perform three case studies on two systems: one open-source system and one large-scale enterprise system. Our approach flags  $\leq 0.1\%$  of the execution logs with a precision  $\geq 80\%$ .

**Keywords**—Performance Engineering; Load Testing; Performance Counters; Execution Logs

## I. INTRODUCTION

The rise of ultra-large-scale (ULS) software systems (e.g., Amazon.com, Google's GMail and AT&T's infrastructure), poses new challenges for the software maintenance field [1]. ULS systems require near-perfect up-time and potentially support thousands of concurrent connections and operations. Failures in such systems are more likely to be associated with an inability to scale, than with feature bugs [2], [3]. This inability to meet performance demands has led to several high-profile failures, including the launch of Apple's MobileMe [4] and the release of Firefox 3.0 [5], with significant financial and reputational repercussions [6], [7].

Load testing has become a critical component in the prevention of these failures. Performance analysts are responsible for performing load tests that monitor how the system behaves under realistic workloads to ensure that ULS systems are able to perform under the expected workloads. Such load tests allow analysts to determine the maximum operating capacity of a system, validate non-functional performance requirements and uncover bottlenecks. Despite the important of load testing, current load test analysis techniques require considerable manual effort and a high degree of system-specific expertise to review hundreds of megabytes or gigabytes of performance counters (to understand resource usage) and execution logs (to understand system behaviour) [2], [8], [9].

Performance analysts must also be aware of a wide variety of performance issues. In particular, memory-related issues, which may degrade performance (by increasing memory management overhead and depleting the available memory) and cause crashes (by completely exhausting the available memory), are difficult to diagnose. Memory-related issues can be broadly classified as transient or persistent. *Transient memory issues* (memory spikes) are large increases in memory usage over a relatively short period of time. *Persistent memory issues* are steady increases in memory usage over time. Persistent memory issues can be further divided into memory bloat (caused by inefficient implementations) and memory leaks (caused by a failure to release unneeded memory). Such issues have led to high profile failures, including the October 22, 2012 failure of Amazon Web Services (caused by a memory leak) that affected thousands of customers [10].

We present a novel approach to support performance analysts in diagnosing memory-related issues in load tests by combining performance counters and execution logs. First, we abstract the execution logs into execution events. We then combine the performance counters and execution events by discretizing them into time-slices. Finally, we use statistical techniques to identify a set of execution events corresponding to a memory-related issue.

Our approach focuses on the *diagnosis*, as opposed to the *detection*, of memory-related issues. Performance analysts could use a variety of existing techniques to detect issues prior to using our approach for diagnosis. For example, performance analysts may plot memory usage over time to determine whether there are any persistent memory issues (where the memory usage continually increases) or compare the minimum, mean and maximum memory usage to determine whether there are any transient memory issues.

This paper makes two main contributions:

- 1) Existing load test analysis techniques use either execution logs or performance counters. We present the first approach that combines both sources to diagnose memory-related issues in load tests.
- 2) Our approach is fully automated and scales well with large-scale enterprise systems, flagging  $\leq 0.1\%$  of the log lines for further analysis by system experts with a precision  $\geq 80\%$ .

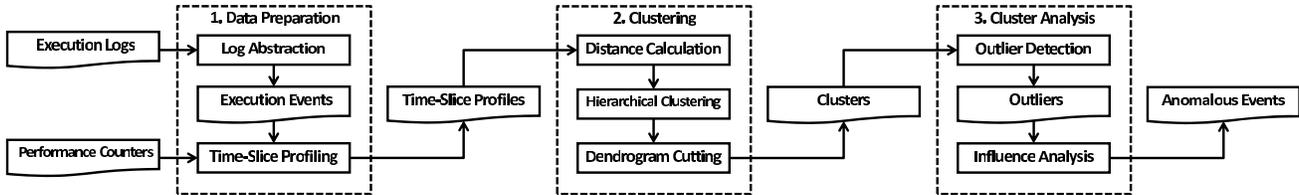


Fig. 1. Overview of Our Approach.

This paper is organized as follows: Section II provides a motivational example of how our approach may be used in practice. Section III describes our approach in detail. Section IV presents the setup and results of our case studies and Section V discusses how changes to the input data impact our results. Section VI outlines the threats to validity of our case studies. Section VII presents related work. Finally, Section VIII concludes the paper and presents future work.

## II. MOTIVATIONAL EXAMPLE

Jack, a performance analyst, performed a 24 hour load test on a ULS system. He discovers that there may be a persistent memory issue in the system based on a visual inspection of memory usage, which continues to increase throughout the entire load test. However, in order to properly report this issue to the developers, Jack must understand the underlying cause (i.e., the usage scenario and functionality that causes this issue). This is a very challenging task because Jack needs to correlate the 3 million log lines and 2,000 samples of memory usage that were collected during the load test.

Jack is introduced to a new load test analysis approach to help him diagnose the memory issue uncovered during the load test. When applied to the execution logs and performance counters that Jack collected, this approach flags 10 events, less than 0.001% of the log lines, for further analysis. These events directly correspond to specific usage scenarios and functionality within the system.

The new approach has produced a much smaller data set that Jack is able to manually analyze. Jack analyzes this much smaller set of execution events and concludes that this memory issue is caused by a particular usage scenario where an error handler fails to release memory back to the system once the error has occurred (i.e., a memory leak). Jack reports this issue, along with the associated events, to the developers.

## III. APPROACH

This section outlines our approach to diagnose memory-related issues in load tests by combining and leveraging the information provided by performance counters and execution logs. Figure 1 provides an overview of our approach. We describe each step in detail below.

The first step in our approach is data preparation. In this step we abstract the execution logs into execution events and combine the performance counters and execution events by discretizing them into time-slices. Each time-slice represents a period of time where we can measure the number of execution

events that have occurred and the change in memory usage. The second step is to cluster the time-slices into groups where a similar set of events have occurred. The third step in our approach is to identify the events that are most likely to correspond to the functionality that is causing the memory issue by analyzing the clusters.

We will demonstrate our approach with a working example of a real-time chat application.

### Input Data

*Execution Logs:* Execution logs describe the occurrence of important events in the system. They are generated by output statements that developers insert into the source code of the system. These output statements are triggered by specific execution events (e.g., starting, queueing or completing a job or encountering a specific error). Execution logs record notable events at runtime and are used by developers (to debug a system) and system administrators (to monitor the operation of a system).

The second column of Table I presents the execution logs from our working example. These execution logs contain both static (e.g., `starts a chat`) and dynamic (e.g., `Alice and Bob`) information.

*Performance Counters:* Performance counters describe system resource usage (e.g., CPU usage, memory usage, network I/O and disk I/O). Memory usage may be measured by a number of different counters, such as 1) the amount of allocated virtual memory, 2) the amount of allocated private (non-shared) memory and 3) the size of the working set (amount of memory used in the previous time-slice). Performance analysts must identify and collect the set of counters that are relevant to their system. Each of these counters may then be analyzed independently. Performance counters are sampled at periodic intervals by resource monitoring tools (e.g., PerfMon) [11].

Table II presents the performance counters (i.e., memory usage) for our working example. A transient memory issues is seen at 00:12 (i.e., memory spikes to 100).

TABLE II  
PERFORMANCE COUNTERS

Time	Memory (MB)
00:00	10
00:04	15
00:08	20
00:12	100
00:16	20
00:20	10

TABLE I  
ABSTRACTING EXECUTION LOGS TO EXECUTION EVENTS

Time	Log Line	Execution Event	Execution Event ID
00:01	Alice starts a chat with Bob	USER starts a chat with USER	1
00:01	Alice says 'hi' to Bob	USER says MSG to USER	2
00:02	Bob says 'hello' to Alice	USER says MSG to USER	2
00:05	Charlie starts a chat with Dan	USER starts a chat with USER	1
00:05	Charlie says 'here is the file' to Dan	USER says MSG to USER	2
00:06	Alice says 'are you busy?' to Bob	USER says MSG to USER	2
00:08	Initiate file transfer (Charlie to Dan)	Initiate file transfer (USER to USER)	3
00:13	Complete file transfer (Charlie to Dan)	Complete file transfer (USER to USER)	4
00:14	Charlie ends the chat with Dan	USER ends the chat with USER	5
00:17	Bob says 'yes' to Alice	USER says MSG to USER	2
00:18	Alice says 'ok, bye' to Bob	USER says MSG to USER	2
00:18	Alice ends the chat with Bob	USER ends the chat with USER	5

### 1. Data Preparation

The first step in our approach is to prepare the execution logs and performance counters for automated, statistical analysis. Data preparation is a two-step process. First, we remove implementation and instance-specific details from the execution logs to generate a set of execution events. Second, we count the number of execution events and the change in memory usage over each sampling interval.

*Log Abstraction:* Execution logs are not typically designed for automated analysis. Each occurrence of an execution event results in a slightly different log line, because each log line contains static components as well as dynamic information (which may be different for each occurrence of the execution event). Therefore, we must remove this dynamic information from the log lines prior to our analysis in order to identify similar execution events. We refer to the process of identifying and removing dynamic information from a log line as “abstracting” the log line.

Our technique for abstracting log lines recognizes the static and dynamic components of each log line using a technique similar to token-based code cloning techniques [12]. In addition to preserving the static components of each log line, some dynamic information is also partially preserved. This is because some dynamic information may be relevant to memory-issues (e.g., the size of a queue or file). Therefore, this dynamic information is partially preserved by abstracting the numbers into ranges (e.g., quantiles or the order of magnitude).

In order to verify the correctness of our abstraction, many execution logs and their corresponding execution events have been manually reviewed by system experts.

Table I presents the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event for automated analysis and brevity) for the log lines in our working example.

*Time-Slice Profiling:* We combine performance counters with the execution events using time stamps. When a log line is generated or a performance counter is sampled, the log line or performance counter is written to a log/counter file along with the date and time of generation/sampling.

Although performance counters and execution logs both contain time stamps, combining these two is a major challenge. This is because performance counters are sampled at periodic

intervals, whereas execution logs are generated continuously. Therefore, we must discretize the execution logs such that they co-occur with the performance counters.

Discretization also allows us to account for the delayed impact of some functionality on the performance counters. For example, there may be a slight delay between when a log line is generated and when the associated functionality is executed. Discretization also helps to reduce the overhead imposed on the system during load testing because the performance counter sampling frequency can be reduced.

During the period of time between two successive samples of the performance counters (i.e., a “time-slice”), zero or more log lines may be generated by events occurring within the system. For example, a log line may be generated when a new work item is started, queued or completed or a specific error is encountered during the time-slice. The execution events are then discretized by creating a profile for each time-slice. This profile is created by counting the number of times that each type of execution event occurred during the time-slice and by calculating the change in memory usage between the start and end of the time-slice. Therefore, each time-slice has a profile with two components: 1) a log activity component, which is a count of each execution event that has occurred during the time-slice and 2) a memory delta over the time-slice. We refer to the process of connecting the performance counters with the execution events as “profiling” the time-slices.

Our profiling technique is agnostic to the contents and format of the performance counters and execution logs. We do not rely on transaction/thread/job IDs and we do not assume any tags other than a time stamp.

Table III shows the results of profiling the time-slices from our working example.

TABLE III  
TIME-SLICE PROFILES

Time	Log Activity (Execution Event ID)					Memory Delta
	1	2	3	4	5	
00:04	1	2	0	0	0	5
00:08	1	2	0	0	0	5
00:12	0	0	1	0	0	80
00:16	0	0	0	1	1	-80
00:20	0	2	0	0	1	-10

## 2. Clustering

The second step in our approach is to cluster the time-slice profiles into groups with similar log activity (i.e., where a similar set of events have occurred). This is because we expect that similar log activity should lead to similar memory deltas. Memory-related issues will impact these memory deltas. We have automated the clustering step using robust statistical techniques to account for the size of the data.

*Distance Calculation:* Each time-slice profile is represented by one point in a multi-dimensional space. Clustering procedures rely on identifying points that are “close” in this multi-dimensional space. Therefore, we must specify how distance is to be measured in this space. Larger distance between two points imply a greater dissimilarity between the time-slice profiles that these points represent. We calculate the distance between the log activity component of every pair of time-slice profiles. This produces a distance matrix.

We use the Pearson distance, as opposed to the many other distance measures [13]–[15], as this measure often results in a clustering that is closer to the true clustering [14], [15].

We first use the Pearson correlation to calculate the similarity between two profiles. This measure ranges from -1 to +1, where a value of 1 indicates that two profiles are identical, a value of 0 indicates that there is no relationship between the profiles and a value of -1 indicates an inverse relationship between the profiles (i.e., as the occurrence execution logs increase in one profile, they decrease in the other).

$$\rho = \frac{n \sum_i x_i \times y_i - \sum_i x_i \times \sum_i y_i}{\sqrt{(n \sum_i x_i^2 - (\sum_i x_i)^2) \times (n \sum_i y_i^2 - (\sum_i y_i)^2)}} \quad (1)$$

where  $x$  and  $y$  are the log activity components of two time-slice profiles and  $n$  is the number of execution events. We then convert the Pearson correlation to the Pearson distance.

$$d_\rho = \begin{cases} 1 - \rho & \text{for } \rho \geq 0 \\ |\rho| & \text{for } \rho < 0 \end{cases} \quad (2)$$

Table IV presents the distance matrix produced by calculating the Pearson distance between every pair of time-slice profiles in our working example.

TABLE IV  
DISTANCE MATRIX

	00:04	00:08	00:12	00:16	00:20
00:04	0	0.333	0.408	0.408	0.667
00:08	0.333	0	0.612	0.612	0.167
00:12	0.408	0.612	0	0.25	0.408
00:16	0.408	0.612	0.25	0	0.388
00:20	0.667	0.167	0.408	0.388	0

*Hierarchical Clustering:* We cluster the time-slice profiles (i.e., to group time-slices where a similar set of logs have occurred) using the distance matrix and an agglomerative, hierarchical clustering procedure. This procedure starts with each profile in its own cluster and proceeds to find and merge the closest pair of clusters (using the distance matrix), until only one cluster (containing everything) is left. Every time two clusters are merged, the distance matrix is updated.

Hierarchical clustering updates the distance matrix based on a specified linkage criteria. We use the average linkage, as opposed to the many other linkage criteria [13], [16], as this linkage is the most appropriate when little information about the expected clustering (e.g., the relative size of the expected clusters) is available. Every time two clusters are merged, the average linkage criteria removes the merged clusters from the distance matrix and adds the new cluster by calculating the distance between the new cluster and all existing clusters. The distance between two clusters is the average distance (as calculated by the Pearson distance) between the profiles of the first cluster and the profiles of the second cluster [13], [16].

Figure 2 shows the dendrogram produced by hierarchically clustering the time-slice profiles using the distance matrix (Table IV) from our working example.

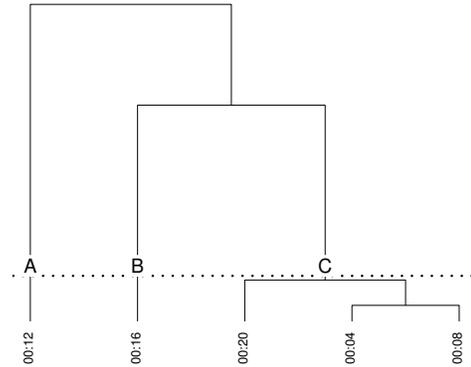


Fig. 2. Sample Dendrogram.

*Dendrogram Cutting:* The result of a hierarchical clustering procedure is a hierarchy of clusters that are typically visualized using hierarchical cluster dendrograms (e.g., Figure 2). These are binary tree-like diagrams that show each stage of the clustering procedure as nested clusters [16].

To complete the clustering procedure, the dendrogram must be cut at some height. This results in a clustering where each time-slice profile is assigned to only one cluster. Such cutting of the dendrogram is done either by manual (visual) inspection or by statistical tests (referred to as stopping rules).

Although a visual inspection of the dendrogram is flexible and fast, it is subject to human bias and may not be reliable. We use the Calinski-Harabasz stopping rule, as opposed to the many other stopping rules [17]–[21], as this rule is commonly referred to as the most accurate [19]. The Calinski-Harabasz stopping rule is a pseudo-F-statistic, which is a ratio reflecting within-cluster similarity and between-cluster dissimilarity. The optimal clustering will have high within-cluster similarity (i.e., the time-slice profiles within a cluster are very similar) and a high between-cluster dissimilarity (i.e., the time-slice profiles from two different clusters are very dissimilar).

The horizontal line in Figure 2 shows how the Calinski-Harabasz stopping rule is used to cut the dendrogram from our working example into three clusters. Cluster A contains one time-slice profile (00:12), cluster B contains one (00:16) and cluster C contains three (00:20, 00:04 and 00:08).

TABLE V  
SCORING TECHNIQUES FOR IDENTIFYING OUTLYING CLUSTERS

	Transient Memory Issues	Persistent Memory Issues	
	<i>Memory Spike</i>	<i>Memory Bloat</i>	<i>Memory Leak</i>
<b>Motivation</b>	Functionality causing a memory spike is characterized by a higher than average memory delta in small clusters or a combination of a higher than average memory delta and higher than average memory delta standard deviation in larger clusters. The standard deviation component is scaled with the cluster size to emphasize the standard deviation component in larger clusters.	Functionality causing memory bloat is characterized by a higher than average memory delta.	Functionality causing a memory leak is characterized by a combination of a higher than average memory delta and a higher than average memory delta standard deviation. The standard deviation component is added because functionality with variable memory deltas may indicate a memory leak.
<b>Score</b>	$spike_i = \frac{n_i \times \sigma_i}{\max(n \times \sigma)} + \frac{\mu_i}{\max \mu}$	$bloat_i = \frac{\mu_i}{\max \mu}$	$leak_i = \frac{\sigma_i}{\max \sigma} + \frac{\mu_i}{\max \mu}$
	$\mu_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \Delta memory_{i,j} \quad \sigma_i = \sqrt{\frac{1}{n_i - 1} \sum_{j=1}^{n_i} (\Delta memory_{i,j} - \mu_i)^2}$ <p>where <math>i</math> is the cluster number, <math>n_i</math> is the size of cluster <math>i</math>, <math>\mu_i</math> is the average memory delta across each time-slice profiles, <math>j</math>, in cluster <math>i</math>, <math>\sigma_i</math> is the standard deviation of the memory deltas of cluster <math>i</math> and <math>spike_i</math>, <math>bloat_i</math> and <math>leak_i</math> are the scores assigned to cluster <math>i</math>. <math>n</math>, <math>\mu</math> and <math>\sigma</math> are vectors containing the cluster size, average memory delta and standard deviation of the memory deltas of all clusters. The standard deviation of a cluster with one time-slice profile is arbitrarily assigned the maximum standard deviation (i.e., such that the first component of the <i>spike</i> and <i>leak</i> scores is equal to one).</p>		

### 3. Cluster Analysis

The third step in our approach is to identify the log lines that correspond to the functionality that is responsible for the memory issue exhibited by the data set. First, outlying clusters are detected. Second, the key log lines of the outlying clusters are identified. As in our previous step, we have used robust statistical techniques to automate this step.

*Outlier Detection:* We identify outlying clusters by examining the memory deltas from each of the time-slice profiles within each of the clusters. Outlying clusters will contain time-slice profiles that have a significant impact on memory usage (as evidenced by the memory deltas). Given the wide variety of memory-related performance issues, identifying time-slice profiles that have the “right” impact on memory usage is a major challenge.

After discussions with system experts and a review of memory-related issue reports from an enterprise system, we developed scoring techniques to identify clusters that contain evidence of these memory issues. Table V presents the scoring technique used to identify transient memory issues (i.e., memory spikes) and persistent memory issues (i.e., memory bloat or memory leaks).

We calculate either the memory spike (*spike*) or memory bloat (*bloat*) and memory leak (*leak*) scores for each cluster, depending on whether a transient or persistent memory issue is detected. Outlying clusters are identified as having a score that is more than twice the standard deviation above the average

score (i.e., a z-score greater than 2). The z-score is the number of standard deviations a data point is from the average.

Table VI presents the *spike* score for each of the clusters in our working example (i.e., each of the clusters that were identified when the Calinski-Harabasz stopping rule was used to cut the dendrogram in Figure 2). We calculate the *spike* score because a transient memory issues is seen at 00:12 (i.e., memory spikes to 100 at 00:12).

TABLE VI  
*Spike Scores for Figure 2 Clusters*

Cluster	<i>Spike Score</i>
Cluster A	2
Cluster B	0
Cluster C	1
$\mu_{spike}$	1
$\sigma_{spike}$	1

From Table VI, we find that the average *spike* score ( $\mu_{spike}$ ) is 1 and the standard deviation in the *spike* scores ( $\sigma_{spike}$ ) is 1. Therefore, no clusters are identified as outliers (i.e., there are no *spike* scores  $\geq \mu_{spike} + 2 \times \sigma_{spike} = 3$ ). However, outliers are extremely difficult to detect in such small data sets. Therefore, for the purposes of this working example, we will use one standard deviation (as opposed to two standard deviations). Consequently, we identify Cluster A as an outlying cluster (as it is one standard deviation above the average).

*Influence Analysis:* We perform an influence analysis on the outlying clusters to determine which execution events differentiate the time-slice profiles in outlying clusters from the average (“normal”) time-slice profile. These execution events are most likely to be responsible for the cause of memory-related issues.

We first calculate the centre of the outlying cluster and the universal centre. The centre of the outlying cluster is calculated by averaging the count of each event in all the time-slice profiles of a cluster. Similarly, the universal centre is calculated by averaging the count of each event in all the time-slice profiles of all clusters. These centres represent the location, in an  $n$ -dimensional space (where  $n$  is the number of unique execution events), of each of the clusters, as well as the average (“normal”) time-slice profile.

We then calculate the Pearson distance (Equation 1 and Equation 2) between the centre of the outlying cluster and the universal centre. This “baseline” distance quantifies the difference between the time-slice profiles in outlying clusters and the universal average time-slice profile.

We then calculate the change in the baseline distance between the outlying cluster’s centre and the universal centre with and without each execution event. This quantifies the influence of each execution event. When an overly influential execution event is removed, the outlying cluster becomes more similar to the universal average time-slice profile (i.e., closer to the universal center).

Therefore, overly influential execution events are identified as any execution event that, when removed from the distance calculation, decreases the distance between the outlying cluster’s centre and the universal centre by more than twice the standard deviation above the average change in distance.

Table VII presents the change in the distance between Cluster A and average (“normal”) time-slice profile when each event is removed from the distance calculation.

TABLE VII  
IDENTIFYING OVERLY INFLUENTIAL EXECUTION EVENTS

Event ID	$\Delta d_p$
1	$4.267 \times 10^{-2}$
2	$1.999 \times 10^{-1}$
3	$-3.774 \times 10^{-1}$
4	$1.487 \times 10^{-1}$
5	$4.267 \times 10^{-2}$
$\mu_{\Delta d_p}$	$1.131 \times 10^{-2}$
$\sigma_{\Delta d_p}$	$2.278 \times 10^{-1}$

From Table VII, the average  $\Delta d_p$  ( $\mu_{\Delta d_p}$ ) is  $1.131 \times 10^{-2}$  and the standard deviation in  $\Delta d_p$  ( $\sigma_{\Delta d_p}$ ) is  $2.278 \times 10^{-1}$ . Therefore, no clusters are identified as outliers (i.e., no  $\Delta d_p$  is  $\geq \mu_{\Delta d_p} - 2 \times \sigma_{\Delta d_p} = -4.442 \times 10^{-1}$ ). Again, for the purposes of this example, we will use one standard deviation (as opposed to two standard deviations). Therefore, we identify event 3 as overly influential. This flagged event corresponds to initiating the transfer of a file. Performance analysts and developers now have a concrete starting point for their investigation of this memory issue.

## IV. CASE STUDY

This section outlines the setup and results of our case study. First, we provide an overview of the subject systems. We then present a case study using a Hadoop application. Finally, we discuss the results of an enterprise case study.

### A. Subject Systems

Our case studies use performance counters and execution logs from two systems. Table VIII outlines the systems and data sets used in our case study.

*Hadoop Case Study:* Our first system is an application that is built on Hadoop. Hadoop is an open-source distributed data processing platform that implements the MapReduce data processing framework [22], [23].

MapReduce is a distributed data processing framework that allows large amounts of data to be processed in parallel by the nodes of a distributed cluster of machines [23]. The MapReduce framework consists of the Map component, which divides the input data amongst the nodes of the cluster, and the Reduce component, which collects and combines the results from each of the nodes.

*Enterprise System Case Study:* Our second system is a large-scale enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system’s architecture, however the system is responsible for simultaneously processing thousands of client requests and has very high performance requirements.

### B. Hadoop Case Study

Our Hadoop case study focuses on the WordCount application [24]. The WordCount application is a standard example of a Hadoop application that is used to demonstrate the Hadoop MapReduce Framework.

*WordCount:* The WordCount application [24] reads one or more text files (a corpus) and counts the number of times each unique word occurs within the corpus. The output is one or more text files (depending on the number of unique words in the corpus), with one unique word and the number of times that word occurs in the corpus per line.

*Load Test Configuration:* We load test the Hadoop WordCount application on our cluster by attempting to count the number in times each unique word occurs in two 150MB files and one 15GB file. Linefeeds and carriage-returns are removed from one of the 150MB files so that the file is composed on one line.

According to the official Hadoop documentation: *as the Map operation is parallelized the input file set is first split to several pieces called FileSplits. If an individual file is so large that it will affect seek time it will be split to several Splits. The splitting does not know anything about the input file’s internal logical structure... [25].*

Input files are split using linefeeds or carriage-returns [24], [26]. Therefore, attempting to read the one-line 150MB file (which lacks linefeeds and carriage-returns) will result in a persistent memory issue (i.e., memory bloat as the application attempts to read the entire file into memory).

TABLE VIII  
CASE STUDY SUBJECT SYSTEMS.

	Hadoop	Enterprise System	
Application domain	Data processing	Telecom	
License	Open-source	Enterprise	
Memory issue	Memory bloat	Memory leak	Memory spike
Load test duration	49.2 minutes	17.5 hours	45.5 hours
Number of log lines	5,303	2,685,838	182,298,912
Number of flagged events	1	10	4
Reduction in analysis effort	1 - 0.019%	1 - 0.00037%	1 - 0.0000022%
Precision	100%	80%	100%

*Application Failure:* During the load test, the application fails prior to processing the input files. As expected, the cause of the failure is a memory-related issue. The following log line (an error message) is seen in Hadoop’s execution logs, indicating that the WordCount application has a memory issue (i.e., the application is trying to allocate more memory than available in the heap): `FATAL org.apache.hadoop.mapred.TaskTracker: Task: attempt_id - Killed: Java heap space`

Further, a plot of memory usage (i.e., memory heap usage) for the WordCount application on the virtual machine with the failure (Figure 3) clearly shows a persistent memory issue.

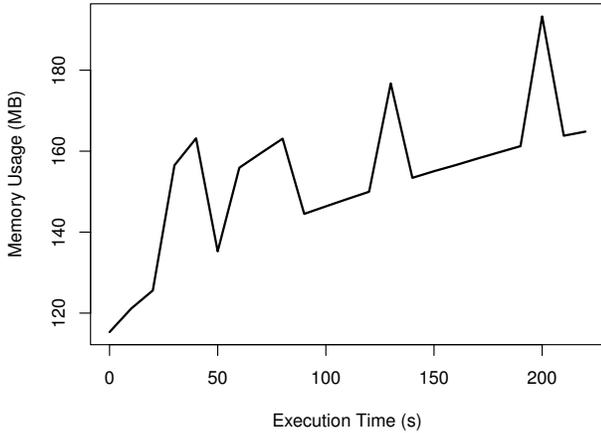


Fig. 3. Memory Usage

*Applying Our Approach:* We apply our approach to the execution logs and performance counters collected during our load test. Our approach identifies the following log line as most likely to be responsible for this issue and flags it for further analysis: `INFO org.apache.hadoop.mapred.TaskTracker attempt_id hdfs file start 1 end 8`

*Results:* Our approach flagged only one log line for expert analysis. As expected, this log line relates to data input. Further, the numbers in this log line (e.g., “start 1 end 8”) relates to the order of magnitude of the seek position (*start*) within the file and the number of bytes (*end*) to be read. As expected, our approach has flagged the execution event where the data is read and split amongst the nodes of the cluster for processing [26]. Our approach managed to reduce 5,303 log lines into 1 execution event (a  $100\% - 0.019\% = 99.981\%$  reduction in analysis effort) with a precision of 100% (i.e., this log line is relevant to the memory issue).

### C. Enterprise System Case Study

Although the results of our Hadoop study were promising, we apply our approach to two load tests of our Enterprise System to examine the scalability of our approach. Similar to our Hadoop data set, these load tests have exposed memory issues, however, they are significantly larger than our Hadoop data set. From Table VIII, we find that our enterprise case studies comprise of data sets that contain 500 times more log lines from load tests that are 20 times longer than our Hadoop data set.

We perform two case studies using the performance counters and execution logs collected during two separate load tests of the system that have been performed and analyzed by system experts. The first load test exposed a memory leak caused by a specific piece of functionality. The second load test exposed a memory spike caused by rapidly queueing a large number of work items. Each of these load tests have been analyzed by system experts and their conclusions have been verified by at least one additional expert. Therefore, we are confident that the true cause of the memory issues have been correctly identified.

Our approach was applied to the performance counters and execution logs collected during each of these load tests. We flag 10 log lines (0.00037% of the 2,685,838 log lines) and 4 log lines (0.0000022% of the 182,298,912 log lines) for the memory leak and memory spike data sets respectively. We correctly identified 8 log lines (80% precision) and 4 log lines (100% precision) for the memory leak and memory spike data sets respectively. These log lines correspond directly to the functionality and usage scenario that system experts have determined to be the cause of the memory issue. For confidentiality reasons, we cannot disclose the log lines our approach has flagged or the functionality and usage scenario to which they correspond. However, our results have been independently verified by system experts.

## V. DISCUSSION

Although our approach has been fully automated to analyze the execution logs and performance counter generated during a load test, how this data is generated and collected is open to the system’s developers and load testers. Developers must ensure that they are inserting accurate execution logs that cover the system’s features and load testers must specify how often the performance counters are sampled (i.e., the sampling interval).

The sampling interval of our data sets range from 5 seconds (Hadoop data set) to 30 seconds and 3 minutes (Enterprise data sets). To explore how varying the sampling interval would impact our results, we simulate longer sampling intervals in

the data set where a memory leak was found in our Enterprise System. The sampling interval for this data set is 30 seconds, however we simulate longer sampling intervals by merging successive time-slice profiles. For example, a 60 second sampling interval can be created by counting the execution events and calculating the memory delta over two successive 30 second intervals. Using this approach, we simulate 60, 90, 120, 150 and 180 second sampling intervals.

Table IX presents how the number of flagged events, the precision (the percentage of correctly flagged events) and the recall is impacted by an increased sampling interval. As we do not have a gold standard data set, we calculate recall using the best results in Table IX. Sampling intervals between 90 and 150 seconds correctly flag 13 events. Hence, we measure recall as the percentage of these 13 events that are flagged.

TABLE IX  
IMPACT OF INCREASING THE SAMPLING INTERVAL

	Sampling Interval					
	30s	60s	90s	120s	150s	180s
Flagged events	10	5	15	15	15	2
Precision (%)	80	100	87	87	87	0
Recall (%)	62	38	100	100	100	0

From Table IX, we find that sampling intervals between 90 seconds and 150 seconds flag events with high precision and recall. However, performance analysts may need to tune this parameter based on the duration of their load tests and the sampling overhead on their system.

## VI. THREATS TO VALIDITY

### A. Threats to Construct Validity

1) *Monitoring Performance Counters*: Our approach is based on the ability to identify log lines that have a significant impact on memory usage. This is based on the assumption that memory is allocated when requested and the allocation of memory can be reliably monitored. Our approach should be able to correctly identify the cause of memory issues in any system that shares this property. To date, we have not yet encountered a system where this property does not hold.

2) *Timing of Events*: Our approach is also based on the ability to combine the performance counters (specifically memory usage) and execution logs. This is done using the date and time from each log line and performance counter sample. However, large-scale software systems are often distributed, therefore the timing of events may not be reliable [27]. However, the performance counters and execution logs used in our case studies were generated from the same machine. Therefore, there are no issues regarding the timing of events. System experts also agree that this timing information is correct.

The timing of events may also be impacted if the time stamps in the performance counters and execution logs do not reliably reflect when the counters/logs were sampled/generated. However, we have found that the time stamps reflect the times that the counters/logs were sampled/generated, as opposed to the time the counters/logs were written to a file. Therefore, the time stamps of the performance counters and execution logs reliably reflect the true order of the events.

Our approach should only be used when the performance counters and execution logs are reliably collected.

3) *Evaluation*: We have evaluated our approach by determining the precision with which our approach flags execution events (i.e., the percentage of flagged events that are relevant to the memory issue). While system analysts have verified these results, we do not have a gold standard data set. Further, complete system knowledge would be required to identify all of the execution events that are relevant to a particular issue. Therefore, we cannot calculate the recall of our approach. However, our approach is intended to help performance analysts diagnose memory-related issues by flagging execution events for further analysis (i.e., to provide analysts with a starting point). Therefore, our goal is to maximize precision so that analysts have confidence in our approach. In our experience, performance analysts agree with this view. Additionally, we were able to identify at least one event that was relevant to the memory issue at hand in all three case studies.

### B. Threats to Internal Validity

1) *Selecting Performance Counters*: Our approach requires performance counters measuring memory usage. However, memory usage may be measured by a number of different counters including, 1) allocated virtual memory, 2) allocated private (non-shared) memory and 3) the size of the working set (amount of memory used in the previous time-slice). Performance analysts should sample all of the counters that may be relevant. Once the load test is complete, performance analysts can then detect whether memory-related issues are seen in any of the counters and use our approach to diagnose these issues. However, performance analysts may require system-specific expertise to select an appropriate set of performance counters.

2) *Execution Log Quality/Coverage*: Our approach assumes that the cause of any memory issue is manifested within the execution logs (i.e., there are log lines associated with the exercise of this functionality). However, it is possible that there are no execution logs to indicate when certain functionality is exercised. Therefore, our approach is incapable of identifying this functionality in the case that they cause memory issues. Further, our approach is incapable of identifying functionality that does not occur while performance counters are being collected (e.g., if the system crashes). However, this is true for all execution log based analysis, including manual analysis.

This issue may be mitigated by utilizing automated instrumentation tools that would negate the need for developers to manually insert output statements into the source code. However, we leave this to future work as automated instrumentation imposes a heavy overhead on the system [28].

### C. Threats to External Validity

1) *Generalizing Our Results*: The studied software systems represent a small subset of the total number of software systems. Therefore, it is unclear how our results will generalize to additional systems, particularly systems from other domains (e.g., e-commerce). However, our approach does not assume any particular architectural details.

## VII. RELATED WORK

Our approach is a form of dynamic program analysis, however much of the existing work on dynamic analysis focuses on the functional behaviour of a system (except for some work on visualizing threads [29], [30]), whereas we focus on the performance of a system. Cornelissen et al. present an excellent survey of dynamic analysis [31]. Performance testing, load test analysis, performance monitoring and software ageing are the closest areas of research to our work.

### A. Performance Testing

Grechanik et al. propose a novel approach to performance testing based on black-box software testing [32]. Their approach analyzes execution traces to learn rules describing the computational intensity of a workload based on the input data. An automated test script then selects test input data that potentially expose performance bottlenecks (where such bottlenecks are limited to one or few components). Our approach is not limited to finding performance bottlenecks and relies on existing testing infrastructure.

### B. Load Test Analysis Using Execution Logs

Jiang et al. mine execution logs to determine the dominant (expected) behaviour of the application and to flag anomalies from the dominant behaviour [9]. Their approach is able to flag <0.01% of the execution log lines for closer analysis. Our approach does not assume that performance problems are associated with anomalous behaviour.

Jiang et al. also flag performance issues in specific usage scenarios by comparing the distribution of response times for the scenario against a baseline derived from previous tests [2]. Their approach reports scenarios that have performance problems with few false positives (77% precision). Our approach does not rely on baselines derived from previous tests.

### C. Load Test Analysis Using Performance Counters

Load test researchers have also used performance counters to detect performance problems and identify the probable cause of performance regressions [8], [28], [33]–[35].

Foo et al. use association rule mining to extract correlations between the performance counters collected during a load test [33]. The authors compare the correlations from one load test to a baseline to identify performance deviations. Nguyen et al. use control charts to identify load tests with performance regressions and detect which component is the cause of the regression [34], [35]. Malik et al. have used principle component analysis (PCA) to generate performance signatures for each component of a system. The authors assess the pair-wise correlations between the performance signatures of a load test and a baseline to identify performance deviations with high accuracy (79% accuracy) [8], [28].

Our approach does not rely on performance baselines derived from previous tests. Further, our approach can pinpoint the cause of performance problems at a much lower level (i.e., execution log level) compared to Nguyen et al. [34], [35] and Malik et al. [8], [28] (i.e., the component level). Finally, our

approach focuses on *diagnosing* (i.e., discovering the cause) memory issues that have already been *detected*.

In our previous work, we proposed an approach to identify performance deviations in thread pools using performance counters [36], [37]. Our approach identified performance deviations (e.g., memory leaks) with high precision and recall. However, we did not make use of execution logs. Hence, we could not identify the underlying cause of these deviations.

### D. Performance Monitoring

Research in automated performance monitoring has developed application signatures based on performance counters that can be used to detect changes to the performance of an application as it evolves over time [38]–[40]. However, these methodologies require a baseline model of the application's performance in order to characterize changes resulting from software evolution and maintenance.

### E. Software Ageing Monitoring

Work in software ageing has developed monitoring techniques to detect the effects of software ageing. Software ageing is defined as the progressive degradation of a system's performance during its operational lifetime [41]. This degradation is typically caused by resource exhaustion. Researchers have noted the importance of memory issues in software ageing; memory is the most cited cause of software ageing [42]–[50].

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel approach to combine the information provided by both performance counters and execution logs. Our approach is intended to help performance analysts diagnose memory-related performance issues.

We performed three case studies using Hadoop and an Enterprise System. Each of our case studies investigated a different memory issue (i.e., memory bloat, memory leak and memory spike). We have shown that our approach can correctly diagnose memory issues.

Although our approach performed well in diagnosing memory issues, we intend to explore our ability to diagnose other performance issues (e.g., CPU spikes).

### ACKNOWLEDGEMENT

We would like to thank BlackBerry for providing access to the enterprise system used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of BlackBerry's products.

### REFERENCES

- [1] S. E. Institute, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.
- [2] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Proceedings of the International Conference on Software Maintenance*, Sep 2009, pp. 125–134.
- [3] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, Dec 2000.

- [4] "Steve Jobs on MobileMe," [www.arstechnica.com/apple/2008/08/steve-jobs-on-mobileme-the-full-e-mail/](http://www.arstechnica.com/apple/2008/08/steve-jobs-on-mobileme-the-full-e-mail/), Last Accessed: 17-Apr-2013.
- [5] "Firefox Download Stunt Sets Record For Quickest Melt-down," [www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/](http://www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/), Last Accessed: 17-Apr-2013.
- [6] "IT Downtime Costs \$26.5 Billion In Lost Revenue," [www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441](http://www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441), Last Accessed: 17-Apr-2013.
- [7] "The Avoidable Cost of Downtime," <http://www.arcsolve.com/us/lpg/costofdowntime.aspx>, Last Accessed: 17-Apr-2013.
- [8] H. Malik, "A methodology to support load test analysis," in *Proceedings of the International Conference on Software Engineering*, May 2010, pp. 421–424.
- [9] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the International Conference on Software Maintenance*, Oct 2008, pp. 307–316.
- [10] "Summary of the October 22, 2012 AWS Service Event in the US-East Region," <http://aws.amazon.com/message/680342/>, Last Accessed: 17-Apr-2013.
- [11] "PerfMon," [www.technet.microsoft.com/en-us/library/bb490957.aspx](http://www.technet.microsoft.com/en-us/library/bb490957.aspx), Last Accessed: 17-Apr-2013.
- [12] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance and Evolution*, vol. 20, no. 4, pp. 249–267, Jul 2008.
- [13] I. Frades and R. Matthiesen, "Overview on techniques in cluster analysis," *Bioinformatics Methods In Clinical Research*, vol. 593, pp. 81–107, Mar 2009.
- [14] A. Huang, "Similarity measures for text document clustering," in *Proceedings of the New Zealand Computer Science Research Student Conference*, Apr 2008, pp. 44–56.
- [15] N. Sandhya and A. Govardhan, "Analysis of similarity measures with wordnet based text document clustering," in *Proceedings of the International Conference on Information Systems Design and Intelligent Applications*, Jan 2012, pp. 703–714.
- [16] P.-N. Tan, M. Steinbach, and V. Kumar, *Cluster Analysis: Basic Concepts and Algorithms*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., 2005.
- [17] T. Calinski and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics*, vol. 3, no. 1, pp. 1–27, Jan 1874.
- [18] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, 1st ed. John Wiley & Sons Inc, 1973.
- [19] G. W. Milligan and M. C. Cooper, "An examination of procedures for determining the number of clusters in a data set," *Psychometrika*, vol. 50, no. 2, pp. 159–179, Jun 1985.
- [20] R. Mojena, "Hierarchical grouping methods and stopping rules: An evaluation," *The Computer Journal*, vol. 20, no. 4, pp. 353–363, Nov 1977.
- [21] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, no. 1, pp. 53–65, Nov 1987.
- [22] "Hadoop," [www.hadoop.apache.org/](http://www.hadoop.apache.org/), Last Accessed: 17-Apr-2013.
- [23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008.
- [24] "MapReduce Tutorial," [http://hadoop.apache.org/docs/stable/mapred\\_tutorial.html](http://hadoop.apache.org/docs/stable/mapred_tutorial.html), Last Accessed: 17-Apr-2013.
- [25] "HadoopMapReduce," <http://wiki.apache.org/hadoop/HadoopMapReduce>, Last Accessed: 17-Apr-2013.
- [26] "TextInputFormat," <http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/TextInputFormat.html>, Last Accessed: 17-Apr-2013.
- [27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul 1978.
- [28] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Automatic comparison of load tests to support the performance analysis of large enterprise systems," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Mar 2010, pp. 222–231.
- [29] S. P. Reiss, "Efficient monitoring and display of thread state in java," in *Proceedings of the International Workshop on Program Comprehension*, May 2005, pp. 247–256.
- [30] —, "Controlled dynamic performance analysis," in *Proceedings of the International Workshop on Software and Performance*, Jun 2008, pp. 43–54.
- [31] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, Sep 2009.
- [32] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proceedings of the International Conference on Software Engineering*, Jun 2012, pp. 156–166.
- [33] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, K. Martin, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Proceedings of the International Conference on Quality Software*, Jul 2010, pp. 32–41.
- [34] T. H. D. Nguyen, "Using control charts for detecting and understanding performance regressions in large software," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, Apr 2012, pp. 491–494.
- [35] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated verification of load tests using control charts," in *Proceedings of the Asia-Pacific Software Engineering Conference*, Dec 2011, pp. 282–289.
- [36] M. D. Syer, B. Adams, and A. E. Hassan, "Industrial case study on supporting the comprehension of system behaviour," in *Proceedings of the International Conference on Program Comprehension*, Jun 2011, pp. 215–216.
- [37] —, "Identifying performance deviations in thread pools," in *Proceedings of the International Conference on Software Maintenance*, Sep 2011, pp. 83–92.
- [38] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2008, pp. 452–461.
- [39] —, "Automated anomaly detection and performance modeling of enterprise applications," *Transactions on Computer Systems*, vol. 27, no. 3, pp. 6:1–6:32, Nov 2009.
- [40] N. Mi, L. Cherkasova, K. Ozonat, J. Symons, and E. Smirni, "Analysis of application performance and its change via representative application signatures," in *Proceedings of the Symposium on Network Operations and Management*, Apr 2008, pp. 216–223.
- [41] D. L. Parnas, "Software aging," in *Proceedings of the international conference on Software engineering*, May 1994, pp. 279–287.
- [42] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A methodology for detection and estimation of software aging," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov 1998, pp. 283–292.
- [43] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, Sep 2010.
- [44] M. Grottko, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of software aging in a web server," *Transactions on Reliability*, vol. 55, no. 3, pp. 411–420, Sep 2006.
- [45] M. Grottko, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proceedings of the International Workshop on Software Aging and Rejuvenation*, Nov 2008, pp. 1–6.
- [46] A. Macedo, T. B. Ferreira, and R. Matias, "The mechanics of memory-related software aging," in *Proceedings of the International Workshop on Software Aging and Rejuvenation*, Nov 2010, pp. 1–5.
- [47] R. Matias, B. Evangelista, and A. Macedo, "Monitoring memory-related software aging: An exploratory study," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, Nov 2012, pp. 247–252.
- [48] R. Matias and P. J. F. Filho, "An experimental study on software aging and rejuvenation in web servers," in *Proceedings of the International Computer Software and Applications Conference*, Sep 2006, pp. 189–196.
- [49] Q. Ni, W. Sun, and S. Ma, "Memory leak detection in sun solaris os," in *Proceedings of the International Symposium on Computer Science and Computational Technology*, Dec 2008, pp. 703–707.
- [50] M. Shereshevsky, J. Crowell, B. Cukic, V. Gandikota, and Y. Liu, "Software aging and multifractality of memory resources," in *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2003, pp. 721–730.

# Exploring the Limits of Domain Model Recovery

Paul Klint, Davy Landman, Jurgen Vinju  
 Centrum Wiskunde & Informatica, Amsterdam, The Netherlands  
 {Paul.Klint, Davy.Landman, Jurgen.Vinju}@cwi.nl

**Abstract**—We are interested in re-engineering families of legacy applications towards using Domain-Specific Languages (DSLs). Is it worth to invest in harvesting domain knowledge from the source code of legacy applications?

Reverse engineering domain knowledge from source code is sometimes considered very hard or even impossible. Is it also difficult for “modern legacy systems”? In this paper we select two open-source applications and answer the following research questions: which parts of the domain are implemented by the application, and how much can we manually recover from the source code? To explore these questions, we compare manually recovered domain models to a reference model extracted from domain literature, and measured precision and recall.

The recovered models are accurate: they cover a significant part of the reference model and they do not contain much junk. We conclude that domain knowledge is recoverable from “modern legacy” code and therefore domain model recovery can be a valuable component of a domain re-engineering process.

## I. INTRODUCTION

There is ample anecdotal evidence [1] that the use of Domain-Specific Languages (DSLs) can significantly increase the productivity of software development, especially the maintenance part. DSLs model expected variations in both time (versions) and space (product families) such that some types of maintenance can be done on a higher level of abstraction and with higher levels of reuse. However, the initial investment in designing a DSL can be prohibitively high because a complete understanding of a domain is required. Moreover, when unexpected changes need to be made that were not catered for in the design of the DSL the maintenance costs can be relatively high. Both issues indicate how both the quality of domain knowledge and the efficiency of acquiring it are pivotal for the success of a DSL based software maintenance strategy.

In this paper we investigate the source code of existing applications as valuable sources of domain knowledge. DSLs are practically never developed in green field situations. We know from experience that rather the opposite is the case: several comparable applications by the same or different authors are often developed before we start considering a DSL. So, when re-engineering a family of systems towards a DSL, there is opportunity to reuse knowledge directly from people, from the documentation, from the user interface (UI) and from the source code. For the current paper we assume the people are no longer available, the documentation is possibly wrong or incomplete and the UI may hide important aspects, so we scope the question to recovering domain knowledge from source code. Is valuable domain knowledge present that can be included in the domain engineering process?

From the field of reverse engineering we know that recovering this kind of design information can be hard [2]. Especially for legacy applications written in low level languages, where code is not self-documenting, it may be easier to recover the information by other means. On the other hand, if a legacy application was written in a younger object-oriented language, should we not expect to be able to retrieve valuable information about a domain? This sounds good, but we would like to observe precisely how well domain model recovery from source code could work in reality. Note that both the quality of the recovered information and the position of the observed applications in the domain are important factors.

### A. Positioning domain model recovery

One of the main goals of reverse engineering is *design recovery* [2] which aims to recover design abstractions from any available information source. A part of the recovered design is the domain model.

Design recovery is a very broad area, therefore, most research has focused on sub-areas. The *concept assignment problem* [3] tries to both discover human-oriented concepts and connect them to the location in the source code. Often this is further split into *concept recovery*<sup>1</sup> [4]–[6], and *concept location* [7]. Concept location, and to a lesser extent concept recovery, has been a very active field of research in the reverse engineering community.

However, the notion of a concept is still very broad and *features* are an example of narrowed-down concepts and one can identify the sub-areas of *feature location* [8] and *feature recovery*. *Domain model recovery* as we will use in this paper is a closely related sub-area. We are interested in a pure domain model, without the additional artifacts introduced by software design and implementation. The location of these artifacts is not interesting either. For the purpose of this paper, a domain model (or model for short) consists of entities and relations between these entities.

Abebe et al.’s [9], [10] *domain concept extraction* is similar to our sub-area. As is Ratiu et al.’s [11] *domain ontology recovery*. In Section IX we will further discuss these relations.

### B. Research questions

To learn about the possibilities of domain model recovery we pose this question: how much of a domain model can be recovered under *ideal* circumstances? By ideal we mean that the applications under investigation should have well-structured and self-documenting object-oriented source code.

<sup>1</sup>Also known as *concept mining*, *topic identification*, or *concept discovery*.

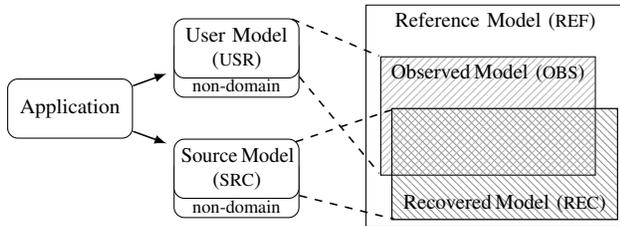


Fig. 1. Domain model recovery for one application.

This leads to the following research questions:

- Q1. Which parts of the domain are implemented by the application?
- Q2. Can we manually recover those implemented parts from the object-oriented source code of an application?

Note that we avoid automated recovery here because any inaccuracies introduced by tool support could affect the validity or accuracy of our results.

Figure 1 illustrates the various domains that are involved: The *Reference Model (REF)* represents all the knowledge about a specific domain and acts as oracle and upper limit for the domain knowledge that can be recovered from any application in that domain. The *Recovered Model (REC)* is the domain knowledge obtained by inspecting the source code of the application. The *Observed Model (OBS)* represents the part of the reference domain that an application covers, i.e. all the knowledge about a specific application in the domain that a user may obtain by observing its external behavior and its documentation but not its internal structure.

Ideally, both domain models should completely overlap, however, there could be entities in OBS not present in REC and vice versa. Therefore, figure 2 illustrates the final mapping we have to make, between SRC and USR. The *Intra-Application Model (INT)* represents the knowledge recovered from the source code, also present in the user view, without limiting it to the knowledge found in REF.

In Section II we describe our research method, explaining how we will analyze the mappings between USR and REF (OBS), SRC and REF (REC), and SRC and USR (INT) in order to answer Q1 and Q2. The results of each step are described in detail in Sections III to VIII. Related work is discussed in Section IX and Section X (Conclusions) completes the paper.

## II. RESEARCH METHOD

In order to investigate the limits of domain model recovery we study *manually* extracted domain models. The following questions guide this investigation:

- A) Which domain is suitable for this study?
- B) What is the upper limit of domain knowledge, or what is our reference model (REF)
- C) How to select two representative applications?
- D) How do we recover domain knowledge that can be observed by the user of the application (Q1 & OBS)?
- E) How do we recover domain knowledge from the source code (Q2 & REC)?

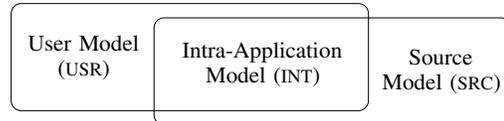


Fig. 2. INT is the model in the shared vocabulary of the application, unrelated to any reference model. It represents the concepts found in both the USR and SRC model.

- F) How do we compare models that use different vocabularies (terms) for the same concepts? (Q1, Q2)?
- G) How do we compare the various domain models to measure the success of domain model recovery? (Q1,Q2)?

We will now answer the above questions in turn. Although we are exploring manual domain model recovery, we want to make this manual process as traceable as possible since this enables independent review of our results. Where possible we automate the analysis (calculation of metrics, precision and recall), and further processing (visualization, table generation) of manually extracted information. Both data and automation scripts are available online.<sup>2</sup>

### A. Selecting a target domain

We have selected the domain of project planning for this study since it is a well-known, well-described, domain of manageable size for which many open source software applications exist. We use the Project Management Body of Knowledge (PMBOK) [12] published by Project Management Institute (PMI) for standard terminology in the project management domain. Note that as such the PMBOK covers a lot more than just project planning.

### B. Obtaining the Reference Model (REF)

Validating the results of a reverse engineering process is difficult and requires an oracle, i.e., an *actionable* domain model suitable for comparison and measurement. We have transformed the descriptive knowledge in PMBOK into such a reference model using the following, traceable, process:

- 1) Read the PMBOK book.
- 2) Extract project planning facts.
- 3) Assign a number to each fact and store its source page.
- 4) Construct a domain model, where each entity, attribute, and relation are linked to one or more of the facts.
- 5) Assess the resulting model and repeat the previous steps when necessary.

The resulting domain model will act as our Reference Model. and Section III gives the details.

### C. Application selection

In order to avoid bias towards a single application, we need at least two project planning applications to extract domain models from. Section IV describes the selection criteria and the selected applications.

<sup>2</sup>See <http://homepages.cwi.nl/~landman/icsm2013/>.

#### D. Observing the application

A user can observe an application in several ways, ranging from its UI, command-line interface, configuration files, documentation, scripting facilities and other functionality or information exposed to the user of the application. In this study we use the UI and documentation as proxies for what the user can observe. We have followed these steps to obtain the User Model (USR) of the application:

- 1) Read the documentation.
- 2) Determine use cases.
- 3) Run the application.
- 4) Traverse the UI depth-first for all the use cases.
- 5) Collect information about the model exposed in the UI.
- 6) Construct a domain model, where each entity and relation are linked to a UI element of the application.
- 7) Assess the resulting model and repeat the previous steps when necessary.

We report about the outcome in Section V.

#### E. Inspecting the source code

We have designed the following traceable process to extract a domain model from each application's source code, the Source Model (SRC):

- 1) Read the source code as if it is plain text.
- 2) Extract project planning facts.
- 3) Store its filename, and line number (source location).
- 4) Construct a model, where each entity, attribute, and relation is linked to a source location in the application's source code.
- 5) Assess the model and repeat the previous steps when necessary.

The results appear in Section VI.

#### F. Mapping models

After performing the above steps we have obtained five domain models for the same domain, derived from different sources:

- The Reference Model (REF) derived from PMBOK.
- For each of the two applications:
  - User Model (USR).
  - Source Model (SRC).

While all these model are in the project planning domain, they all use different vocabularies. Therefore, we have to manually map the models to the same vocabulary. Mapping the USR and SRC models onto the REF model, gives the Observed (OBS) and Recovered Model (REC).

The final mapping we have to make, is between the SRC and USR models. We want to understand how much of the User Model (USR) is present in the Source Model (SRC). Therefore, we also map the SRC onto the USR model, giving the Intra-Application Model (INT). The results of all these mappings are given in Section VII.

#### G. Comparing models

To be able to answer Q1 and Q2, we will compare the 11 produced models. Following other research in the field of concept assignment, we use the most common information

retrieval (IR) approach, *recall* and *precision*, for measuring quality of the recovered data. Recall measures how much of the expected model is present in the found model, and precision measures how much of the found model is part of the expected.

To answer Q1, the recall between REF and USR (OBS) explains how much of the domain is covered by the application. Note that the result is subjective with respect to the size of REF: a bigger domain may require looking at more different applications that play a role in it. By answering Q2 first, analyzing the recall between USR and SRC (INT), we will find out whether source code could provide the same recall as REF and USR (OBS). The relation between REF and SRC (REC) will confirm this conclusion. Our hypothesis is that since the selected applications are small, we can only recover a small part of the domain knowledge, i.e. a low recall.

The precision of the above mappings is an indication of the quality of the result in terms of how much extra (unnecessary) details we accidentally would recover. This is important for answering Q2. If the recovered information would be overshadowed by junk information<sup>3</sup>, the recovery would have failed to produce the domain knowledge as well. We hypothesize that due to the high-level object-oriented designs of the applications we will get a high precision.

Some more validating comparisons, their detailed motivation and the results of all model comparisons are described in Section VIII.

### III. PROJECT PLANNING REFERENCE MODEL

Since there is no known domain model or ontology for project planning that we are aware of, we need to construct one ourselves. The aforementioned PMBOK [12] is our point of departure. PMBOK avoids project management style specific terminology, making it well-suited for our information needs.

#### A. Gathering facts

We have analyzed the whole PMBOK book. This analysis has been focused on the concept of a *project* and everything related to *project planning* therefore we exclude other concepts and processes in the project management domain.

After analyzing 467 pages we have extracted 151 distinct facts related to project planning. A *fact* is either an explicitly defined concept, an implicitly defined concept based on a summarized paragraph, or a relations between concepts. These facts were located on 67 different pages. This illustrates that project planning is a subdomain and that project management as a whole covers many topics that fall outside the scope of the current paper. Each fact was assigned a unique number and the source page number where it was found in PMBOK. Two example facts are: "A milestone is a significant point or event in the project." (id:108, page: 136) and "A milestone may be mandatory or optional." (id:109, page: 136).

#### B. Creating the Reference Model REF

In order to turn these extracted facts into a model for project planning, we have translated the facts to entities, attributes

<sup>3</sup>Implementation details or concepts from other domains.

TABLE I

NUMBER OF ENTITIES AND RELATIONS IN THE CREATED MODELS, AND THE AMOUNT OF LOCATIONS IN THE PMBOK BOOK, SOURCE CODE, OR UI SCREENS USED TO CONSTRUCT THE MODEL.

Source	Model	# entities	# relations			unique observations
			associations	specializations	total	
PMBOK	REF	74	75	32	107	83
Endeavour	USR	23	30	8	38	19
	SRC	26	51	8	59	80
OpenPM	USR	22	24	3	27	13
	SRC	28	44	6	50	68

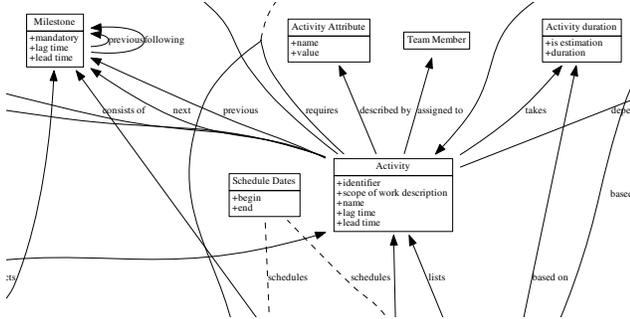


Fig. 3. Fragment of reference model REF visualized as class diagram. Boxes represent entities, arrows relations, and dashed lines link entities to relations.

of entities, and relations between entities. The two example facts (108 and 109), are translated into a relation between the classes Project and Milestone, and the mandatory attribute for the Milestone class. The meta-model of our domain model is a class diagram. We use a textual representation in the meta-programming language Rascal [13] which is also used to perform calculations on these models (precision, recall).

Table I characterizes the size of the project planning reference domain model REF by number of entities, relations and attributes; it contains of 74 entities and 107 relations. There is also a set of 49 attributes, but this seems incomplete, because in general we expect any entity to have more then one property. The lack of details in PMBOK could be an explanation for this. Therefore, we did not use the attributes of the reference model to calculate similarity.

The model is too large to include in this paper, however for demonstration purposes, a small subset is shown in Figure 3.

Not all the facts extracted from PMBOK are used in the Reference Model. Some facts carry only explanations. For example “costs are the monetary resources needed to complete the project”. Some facts explain dynamic relations that are not relevant for an entity/relationship model. These two categories explain 55 of the 68 unused facts. The remaining 13 facts were not clear enough to be used or categorized. In total 83 of the 151 observed facts are represented in the Reference Model.

### C. Discussion

We have created a Reference Model that can be used as oracle for domain model recovery and other related reverse engineering tasks in the project planning domain. The model was created by hand by the second author, and care was taken

to make the whole process traceable. We believe this model can be used for other purposes in this domain as well, such as application comparison and checking feature completeness.

*Threats to validity:* We use PMBOK as main source of information for project planning. There are different approaches to project planning and a potential threat is that some are not covered in this book. Since PMBOK is an industry standard (ANSI and IEEE), we consider this to be a low-risk threat and have not mitigated it.

Another threat is that model recovery by another person could lead to a different model. The traceable extraction of the reference model makes it possible to understand the decisions on which the differences are based. Due to the availability of our analysis scripts, the impact of differences can be easily computed.

## IV. APPLICATION SELECTION

We are interested in finding “ideal” project planning systems to manually read and extract domain models from. The following requirements have guided our search:

- Source code is available: to enable analysis at all.
- No more than 30 KSLOC: to keep manual analysis feasible.
- Uses an explicit data model, for example Model View Controller (MVC), or an Object-relational mapping (ORM): to ensure that domain elements can be identified in the source code.

We have made a shortlist of 10 open source project planning systems<sup>4</sup>. The list contains applications implemented in different languages (Java, Ruby, and C++) and sizes ranging from 18 KSLOC to 473 KSLOC.

From this Endeavour and OpenPM satisfy the aforementioned requirements. Endeavour is a Java application that uses a custom MVC design with ThinWire as front-end framework, and Hibernate as ORM. OpenPM uses Java servlets in combination with custom JavaScript. It also uses Hibernate as ORM. Table II and III describe the structure and size of the two applications<sup>5</sup>. Note that OpenPM’s view package contained MVC controller logic, and the servlets the MVC views.

Both systems aim at supporting the process of planning by storing the process state but they hardly support process enforcement, except recording dependence between activities.

### A. Discussion

A threat to external validity is that both systems are implemented in Java. Looking at systems in multiple modern languages is considered future work.

## V. OBTAINING THE USER MODEL

We have used the UI and documentation of the applications to construct the User Model (USR). Use cases were extracted from the documentation when possible.<sup>6</sup> Following these use cases, a depth-first exploration of the UI is performed. For

<sup>4</sup>ChilliProject, Endeavour, GanttProject, LibrePlan, OpenPM, OpenProj, PLANDora, project.net, taskjuggler, Xplanner+.

<sup>5</sup>Number of files and SLOC are calculated using the cloc tool [14].

<sup>6</sup>Unfortunately, OpenPM does not provide documentation.

TABLE II  
ENDEAVOUR: STRUCTURE AND SIZE.

Package	# files	SLOC	description
model	29	4474	MVC model.
view	108	10480	MVC view (UI).
controller	49	3404	MVC controller.
Total	186	18358	

TABLE III  
OPENPM: STRUCTURE AND SIZE.

Package	# files	SLOC	description
model	29	5591	MVC model.
view	21	1546	MVC controller.
servlets	33	3482	MVC view (UI).
test	75	7137	UI & integration tests.
Total	158	17756	

every entity and relation we have recorded in which UI screen we first observed it. Table I describes the User Models for both Endeavour and OpenPM.

For example the Task entity in Endeavour’s USR Model was based on the sub-window “Task Details“ of the “Home” window.

#### A. Discussion

We have tried to understand the domain knowledge represented by the applications by manually inspecting it from the user’s perspective. Both applications used Ajax to provide an interactive experience.

Endeavour uses the Single Page Application style, with a windowing system similar to MS Windows®. The UI is easy to understand, and different concepts are consistently linked across the application. OpenPM uses a more modern interface. However, we experienced more confusion on how to use it. It assumes a specific project management style (SCRUM), and requires more manual work by the user.

We have observed that creating a User Model is simple. For systems of our size, a single person can construct a User Model in one day. This is considerably less than creating a Source Model and suggests that the UI is an effective source for recovering domain models.

*Threats to validity:* We use the User Model as a proxy for the real domain knowledge exposed by the application. The limit of this knowledge is hard to define, but we believe our approach is an accurate approximation.

We can not be sure about our coverage of the User Model. It could be possible there are other interfaces to the application we are unaware of. Moreover, there could be conditions, triggers, or business rules only observable in very specific scenarios. Some of these issues will be observed in the various model comparisons. We are not aware of other approaches to further increase confidence in our coverage.

## VI. OBTAINING MODELS FROM SOURCE CODE

### A. Domain model recovery

We have chosen the Eclipse Integrated Development Environment (IDE) to read the source code of the selected applications.

Our goal was to maximize the amount of information we could recover. Therefore, we have first read the source code and then used Rascal to analyze relations in the source code. Rascal uses Eclipse’s JDT to analyze Java code, and provides a visualization library that can be used to quickly verify hypothesis formed during the first read-through.

For the actual creation of the model, we have designed and followed these rules:

- Read only the source code, not the database scheme/data.
- Do not run the application.
- Use the terms of the applications, do not translate them to terms used in the Reference Model.
- Include the whole model as seen by the application, do not filter out obvious implementation entities.
- Do read comments and string literals.

We have used the same meta-model as used for describing the Reference Model. We replaced the fact’s identifiers with source locations (filename and character range), which are a native construct in Rascal. To support the process of collecting facts from the source code we added a menu-item to the context-menu of the Java editor to write the cursor’s source location to the clipboard.

The domain model for each application was created in a similar fashion as we did when creating the reference model. All the elements in the domain model are based on one or more specific observations in the source code (see table I).

For example the relation between Task and Dependency in Endeavour’s SRC model is based on the `List<Dependency>` dependencies field found on line 35 in file `Endeavour-Mgmt/-model/org/endeavour/mgmt/model/Task.java`.

### B. Results

Table I shows the sizes of the extracted models for both applications expressed in number of entities, relations and attributes and the number of unique source code locations where they were found.

1) *Endeavour:* In Endeavour 26 files contributed to the domain model. 22 of those files were in the model package, the other 4 were from the controller package. The controller classes each contributed one fact. 155 of the source locations were from the model package.

2) *OpenPM:* In OpenPM 22 files contributed to the domain model. These files were all located in the model package.

### C. Discussion

We have performed domain model recovery on two open source software applications for project planning.

Both applications use the same ORM system, but a different version of the API. Endeavour also contains a separate view model, which is used in the MVC user interface. However, it has been implemented as a pass-through layer for the real model.

*Threats to validity:* A first threat (to internal validity) is that manual analysis is always subject to bias from the performer and that this was performed by the same author who created the other models. We have mitigated this by maximizing the traceability of our analysis: we have followed a fixed analysis

TABLE IV  
CATEGORIES FOR SUCCESSFULLY MAPPED ENTITIES

Mapping name	Description
Equal Name	Entity has the same name as an entity in the other model. Note that this is the only category which can also be a failure when the same name is used for semantically different entities
Synonym	Entity is a direct synonym for an entity in the other model, and is it not a homonym.
Extension	Entity captures a wider concept than the same entity in the other model.
Specialization	Entity is a specific or concrete instance of the same entity in the other model.
Implementation specialization	Comparable to specialization but the specialization is related to an implementation choice.

TABLE V  
CATEGORIES FOR UNSUCCESSFULLY MAPPED ENTITIES

Mapping name	Description
Missing	The domain entity is missing in the other model, i.e. a false positive. This is the default mapping failure when an entity cannot be mapped via any of the other categories.
Implementation	The entity is an implementation detail and is not a real domain model entity.
Too detailed	An entity is a domain entity but is too detailed in comparison with the other model.
Domain detail	The entity is a detail of a sub domain, this category is a subclass of “too detailed”.

process and have performed multiple analysis passes over the source code and published the data.

A second threat (to external validity) is the limited size of the analyzed applications, both contain less than 20 KSLOC Java. Larger applications would make our conclusions more interesting and general, but they would also make the manual analysis less feasible.

## VII. MAPPING MODELS

We now have five domain models of project planning: one reference model (REF) to be used as oracle, and four domain models (SRC, USR) obtained from the two selected project planning applications. These models use different vocabulary, we have to map them onto the same vocabulary to be able to compare them.

### A. Lightweight domain model mapping

We manually map the entities between different comparable models. The question is how to decide whether to entities are the same. Strict string equality is too limited and should be relaxed to some extent.

Table IV and V show the mapping categories we have identified for the (un)successful mapping of model entities.

### B. Mapping results

We have manually mapped all the entities in the User Model (USR) and the Source Model (SRC) to the Reference Model

TABLE VI  
ENDEAVOUR: ENTITIES IN THE MAPPED MODELS, PER MAPPING CATEGORY

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	7	7	7	7	21	21
Synonym	2	3	2	3	3	2
Extension	0	0	0	0	0	0
Specialization	5	3	5	3	0	0
Implementation specialization	1	1	1	1	0	0
Total	15	14	15	14	24	23
Equal Name†	1	-	1	-	0	-
Missing	1	-	2	-	0	-
Implementation	1	-	2	-	2	-
Domain Detail	5	-	6	-	0	-
Too Detailed	0	-	0	-	0	-
Total	8	-	11	-	2	-

† A false positive, in Endeavour the term Document means something different then the term Documentation in the Reference Model.

TABLE VII  
OPENPM: ENTITIES IN MAPPED MODELS, PER MAPPING CATEGORY

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	1	1	1	1	18	18
Synonym	3	3	4	4	4	4
Extension	1	1	1	1	0	0
Specialization	0	0	0	0	0	0
Implementation specialization	1	1	1	1	0	0
Total	6	6	7	7	22	22
Missing	2	-	2	-	1	-
Implementation	12	-	17	-	5	-
Domain Detail	0	-	0	-	0	-
Too Detailed	2	-	2	-	0	-
Total	16	-	21	-	6	-

(REF), and SRC to USR. For each mapping we have explicitly documented the reason for choosing this mapping. For example, in Endeavour’s SRC model the entity *Iteration* is mapped to *Milestone* in the Reference Model using specialization, with documented reason: “*Iterations split the project into chunks of work, Milestones do the same but are not necessarily iterative.*”

Table VI and VII contain the number of mapping categories used for both applications, per mapping. For some mapping categories, it is possible for one entity to map to multiple, or multiple entities to one. For example the *Task* and *WorkProduct* entities in Endeavour’s SRC model are mapped on the *Activity* entity in the Reference Model. Therefore, we report the numbers of the entities in both the models, the source and the target.

The relatively large number of identically named entities (7/15) between Endeavour and the reference model is due to the presence of a similar structure of five entities, describing all the possible activity dependencies.

An example of a failed mapping is the *ObjectVersion* entity in the Source Model of OpenPM. This entity is an implementation detail. It is a variant of the Temporal Object pattern<sup>7</sup> where

<sup>7</sup>See <http://martinfowler.com/eaDev/TemporalObject.html>.

TABLE VIII  
ENTITIES FOUND IN THE VARIOUS DOMAIN MODELS.

Source	Model	Entities <sup>†</sup>
PMBOK	REF	Action, <b>Activity</b> , <b>Activity Attribute</b> , <b>Activity Dependency</b> , <b>Activity duration</b> , Activity list, Activity resource, Activity sequence, Activity template, Approver, Budget, Change Control Board, <b>Change request</b> , Closing, Communications plan, Composite resource calendar, Composite resource calendar availability, Constrain, Corrective action, <b>Defect</b> , Defect repair, <b>Deliverable</b> , <b>Documentation</b> , Environment, Equipment, External, <b>FinishFinish</b> , <b>FinishStart</b> , Human Resource Plan, Information, Internal, Life cycle, Main, Material, <b>Milestone</b> , Objective, Organisation, Organizing, People, Person, Phase, Planned work, Portfolio, Preparing, Preventive action, Process, Product, <b>Project</b> , Project management, Project plan, <b>Project schedule</b> , Project schedule network diagram, Quality, <b>Requirement</b> , Resource, Resource calendar, Resource calendar availability, Result, Risk, Risk management plan, Schedule, Schedule Dates, Schedule baseline, Schedule data, Scope, Service, Stakeholder, <b>StartFinish</b> , <b>StartStart</b> , Supplies, <b>Team Member</b> , Work Breakdown Structure, Work Breakdown Structure Component, Work Package
Endeavour	USR	<b>Actor</b> , <b>Attachment</b> , <b>Change Request</b> , Comment, <b>Defect</b> , <b>Document</b> , Event, <b>FinishFinish</b> , <b>FinishStart</b> , Glossary, <b>Iteration</b> , <b>Project</b> , <b>ProjectMember/Stakeholder</b> , Security Group, <b>StartFinish</b> , <b>StartStart</b> , <b>Task</b> , <b>Task Dependency</b> , Test Case, Test Folder, Test Plan, Use Case, X
Endeavour	SRC	<b>Actor</b> , <b>Attachment</b> , <b>ChangeRequest</b> , Comment, <b>Defect</b> , <b>Dependency</b> , <b>Document</b> , Event, <b>FinishFinish</b> , <b>FinishStart</b> , Glossary-Term, <b>Iteration</b> , Privilege, <b>Project</b> , <b>ProjectMember</b> , Security-Group, <b>StartFinish</b> , <b>StartStart</b> , <b>Task</b> , TestCase, TestFolder, TestPlan, TestRun, <b>UseCase</b> , Version, <b>WorkProduct</b>
OpenPM	USR	Access Right, <b>Attachment</b> , Button, Comment, Create, Delete, <b>Effort</b> , Email Notification, FieldHistory, HistoryEvent, <b>Iteration</b> , Label, Link, ObjectHistory, <b>Product</b> , Splitter, State, Tab, <b>Task</b> , Type, Update, <b>User</b>
OpenPM	SRC	Access, Add, <b>Attachment</b> , Comment, Create, Delete, <b>Effort</b> , Email-Subscription, EmailSubscriptionType, Event, FieldType, FieldVersion, Label, Link, <b>Milestone</b> , ObjectType, ObjectVersion, <b>Product</b> , Remove, Splitter, <b>Sprint</b> , Tab, <b>Task</b> , TaskButton, TaskState, Task-Type, Update, <b>User</b>

<sup>†</sup> Bold entity in Reference Model is used in application models. Bold entity in application model could be mapped to entity in Reference Model.

every change of an entity is stored to explicitly model the history of all the objects in the application.

Table VIII contains all the entities per domain model, and highlights the mapped entities.

### C. Discussion

We have used a lightweight approach for mapping domain models. Our mapping categories may be relevant for other projects and can be further extended and evaluated.

For future work, we can investigate if whether more automated natural language processing can help, however, remember our motivations for excluding automatic approaches in our current research method.

At most half of the domain models recovered from the applications could be mapped to the reference model. The other half of the extracted models regarded details of the domain or the implementation.

*Threats to validity:* A threat to external validity is that we have used an informal approach to map the domain models of the two applications to the reference model. The mapping categories presented above, turned out to be sufficient for these two applications, however we have no guarantees for other application of these categories. The categories have evolved during the process and each time a category was added or modified all previous classifications have been reconsidered.

## VIII. COMPARING THE MODELS

We now have five manually constructed and six derived domain models for project planning:

- One reference model (REF) to be used as oracle.
- Four domain models (SRC, USR) obtained from each of the two selected project planning applications.
- Six derived domain models (OBS, REC, INT) resulting from the mapping of the previous four (SRC, USR).

How can we compare these models in a meaningful way?

### A. Recall and Precision

The most common measures to compare the results of an IR technique are *recall* and *precision*. Often it is not possible to get the 100% in both, and we have to discuss which measure is more important in the case of our model comparisons.

We have more than two datasets, and depending on the combination of datasets, recall or precision is more important. Table IX explains in detail how recall and precision will be used and explains for the relevant model combinations which measure is useful and what will be measured.

Given two models  $M_1$  and  $M_2$ , we use the following notation. The comparison of two models is denoted by  $M_1 \diamond M_2$  and results in recall and precision for the two models. If needed,  $M_1$  is first mapped to  $M_2$  as described in Tables VI and VII.

### B. Results

Tables X and XI shows the results for, respectively, Endeavour and OpenPM. Which measures are calculated is based on the analysis in Table IX.

### C. Relation Similarity

Since recall and precision for sets of entities provides no insight into similarity of the relations between entities, we need an additional measure. Our domain models contain entities and their relations. Entities represent the concepts of the domain, and relations their structure. If we consider the relations as a set of edges, we can directly calculate recall and precision in a similar fashion as described above.

We also considered some more fine grained metrics for structural similarity. Our domain model is equivalent to a subset of Unified Modeling Language (UML) class diagrams and several approaches exist for calculating the minimal difference between such diagrams [15], [16]. Such “edit distance” methods give precise indications of how big the difference is. Similarly we might use general graph distance metrics [17]. We tried this latter method and found that the results, however more sophisticated, were harder to interpret. For example, USR and REF were 11% similar for Endeavor. This seems to be in line with the recall numbers, 6% for relations and 19% for entities, but the interesting precision results (64% and 15%) are lost in this metric. So we decided not to report these results and stay with the standard accuracy analysis.

TABLE IX  
RECALL AND PRECISION EXPLAINED PER MODEL COMBINATION.

Retrieved	Expected	Recall	Precision
USR	REF	Which part of the domain is covered by an application. This is subjective to the size of REF.	How many of the concepts in USR are actually domain concepts, e.g., how much implementation details are in the <i>application</i> ?
SRC	REF	How much of REF can be recovered from SRC. If high then this should confirm high recall for both USR $\diamond$ REF and SRC $\diamond$ USR.	How much of SRC are actually domain concepts, e.g., how much implementation junk is accidentally recovered from <i>source</i> ?
SRC	USR	How much of USR can be recovered by analyzing the source code (SRC). This gives no measure of the amount of actual domain concepts found.	How many details are in SRC, but not in USR? If USR were a perfect representation of the application knowledge, this category would only contain dead-code and unexposed domain knowledge.

TABLE X  
ENDEAVOUR: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR $\diamond$ REF	19%	6%	64%	15%
SRC $\diamond$ REF	19%	6%	56%	13%
SRC $\diamond$ USR	100%	92%	92%	74%

TABLE XI  
OPENPM: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR $\diamond$ REF	7%	3%	23%	16%
SRC $\diamond$ REF	9%	6%	25%	18%
SRC $\diamond$ USR	100%	80%	79%	44%

TABLE XII  
COMBINED: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR $\diamond$ REF	22%	7%	40%	14%
SRC $\diamond$ REF	23%	9%	36%	13%

#### D. Discussion

1) *Low precision and recall for relations*: On the whole the results for the precision and recall of the relation part of the models are lower than the quality of the entity mappings. We investigated this by taking a number of samples. The reason is that the Reference model is more detailed, introducing intermediate entities with associated relations. For every intermediate entity, two or more relations are introduced which can not be found in the recovered models.

These results indicate that the recall and precision metrics for sets of relations underestimate the structural similarity of the models.

2) *Precision of OBS: USR  $\diamond$  REF*: We found the precision of OBS to be 64% (Endeavour) and 23% (OpenPM), indicating that both applications contain a significant amount of entities that are unrelated to project planning as delimited by the Reference Model. For Endeavour, out of the 8 unmappable entities (see Table VI in section VII), only 2 were actual implementation details. The other 6 are sub-domain details not globally shared within the domain. If we recalculate to correct for this, Endeavour's Observed Model even has a precision of 91%. For OpenPM there are only 2 out of the 16 for which this correction can be applied, leaving the precision at 36%. For the best scenario, in this case represented by Endeavour, 90% of the User Model (USR) is part of the Reference Model (REF).

OpenPM's relatively low precision (36%) can be explained by table VII, which show the USR model has a lot of

implementation detail (related to version control operations).

3) *Recall of OBS: USR  $\diamond$  REF*: The recall for the Observed Model (OBS) is for Endeavour 19% and for OpenPM 7%. Which means both applications cover less than 20% of the project planning domain.

4) *Precision of REC: SRC  $\diamond$  REF*: The precision of the Recovered Model (REC) is for Endeavour 56% (corrected 88%), and for OpenPM 25% (corrected 39%). This shows that for the best scenario, represented again by Endeavour, the Source Model only contains 12% implementation details.

5) *Recall of REC: SRC  $\diamond$  REF*: The recall for the Recovered Model (REC) is for Endeavour 19% and for OpenPM 9%. The higher recall for OpenPM, compared to OBS, for both entities and relations is an example where the Source Model contained more information than the User Model, which we will discuss in the next paragraph.

6) *Precision and recall for INT: SRC  $\diamond$  USR*: How much of the User Model can be recovered by analyzing only the Source Model? For both Endeavour and OpenPM, recall is 100%. This means that every entity in the USR model was found in the source code. Endeavour's precision was 92% and OpenPM's 79%. OpenPM contains an example where information in the Source Model is not observable in the User Model: comments in the source code explain the *Milestones* and their relation to *Iterations*.

The 100% recall and high precision mean that these applications were indeed amenable for reverse engineering (as we hypothesized when selecting these applications). We could extract most of the information from the source code.

For this comparison, even the relations score quite high. This indicates that User Model and Source Model are structurally similar. Manual inspection of the models confirms this.

7) *Recall for Endeavour and OpenPM combined*: Endeavour's and OpenPM's recall of USR  $\diamond$  REF and SRC  $\diamond$  REF measure the coverage of the domain a re-engineer can achieve. How much will the recall improve if we combine the recovered models of the two systems?

We only have two small systems, however, Table XII contains the recall and precision for Endeavour and OpenPM combined. A small increase in recall, from 19% to 23%, indicates that there is a possibility for increasing the recall by observing more systems. However, as expected, at the cost of precision.

8) *Interpretation*: Since our models are relatively small, our results cannot be statistically significant but are only indicative. Therefore we should not report exact percentages, but characterizing our recall and precision as *high* seems valid. Further research based on more applications is needed to confirm our results.

## IX. RELATED WORK

There are many connections between ontologies and domain models. The model mappings that we need are more specific than the ones provided by general ontology mapping [18].

Abebe and Tonella [9] introduced a natural language parsing (NLP) method for extracting an ontology from source code. They came to the same conclusion as we do: this extracted ontology contains a lot of implementation details. Therefore, they introduced an IR filtering method [10] but it was not as effective as the authors expected. Manual filtering of the IR keyword database was shown to improve effectiveness. Their work is in the same line as ours, but we have a larger reference domain model, and we focus on finding the limits of domain model recovery, not on an automatic approach. It would be interesting to apply their IR filtering to our extracted models.

Ratiu et al. [11] proposed an approach for domain ontology extraction. Using a set of translation rules they extract domain knowledge from the API of a set of related software libraries. Again, our focus is on finding the limits of model recovery, not on automating the extraction.

Hsi et al. [19] introduced ontology excavation. Their methodology consists of a manual depth-first modeling of all UI interactions, and then manually creating an ontology, filtering out non-domain concepts. They use five graph metrics to identify interesting concepts and clusters in this domain ontology. We are interested in finding the domain model inside the user-interface model, Hsi et al. perform this filtering manually, and then look at the remaining model. Automatic feature extraction of user interfaces is described in [20].

Carey and Gannod [6] introduced a method for concept identification. Classes are considered the lowest level of information of an object-oriented system and Machine Learning is used in combination with a set of class metrics. This determines interesting classes, which should relate to domain concepts. Our work is similar, but we focus on *all* the information in the source code, and are interested in the maximum that can be recovered from the source. It could be interesting to use our reference model to measure how accurately their approach removes implementation concerns.

UML class diagram recovery [21], [22] is also related to our work but has a different focus. Research focuses on the precision of the recovered class diagrams, for example the difference between a composition and aggregation relation. We are interested in less precise UML class diagrams.

Work on recovering the concepts, or topics, of a software system [4], [5] has a similar goal as ours. IR techniques are used to analyze all the terms in the source code of a software system, and find relations or clusters. Kuhn et al. [5] use identifiers in source code to extract semantic meaning and report on the difficulty of evaluating their results. Our work focuses less on structure and grouping of concepts and we evaluate our results using a constructed reference model.

Reverse engineering the relation between concepts or features [8], [23], assumes that there is a set of known features or concepts and tries to recover the relations between them. These approaches are related to our work since the second half of our problem is similar: after we have recovered domain entities, we need to understand their relations.

DeBaud et al. [24] report on a domain model recovery case study on a COBOL program. By manual inspection of the source code, a developer reconstructed the data constructs of the program. They also report that implementation details make extraction difficult, and remark that systems often implement multiple domains, and that the implementation language plays an important role in the discovery of meaning in source code.

We do not further discuss other related work on knowledge recovery that aims at extracting facts about architecture or implementation. One general observation in all the cited work is that it is hard to separate domain knowledge from implementation knowledge.

## X. CONCLUSIONS

We have explored the limits of domain model recovery via a case study in the project planning domain. Here are our results and conclusions.

### A. Reference model

Starting with PMBOK as authoritative domain reference we have manually constructed an actionable domain model for project planning. This model is openly available and may be used for other reverse engineering research projects.

### B. Lightweight model mapping

Before we can understand the differences between models, we have to make them comparable by mapping them to a common model. We have created a manual mapping method that determines for each entity if and how it maps onto the target model. The mapping categories evolved while creating the mappings. We have used this approach to describe six useful mappings, four to the Reference Model and two to the User Model.

### C. What are the limits of domain model recovery?

We have formulated two research questions to get insight in the limits of domain model recovery. Here are the answers we have found (also see Table IX and remember our earlier comments on the interpretation of the percentages given below).

*Q1: Which parts of the domain are implemented by the application?* Using the user view (USR) as a representation of the part of the domain that is implemented by an application, we have created two domain models for each of the two selected

applications. These domain models represent the domain as exposed by the application. Using our Reference Model (REF) we were able to determine which part of USR was related to project planning. For our two cases 91% and 36% of the User Model (USR) can be mapped to the Reference Model (REF). This means 9% and 64% of the UI is about topics not related to the domain. From the user perspective we could determine that the applications implement 19% and 7% of the domain.

The tight relation between the USR and the SRC model (100% recall) shows us that this information is indeed explicit and recoverable from the source code. Interestingly, some domain concepts were found in the source code that were hidden by the UI and the documentation, since for OpenPM the recall between USR and REF was 7% where it was 9% between SRC and REF.

So, the answer for Q1 is: the recovered models from source code are useful, and only a small part of the domain is implemented by these tools (only 7-19%).

*Q2: Can we recover those implemented parts from the source of the application?* Yes, see the answer to Q1. The high recall between USR and SRC shows that the source code of these two applications explicitly models parts of the domain. The high precisions (92% and 79%) also show that it was feasible to filter implementation junk manually from these applications from the domain model.

#### D. Perspective

For this research we manually recovered domain models from source code to understand how much valuable domain knowledge is present in source code. We have identified several follow-up questions:

- How does the quality of extracted models grow with the size and number of applications studied? (Table XII)
- How can differences and commonalities between applications in the same domain be mined to understand the domain better?
- How does the quality of extracted models differ between different domains, different architecture/designs, different domain engineers?
- How can the extraction of a User Model help domain model recovery in general. Although we have not formally measured the effort for model extraction, we have noticed that extracting a User Model requires much less effort than extracting a Source Model.
- How do our manually extracted models compare with automatically inferred models?
- What tool support is possible for (semi-)automatic model extraction?
- How can domain models guide the design of a DSL?

Our results of manually extracting domain models are encouraging. They suggest that when re-engineering a family of object-oriented applications to a DSL their source code is a valuable and trustworthy source of domain knowledge, even if they only implement a small part of the domain.

#### REFERENCES

- [1] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages." *ACM Comput. Surv.*, no. 37, pp. 316–344, 2005.
- [2] T. Biggerstaff, "Design recovery for maintenance and reuse," *Computer*, vol. 22, no. 7, pp. 36–49, Jul. 1989.
- [3] T. J. Biggerstaff, B. G. Mitbender, and D. Webster, "The concept assignment problem in program understanding," in *Proc. 15th international conference on Software Engineering*, ser. ICSE '93. IEEE Computer Society Press, May 1993, pp. 482–498.
- [4] E. Linstead, P. Rigor, S. K. Bajracharya, C. V. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 461–464.
- [5] A. Kuhn, S. Ducasse, and T. Girba, "Semantic clustering: Identifying topics in source code," *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.
- [6] M. M. Carey and G. C. Gannod, "Recovering Concepts from Source Code with Automated Concept Identification," in *15th International Conference on Program Comprehension*, 2007, pp. 27–36.
- [7] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension," in *10th International Workshop on Program Comprehension*, 2002, pp. 271–280.
- [8] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 210–224, 2003.
- [9] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *18th IEEE International Conference on Program Comprehension*, 2010, pp. 156–159.
- [10] —, "Towards the Extraction of Domain Concepts from the Identifiers," in *18th Working Conference on Reverse Engineering*, 2011, pp. 77–86.
- [11] D. Ratiu, M. Feilkas, and J. Jürgens, "Extracting domain ontologies from domain specific APIs," in *12th European Conference on Software Maintenance and Reengineering*, vol. 1, 2008, pp. 203–212.
- [12] P. M. Institute, Ed., *A Guide to the Project Management Body of Knowledge*, 4th ed. Project Management Institute, 2008.
- [13] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proc. 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, September 2009, pp. 168–177.
- [14] "Count Lines of Code Tool," <http://cloc.sourceforge.net>.
- [15] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models," in *Software Engineering 2005*, ser. LNI, P. Liggesmeyer, K. Pohl, and M. Goedicke, Eds., vol. 64, 2005, pp. 105–116.
- [16] D. Ohst, M. Welle, and U. Kelter, "Differences between versions of UML diagrams," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 227–236, Sep. 2003. [Online]. Available: <http://doi.acm.org/10.1145/949952.940102>
- [17] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognition Letters*, vol. 19, no. 3-4, pp. 255–259, 1998.
- [18] N. Choi, I.-Y. Song, and H. Han, "A survey on ontology mapping," *SIGMOD Rec.*, vol. 35, no. 3, pp. 34–41, Sep. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1168092.1168097>
- [19] I. Hsi, C. Potts, and M. M. Moore, "Ontological Excavation: Unearthing the core concepts of the application," in *10th Working Conference on Reverse Engineering*, 2003, pp. 345–352.
- [20] M. Baciková and J. Porubán, "Analyzing stereotypes of creating graphical user interfaces," *Central Europ. J. Computer Science*, vol. 2, no. 3, pp. 300–315, 2012.
- [21] K. Wang and W. Shen, "Improving the Accuracy of UML Class Model Recovery," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, 2007, pp. 387–390.
- [22] A. Sutton and J. I. Maletic, "Mappings for Accurately Reverse Engineering UML Class Models from C++," in *12th Working Conference on Reverse Engineering*, 2005, pp. 175–184.
- [23] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proc. 33rd International Conference on Software Engineering*, 2011, pp. 461–470.
- [24] J.-M. DeBaud, B. Moopen, and S. Rugaber, "Domain Analysis and Reverse Engineering," in *Proc. the International Conference on Software Maintenance*, 1994, pp. 326–335.

# Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams

Yvan Labiche

Carleton University, Department SCE  
Ottawa, Canada  
labiche@sce.carleton.ca

Bojana Kolbah

Carleton University, Department SCE  
Ottawa, Canada  
kolbah@gmail.com

Hossein Mehrfard

Carleton University, Department SCE  
Ottawa, Canada  
mehrfard@sce.carleton.ca

**Abstract**—This paper discusses a step towards reverse engineering source code to produce UML sequence diagrams, with the aim to aid program comprehension and other activities (e.g., verification). Specifically, our objective being to obtain a lightweight instrumentation and therefore disturb the software behaviour as little as possible in order to eventually produce accurate sequence diagrams. To achieve this, we combine static and dynamic analyses of a Java software, reducing information we collect at runtime (lightweight instrumentation) and compensating for the reduced runtime information with information obtained statically from source code. Static and dynamic information are represented as models and UML diagram generation becomes a model transformation problem. Our validation against a previous, correct approach shows that we indeed reduce the execution overhead inherent to dynamic analysis, while still producing useful diagrams.

**Keywords**—Reverse-engineering; Sequence diagram; Scenario diagram; Static analysis; Dynamic analysis

## I. INTRODUCTION

To fully understand a software, information regarding its structure and behavior is required. When no complete information is available, one has to reverse engineer it through static and dynamic analyses. Besides helping comprehension, reverse engineered information can help quality assurance [22, 34]. While there are CASE tools (e.g., Topcased, RSA, Together) and techniques (e.g., [10]) to reverse engineer structure, we focus on reverse engineering behavior.

Following the current trend<sup>1</sup> on hybrid (i.e., static plus dynamic) analysis, we combine execution trace information with control flow information to generate UML sequence diagrams [25]. Compared to previous, purely dynamic techniques, our objective is to reduce instrumentation as much as possible, to disturb behaviour as little as possible (e.g., limiting the risk of inaccuracies in the reverse-engineered information), and compensate for the missing (dynamic) information by collecting static information. Our experiments for instance show that an instrumented software can be two times slower than its non-instrumented version. Although we do not experiment with real-time systems in this paper, disturbing behaviour as little as possible will ensure that object

interactions resulting from thread communications will be correctly reverse-engineered.

More accurately, we reverse engineer one scenario at a time from one execution trace, and we render it using the UML sequence diagram notation. Several scenario diagrams should then be merged into a complete sequence diagram for a given use case. This requires triggering as many varied scenarios as possible through multiple executions of the system (e.g., using black-box testing techniques), and merging them into one sequence diagram. This merging is left however to future work. In this paper, we refer to the generated diagrams as UML scenario diagrams since they show only one scenario at a time. The reader familiar with the UML will argue that this is not standard UML terminology as there is no notion of scenario diagram in UML. However, we decided to use this terminology to not raise false expectations (i.e., we do not merge scenarios). Also, as we focus on reducing instrumentation (impact), we do not discuss the rendering of (possibly very large) traces into UML scenario (or sequence) diagrams. We however ensure we can generate accurate scenario diagrams on small traces. Combining our lightweight instrumentation with techniques to handle large traces (e.g., [8, 11, 12, 19, 27]) is left for future work.

To formalize our approach and specify it from a logical standpoint so it can be analyzed in and compared with future works, we define two models (class diagrams): one for traces and another for control flow graphs; and define mapping rules between them using the OCL [25]. These rules are used as specifications to implement a tool to instrument code so as to generate traces, to analyze source code to create control flow graphs, and then transform (model transformation) an instance of the trace model and instances of control flow graphs (for several methods) into a UML scenario diagram.

The main contributions of this paper are: (1) a hybrid technique combining static and dynamic data for reverse-engineering behaviour with the intent to reduce execution overhead; (2) a precise modeling of the approach (with models and OCL mapping rules), allowing model transformation; (3) the reverse engineering of alternative and iterative executions, including test conditions; (4) case studies, including industry size software, showing reduced instrumentation and execution overhead while providing accurate information.

<sup>1</sup> Half of the articles examined in a 2009 survey on program comprehension through dynamic analyses also employ static information [7].

We discuss related work in section II and present our approach in sections III to V. We report on case studies in section VI. Conclusions are provided in section VII.

## II. RELATED WORK

The area of program comprehension through dynamic analysis is varied and vibrant as a 2009 systematic survey suggests [7]. The authors systematically analyzed 176 papers (out of 4,795 initially selected) published between July 1999 and June 2008 that rely on dynamic analysis to conduct program comprehension activities. Nineteen of those papers<sup>2</sup> use some kind of dynamic analysis (e.g., debugger, source code instrumentation) to reverse engineer object collaborations, rendered under the form of a UML sequence diagram (or a similar diagram). We focus on those 19 papers as they directly relate to our work. These works collect execution information, specifically constructor, static/non static method calls (or executions), to produce scenario diagrams. While some of those approaches use both static and dynamic analyses, none of them actually combines both types of analysis to produce dynamic models: the static analysis is only used to generate structural diagrams (e.g., class diagram) and the dynamic analysis is only used to generate object collaborations. In some rare cases, the static analysis is used to guide the user in selecting elements of the source code to monitor during the dynamic analysis (e.g., [17]). The majority of those works do not reverse engineer information on alternative executions (i.e., control flow) and generated diagrams do not therefore indicate under which conditions or repetitions objects send messages. Only two works [6, 30] are closely related to ours in terms of the generated diagrams, although they are both only dynamic. The closest to our work instruments the source code using aspects and collects method executions and control flow information [5, 6], while the other relies on break points (for method and control statements) being set with a debugger [30]. Other works indicate repetitions in generated diagrams. However, they either use a simplistic heuristic to identify repetitions [12] (specifically, contiguous repeated messages are collapsed into repetitions, which does not produce an accurate diagram in general) or recognize occurrences of known interaction patterns that must be provided by the user [15, 16].

Since the 2009 systematic survey, additional related work has been published. Once again, we focus on reverse engineering object collaborations through dynamic and/or static analysis, discussing whether the techniques rely only on a dynamic analysis, a static analysis or both. (Other characteristics of the techniques are interesting, but they are less relevant to this paper, and are therefore not discussed here.) Some approaches attempt to generate sequence diagrams using a purely static analysis of the source code [13, 28, 29, 32], while others rely on execution traces, though through dynamic analysis only and without (necessarily) recovering alternatives or loops [2, 14, 18, 21, 26, 31, 33, 35]. Similarly to [17], static analysis is sometimes suggested, but not actually used, as a way to guide instrumentation [21, 31]. One purely dynamic technique [1] recognizes loops and alternatives when reverse engineering network communications. It complements

ours: they look at the boundaries of interacting networked applications while we look at the inside of the interacting applications. Another purely dynamic technique [35] identifies loops and alternatives. They focus on how multiple execution traces can be merged to generate sequence diagrams whereas we focus on how such traces can be obtained without disturbing too much the observed behaviour.

One recent hybrid technique [19] is to collect the same information as we do in this paper (sections III to V), specifically line number and signature of invocations (static information) and invocations objects make to one another (dynamic information). It however relies on debug and source code analysis to collect static information, while we only rely on source code analysis. Additionally, it has a different objective than ours: showing how static and dynamic information allow a tool to recognize loops and to report them in a condensed way in order to better visualize object interactions (note the authors do not report on the execution overhead caused by their instrumentation strategy); instead, we are interested in studying how combining both techniques reduces instrumentation overhead, while also recognizing control flows. The two pieces of work are complementary.

Dynamic and hybrid techniques are also used to compact traces or sequence diagrams generated from them [8, 11, 12, 19, 27], assuming traces (or scenario diagrams) already exist. The objective is for instance to recognize repeated sequences of calls/messages and therefore loops. Instead, we work on the generation of such traces, attempting to minimize the instrumentation overhead. These works are therefore complementary to ours. Studying to what extent they can be combined is part of our future work. Other researchers suggest ways to dig into large sequence diagrams [3, 19, 23].

Many tools are capable of reverse engineering sequence diagrams. (We omit tools that only reverse-engineer the class diagram, such as Topcased, Poseidon, ModelMaker, Together, or MoDisco.) They either rely on a purely static analysis of the source code (e.g., MagicDraw, RSA), or trace method executions/calls without collecting control flow information (e.g., MaintainJ, reverseJava, JSonde, javaCallTracer, J2U, TPTP's UML2 trace interaction View, CodeLogic). Note that Fujaba<sup>3</sup> and related projects do not reverse-engineer sequence diagrams. Some Fujaba projects do manipulate traces though, but for the purpose of detecting design patterns. With respect to tool support for reverse-engineering sequence diagrams, some authors discuss the right features such a tool should provide, especially when dealing with large traces/diagrams [3].

To conclude, no sequence diagram reverse engineering technique that we are aware of specifically addresses the issue of reducing the amount of collected runtime information and compensating this lack of information with a static analysis, with the attempt to limit the probe effect while still being able to show control flow information in sequence (scenario) diagrams. To the best of our knowledge, the hybrid approach we present in this paper is therefore unique. Note however that our dynamic analysis, whereby we trace method calls, is not unique: in fact this appears to be the most widely used trace

<sup>2</sup> References number 19, 21, 22, 23, 27, 29, 30, 33, 40, 90, 103, 116, 121, 123, 126, 129, 135, 141, 147 in [7].

<sup>3</sup> <http://www.fujaba.de/>

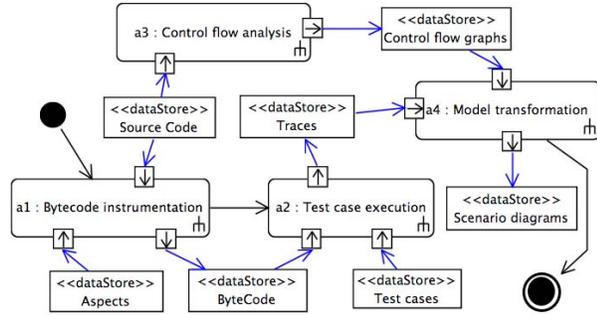


Fig.1. Proposed approach (UML activity diagram)

collecting technique. What is unique is our combination of static and dynamic analyses.

Note that since we are interested in evaluating how the instrumentation overhead can be reduced by combining static and dynamic information (as opposed to only relying on dynamic information), we will compare our new approach with our old solution [6]. We defer the comparison of different hybrid techniques (e.g., [19]) to future work.

### III. OVERALL DESCRIPTION OF THE APPROACH

Our approach is summarized as a UML activity diagram in Fig. 1. We attempt to minimize instrumentation, using aspects (activity *a1*), and execute the instrumented version of the software using test cases (activity *a2*). Offline, we reverse-engineer the control flow graph of methods (activity *a3*). We then combine the trace and control flow information in a model transformation activity (*a4*) to generate scenario diagrams. We created tool support to automate activities *a1*, *a3* and *a4*. Activity *a2* can be automated, for instance using a framework like JUnit.

Our past work [6] has a similar process, the main difference being the absence of activity *a3* (and the generated graphs). As a consequence, activity *a4* is different. The two approaches are equally easy to setup and use since the same activities are automated and the aspects are generic (i.e., can be automatically tailored to any case study software).

The remainder of the paper discusses the control flow and trace information, i.e., the models for <<dataStore>> Control flow graphs and Traces in Fig. 1 (section IV), and our tool to automate activities *a1*, *a3*, and *a4* (section V).

The main issues that drove the design of the trace model and the control flow model are:

- Keeping instrumentation to a minimum required for collecting the necessary information. We only collect method calls: caller and callee objects, method signature, class name and line number of call;
- Uniquely identifying object with a pair (class name, instance counter for that class), similarly to [6];
- Collecting the right information from the source code to match trace information, specifically line number of method calls (including complete signature), and to identify control flow structures;

- Devising models to facilitate model transformations: i.e., the trace model is close to the UML 2 metamodel.

### IV. CONTROL FLOW AND TRACE INFORMATION

We refer the reader to [24] for the UML 2 metamodel and only discuss our trace model (section IV.A) and control flow model (section IV.B).

#### A. Trace model

The trace model (Fig. 2) models execution trace data and is very close in structure to the UML 2 Superstructure's Message components: elements Log, MessageLog, MessageLogOccurrenceSpecification and MessageSort map to the UML's Interaction, Message, MessageOccurrenceSpecification and MessageSort respectively.

Log represents a single program execution and contains a sequence of MessageLogs. A MessageLog represents a message sent to the logger to signal the start of an execution between a sending object and a receiving object (the two associations to MessageLogOccurrenceSpecification). In class MessageLogOccurrenceSpecification, attribute covered is a String containing the identification of an object (a unique identifier representing an object of a class), to be eventually translated into a lifeline in the sequence diagram. MessageLog's attributes specify the kind of the message (messageSort attribute), i.e., a synchronous call or an object creation (the message's signature<sup>4</sup> (in the form returnType package.class.calledMethodName (formal parameter types), i.e., the signature of the method being called), and the name of the class whose instance executes the called method (bindToClass). For a MessageLog instance, using bindToClass and signature attribute values, we know exactly which method in a hierarchy of classes actually executed, i.e., the data allow us to account for overriding. In the case of a static call, bindToClass contains the class defining this static method. This way, the transformation algorithm can determine the specific class and method invoked by the method call. SourceLocation (in MessageLog) specifies the location (name of the class and lineNumber) in the source code from where the logged method call has been made: this is the call site.

#### B. Control flow

The control flow model (Fig. 3) captures a method's code structure in terms of method calls (identified by line numbers), possibly performed under conditions (alternatives, loops). It allows us to accurately locate method calls from the source code based on matching MessageLogs from the trace model and then place them into the UML sequence diagram structure. Knowledge of a method call's host method and, if the method call is inside a condition, its control flow structures, will allow us to accurately construct the sequence of executions with reduced dynamic (trace) data. Specifically, the value of attribute covered of a MessageLog's sendEvent gives us

<sup>4</sup> Although collected at runtime, the signature includes the types of formal parameters, not the actual parameter ones. We can also collect dynamic types and parameter values but this was not necessary for our purpose.

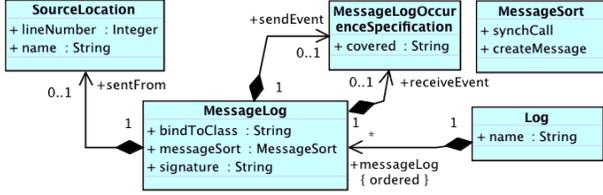


Fig.2. Trace model

the class name from where the call captured by the log originated; the value of attribute `lineNumber` of the `MessageLog`'s `Source-Location` then gives us the exact statement of the calling class source code that executed. Looking for this line number of that class in the corresponding control flow model instance tells us whether the statement (and therefore the call) was performed in a conditional statement and which method of the caller class performed the call.

In Fig. 3, a `Class` whose behaviour is monitored contains `Methods` which in turn contain sequences of `CodeSections`. A `CodeSection` can be a `MethodCall` (we do not distinguish constructors) or a `ConditionalSection`, possibly nested (a `ConditionalSection` contains a sequence of `CodeSections`). A `ConditionalSection` is either an `Opt`, an `Alt` or a `Loop`. A `Loop` has a `LoopType` set to either `for`, `do` or `while`. Attribute `conditionDescription` (class `ConditionalSection`) specifies the actual condition. A `MethodCall` is performed at a specific `lineNumber` in a specific caller method (`isInMethod`) in a specific class (`isInClass`). `isInMethod` contains the signature of the method this method call is in. For `MethodCalls` outside a `ConditionalSection`, `method.signature` (navigating the association between `MethodCall` and `Method`, inherited from `CodeSection`, and accessing attribute `signature` of the calling method) equals `isInMethod`. However, `MethodCalls` inside a `ConditionalSection` do not have direct access to this association and therefore need to carry the `isInMethod` attribute. Attribute `isInClass` contains the name of the `Class` the `MethodCall` is in, and is needed for similar reasons. An `Alt` `ConditionalSection` does not contain information about true/false branches because they are not handled as such in this work: each branch (either true or false) is an `Alt` instance.

## V. TOOL SUPPORT

Section V.A discusses the aspects we used to collect runtime information (traces). Section V.B discusses the control flow model construction. Section V.C discusses the model transformation. We only highlight the main principles due to space constraints. An example illustrating the models and the model transformation is discussed in section V.D.

### A. Aspects

The premise of this work is to provide a lighter instrumentation strategy than previous, only dynamic analysis works that are close to our technique [6, 30], specifically (1) avoiding instrumenting control flow structures in the source code [6] or the byte code [30] and (2) limiting the impact of

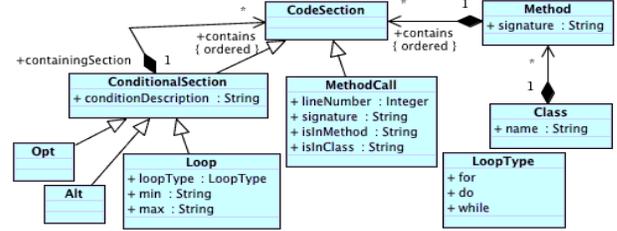


Fig.3. Control flow model

aspects, i.e., reducing the number of recorded logs. To avoid instrumenting the source code, since AspectJ [9] does not provide pointcuts for control flow structures (even `cflow` and `cflowbelow` pointcut designators do not help), we turned to static analysis (control flow graph created by parsing the source code). Note that even if AspectJ were providing such pointcuts, combining a static analysis with a dynamic analysis would still be preferable to limit the probe effect (fewer aspects and pointcuts would be needed).

With a hybrid analysis we need a way to match static information to dynamic trace information. Collecting pairs (class name, method signature) for executing methods is not sufficient since we know nothing about the caller. Instead of instrumenting method executions [5, 6, 30], we therefore instrument method calls as this allows us to collect information about both the caller (line number and the source file name from where the call was made) and the callee (class name, object identity). Combined with a unique identification of executing objects, we can correctly link dynamic and static data since information on class names and line numbers is also in the control flow graphs.

Once we can associate a method call from the trace (`SourceLocation`'s attributes) to the location in the source code where that call is made (`MethodCall`'s attributes), the static analysis allows us to determine from which method in which class the call was made and whether this call is inside a condition or a loop.

Furthermore, when using a call joinpoint, AspectJ can provide information about both the source and destination methods, as opposed to an execution joinpoint which only provides information about the callee. Since control flow is obtained statically, we do not need to detect the start and end of executing methods, i.e., we do not need an around advice: we only need a before advice, which further reduces the number of log statements compared to previous work [6], an improvement we expect to translate into significantly lower overhead and faster executions. One drawback however is that call joinpoints cannot catch calls to `super()` or `this()` while execution joinpoints would (as per the AspectJ specification). We consider this a small limitation.

We designed four aspects. The first one adds to classes the capability to count their instances, and adds to those instances the capability to report on their unique identifier (classes implement interface `ObjectID` which defines method `getObjectID()`). The other aspects intercept calls to methods (either static or not) and constructors and collect information before they are made: call join point, before advice. The

```

1 before(): callMethod () {
2 // local variables defined
3 if (thisJoinPoint.getThis() != null) {
4 // the calling object is instrumented and the calling
// method is not static. We get the ID of the calling
// object using:
// ((ObjectID) thisJoinPoint.getThis()).getObjectID()
5 }
6 else {
7 // the calling object is not instrumented or the calling
// method is static. We get the caller class name using:
// thisJoinPointStaticPart.getSourceLocation().toString()
8 }
9 // Similarly we collect the ID or class name of the callee.
10 // We collect:
11 // - the class name of the callee using:
// MethodAspect.getBindToClassName(
// thisJoinPoint.getTarget().toString())
12 // - the signature of the called method using:
// MethodAspect.getMethodSignature(thisJoinPoint.toString())
13 // - the location (line number) of the call using:
// MethodAspect.getLineNumber(
// thisJoinPointStaticPart.getSourceLocation().toString())
14 // - the filename where the call site is located using:
// MethodAspect.getFileName(
// thisJoinPointStaticPart.getSourceLocation().toString())
15 // We log all this information.
16 }

```

Fig.4. Excerpt of advice for pointcut `callMethod` (the other advices are similar)

collected information includes: unique identifiers (or class name in case of static methods) of interacting objects (or classes), the signature of the method being called (including formal parameter information), and the line number where the call is made. Fig. 4 shows the aspect we used to intercept method calls. It is a representative example of the trace-generating aspects. We do not show all the details of this aspect, such as actual data structures, but indicate which parts of the AspectJ API we use to collect information.

It is important to note that our aspects are defined in an abstract way (i.e., definitions are not specific to any Java class to instrument), providing templates that can be automatically tailored to instrument any Java class.

We do not discuss the transformation of a trace into an instance of the trace model as there is no technical difficulty.

### B. Control flow

We created a JavaCC (with JJTree) parser to generate instances of the control flow model, using a simplified Java grammar since we are only interested in class definitions (including inner class definitions), method/constructor definitions, method and constructor calls (including those passed as arguments of other calls or those used in control flow structures), and control flow structures. Our parser can handle `if`, `else if` and `else`, `while` loop, `for` loop (including `for-each`) but not `?:` and the `do-while` loop (doing so is not a technical challenge). It does not handle exceptions, which we will consider in the future. Note that when a condition contains a method call, the method call will appear in the control flow graph right ahead of the condition (outside of the conditional control flow construct). This is to better match the trace information and the UML sequence diagram notation. Control flow generation is not further discussed here since it does not pose any technical difficulty.

### C. Model transformation

We formalized the different steps of our transformation of instances of the trace and control flow models into an instance of the UML (sequence diagram) metamodel in terms of mapping rules between these models, using the OCL [25]. We created eleven such rules, which range from four to 50 lines of OCL to (1) match every single message log to a message in a sequence diagram, (2) match control flow structures to combined fragments, (3) ensure the order of messages matches the order of logs.

The transformation was then performed with MDWorkbench ([www.mdworkbench.com](http://www.mdworkbench.com)), an Eclipse-based IDE for model driven development that provides a model transformation capability. Transformations are rule-based, specified in a proprietary, imperative model transformation language, and can transform any number of source models into any number of target models.

Our OCL rules can be seen as a specification for the transformation and were useful to identify whether our trace and control flow models had the required

information to accurately perform transformations. They are also used to ensure partial correctness of the generated diagrams.

We specified our models using XML Schema. MDWorkbench uses these rules to manipulate instances of the trace and control flow models (XMI input files), producing an instance of the UML sequence diagram model (XMI output file). This XMI output file is ready to be used by any UML CASE tool (we used IBM RSA). The transformation rules are not detailed here due to lack of space.

### D. Illustrating example

Fig. 5 shows (excerpt) the source code of class A (I), the control flow model of classes A and B (II and III), and the trace model instance for one execution. We only focus on the important aspects that allow us to illustrate the approach: as a result, some information is simply not available in the figure: e.g., attribute values of the `SourceLocation` and `sendEventMLOS` objects linked to `ML5` refer to source code information of the call to `m()` on an instance of `A`, which is not in part I of the figure, although not used in the example, our approach handles method parameters and return values as discussed in sections IV and V and illustrated in figures 2 and 3.

We assume reference `b` (part I) is an instance of class `B`. The `lineNumber` attribute values in parts II and IV correspond to the line numbers in part I. In part IV, `MLOS` simply refers to `MessageLogOccurrenceSpecification`. In part II, we recognize that `m()` contains an `Opt` alternative, which is itself made of a sequence of a `Loop` (performing a method call to `n()` on `b`) followed by a method call (to `m()` on `b`). The numbers 1 and 2 on the `Loop` and `MethodCall` sides of the links simply indicate that the links between the `Opt` object, and the `Loop` and `MethodCall` objects are ordered (`CodeSections` are ordered in a method, Fig. 3). In part IV, we assume that the execution of the program resulted in four initial log messages, followed by log messages `ML5`, `ML6`, ...

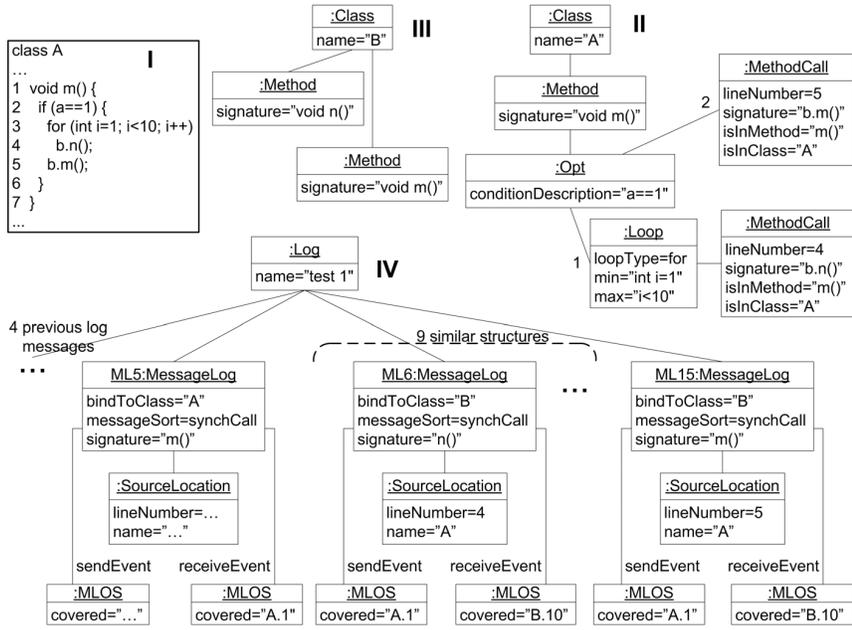


Fig.5. Illustrating example (excerpt): (I) source code for class A; (II) control flow model for class A; (III) control flow model for class B; (IV) trace model for one execution

ML15. Since the loop is executed nine times, there is a total of nine structures similar to ML6 and its linked instances in the sequence of MessageLog instances linked to the Log instance (Fig. 2). Instances of objects executing methods are uniquely identified by our aspects, which is represented in part IV as strings "A.1" and "B.10", suggesting that the instance of A executing m() is the first instance of A ever created in the program, and that calls to n() and m() are performed on the tenth instance of B created.

Let us now illustrate the essence of the model transformation. The trace model instance shows two executing objects: instance 1 of A and instance 10 of B. This allows the transformation to create two lifelines; one for each of these instances. Instance ML5 shows the call to m() on instance 1 of class A: the receiveEvent MLOS linked to ML5. The following MessageLog in the sequence from the Log instance, specifically ML6, shows a call to n() on the tenth instance of B (the receiveEvent MLOS linked to ML6) performed by the first instance of A (the sendEvent MLOS linked to ML6). Since there is no other MessageLog between ML5 and ML6 in the sequence, and the receiveEvent of ML5 and the sendEvent of ML6 have the same covered attribute value (A.1), we can conclude that the call to n() in ML6 is performed by m() which has been called in ML5. This allows the transformation to create an execution specification on the life line for A.1, showing the execution of m(), another execution specification on the life line for B.10, showing the execution of n(), as well as a message from the m() execution specification to the beginning of the n()'s execution specification. The same principle applies to the eight other MessageLog instances similar to ML6, as well as ML15, resulting in Fig. 6 (a).

The transformation also recognizes that the call to n() on an instance of B recorded by ML6 occurs at line 4 (attribute

lineNumber of the ML6's SourceLocation instance), that this call is performed in method A.m() (we already discovered that), and that, from the control flow model of method A.m(), the call to n() on an instance of B at line 4 happens in a loop, which itself happens in an alternative. This applies to all the ML6 to ML14 objects: we therefore obtain nine Loop combined fragments each containing a message labeled n(): Fig. 6 (b). Last, since ML15 is a call that happens in A.m() at line 5 and that this call happens (control flow model) in the alternative, after the loop, we can correctly place message m(): Fig. 6 (b).

Collapsing the nine Loop combined fragments and their message labeled n() into one message in one Loop combined fragment, itself inside an Opt combined fragment, is not an easy task and is part of our future work. In the simple example discussed here, it would be easy to do that. However, in general, the contents of the loop instances may not be the same

(different control flows may be triggered in the loop), making the merging difficult.

## VI. CASE STUDIES

We performed case studies with two research questions in mind: (RQ1) Are the resulting scenario diagrams equivalent to the ones, previously deemed correct in our previous work [6] (UML 1 vs. UML 2)? (RQ2) Is the execution overhead, measured as execution time, reduced when our approach is used compared to this previous work and if so what is the amount of reduction?

Note that we only compare to this previous work since we identified this is the approach closest to ours. Future work will also compare our approach to the other related works (e.g., [30]). In our previous purely dynamic approach [6], we traced method entry and exit (around advice), conditions, and loops. To trace control flow information we instrumented the source code in addition to using aspects since AspectJ did not offer any mechanism (i.e., join point) to do that. The instrumentation

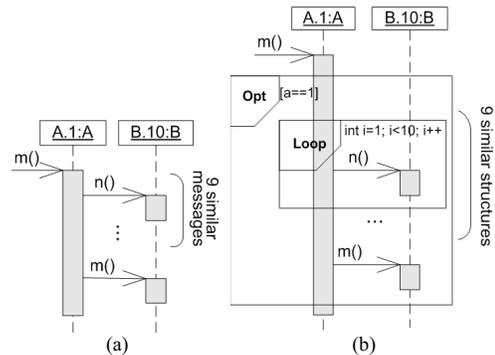


Fig.6. Illustrating the model transformation

strategy therefore involved two calls to the trace logger for every method execution and two additional calls for each control flow structure (i.e., condition or loop).

#### A. Experiment set up

To answer these research questions, we relied on five different case study systems (Table I). The first system, the *Library* (server side only), was used before [6]. The second one is a short *Example* (40 lines of Java code) we specifically built to exercise and control many different situations (e.g., nested loops and numerous iterations of loops). Through a command line argument we can control the amount of times method calls are performed (including calls to different objects), loops are executed, thereby simulating larger program executions. Typically, a command line argument value of  $n$  approximately leads to  $n$  loop iterations,  $5.n-3$  if-then or if-else executions, and  $5.n+2$  method executions. The “Method Calls” column reports on the number of calls to constructors, static and non-static methods we observed, thanks to a dedicated simple AspectJ aspect, when executing the different case studies (see below for details on the test cases). We will use this information when answering RQ2; the data is therefore not relevant for Library. The code for Example does not contain any computation, any GUI, any interaction with IO devices (e.g., reading a file). This should allow us to evaluate to what extent our technique reduces the overhead as the size of the software (simulated by increasing the number of loops and calls) increases, without actually using a larger software. Note however, that we expect execution overhead results to be worse with Example than with a real system exercising a similar pattern of calls and control flows, since we do not have any computation in Example. In other words, the percentage of increased execution time would be smaller than what we report with Example. The third system, *PPC Prover*, implements the Proof Carrying Code (PCC) technique [20], a technique for safe execution of untrusted code. The fourth system is a simple *Calculator*, partly generated by JavaCC, that implements the Visitor design pattern. Finally, we used *Weka*<sup>5</sup>, a well-known machine learning and data mining software.

Our reverse engineering technique necessarily needs executions, i.e., test cases. To answer RQ 1, each of the first four case studies was executed with one test case that we selected to not produce too large traces. This way we could easily verify the correctness of our approach without additional technology to handle large traces or diagrams. We were then looking for expected patterns of object interactions in the reverse-engineered diagrams: In the case of Calculator we expected to see the use of the well-known visitor design pattern, which involves very distinctive object interactions, in the scenario diagram; In the case of PPC Prover, and Example, we were expecting to see known interactions in the scenario diagram since the second author of this paper produced the code; In the case of Library, we used the test input we used before [6] to allow scenario diagram comparisons. We did not answer RQ1 with Weka since generated traces were much larger and we did not have any oracle to decide whether generated scenario diagrams were correct. We believe that answering RQ1 with the four first case studies would bring

TABLE I. CHARACTERISTICS OF THE FIVE CASE STUDY SYSTEMS

Case study name	Classes	LOC	Method Calls	Question(s)
Library (Server)	44	3280	N/A	1
Example (200)	4	40	1000	1, 2
Example (800,000)	4	40	4,000,000	1,2
PPC Prover	8	1280	1138	1, 2
Calculator	16	1175	113	1, 2
Weka TestSuite	1180	238,556	4,497,261	2
Weka TC4	1180	238,556	3,993,699	2

enough confidence in our approach. Future work will include handling large traces and visualizing large scenario (or sequence) diagrams for large case studies such as Weka to confirm this. Due to space constraints we do not discuss the results to RQ1 in details. We simply note that after investigations and comparisons with the source code, and available or expected diagrams (e.g., previous study using the Library system, behaviour of the visitor design pattern for Calculator), we concluded that our diagrams are accurate and provide as much information as in previous work [6].

To answer RQ2 (overhead impact), we measured execution time. We relied on three versions of each one of the last four systems (Table I): one with no instrumentation (referred to as the *base* version), one with our light instrumentation (the *light* version), and one with our previous instrumentation [6] (the *original* version). Note that to avoid a bias in favour of the light instrumentation, we removed the recording of node and thread IDs and timestamps from the original instrumentation, necessary in the original technique to trace RMI and thread communications, thereby making the two instrumentation techniques comparable. Otherwise, the original technique would be collecting more data than the new one and would therefore be put at a disadvantage.

Overhead was studied in three steps. First, for Example (with command line argument value 200), PPC Prover, and Calculator, we executed each test case 100 times in an attempt to control for the possible impact of the operating system on execution time. We selected input 200 for Example as this leads to a number of method calls similar to our execution of PPC Prover (Table I) and would therefore allow comparisons.

In a second step, Example was executed with varying command line argument values to trigger large to very large amounts of method calls and loop executions. Our objective was to study the probe effect of our instrumentation on execution time by simulating the instrumentation impact on execution time for larger systems, without actually using a larger system. Specifically, we executed Example with a command line argument values of  $10^2$ ,  $10^3$ ,  $10^4$ ,  $10^5$ ,  $10^6$ , and  $10^7$ , 100 times each. (Recall from the previous section that this resulted in as many loop iterations and even more if-then or if-else and method executions.) This necessarily resulted in large traces. However, for RQ2 we were only interested in measuring execution time; we were not actually reverse-engineering and visualizing scenario diagrams.

In a third step, as opposed to simulating long executions and therefore large traces, we actually used a much larger system: Weka. For this purpose we collected test cases from the Weka user manual [4] and executed them, by-passing the

<sup>5</sup> <http://www.cs.waikato.ac.nz/~ml/weka/>

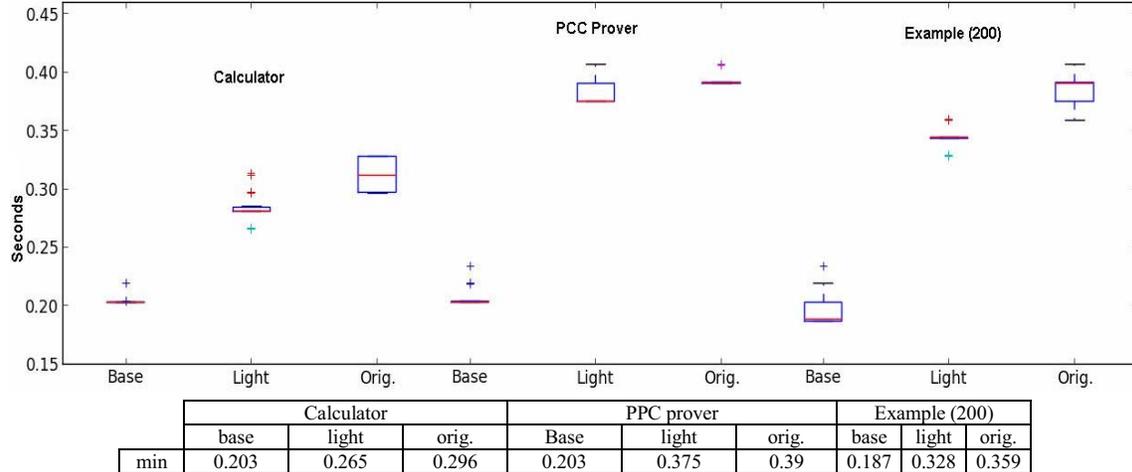


Fig. 7. Execution times (in seconds) for different instrumentations of Calculator, PPC prover and Example (input value of 1000)

GUI: use of *ZeroR* and *J48* classifier models to map from all-but-one dataset attributes to the class attribute (two examples); use of the *meta-classifiers* command to apply multiple classifiers models in parallel for a dataset (stacking) (one example); supervised filtering of datasets based on class information to discretize numeric attributes into nominal attributes and create a stratified subsample of given dataset (resample) (two examples); and list information for the specific package from the Weka server which was locally cached as the network was off at the time of experiment (one example). We used these test cases in two situations. We created a test suite made of all test cases executed in sequence, which we refer to as *Weka\_TestSuite* (Table I); We studied the longest, in terms of execution time, test case separately (*Weka\_TC4* in Table I). We also compared the executions of *Weka\_TC4* with one execution of *Example* (with input 800,000) as they exhibit similar numbers of method calls (Table I).

Execution times were collected on an Dell PC with an Intel(R) Xeon(R) (at 2.66 Ghz) quad core and 16 GB of memory, running WindowsXP 64x OS, JDK 1.7.0\_21, and AspectJ 1.6.7. Time was measured using the `Java System.currentTimeMillis()` function before and after every execution to be measured. (When possible, all other applications and services running on the computer were turned off, and the network was disconnected.) Note that this measure of time includes the start of the JVM as well as other JVM bookkeeping activities (e.g., garbage collector). However, we deemed our number of executions sufficient to average out such unexpected behaviours. Plus, since we intend to compare execution times for the three versions, this should not have any impact on our conclusions. Last, as shown and discussed next, we did not observe many outliers, suggesting that the environment of execution was reasonably stable, therefore allowing us to compare data.

### B. Results—Overhead

Before discussing results, recall that we expect variations from system to system because of differences in instrumentation techniques (e.g., we use a less demanding before advice than an around advice as in [6]) and the

characteristics of the systems (e.g., amount of loops or if statements, amounts of calls to monitor). For instance, if we only consider the number of calls to the logger, which is one of the main sources of overhead, the differences between light and original are due to the fact that for the light instrumentation, the number of calls is the sum of the number of method and constructor calls whereas for the original instrumentation, the number of calls is the sum of twice (because of the around advice) the number of method and constructor calls and twice the number of conditions and loops encountered.

Execution times are reported in Figures 7 to 9: box plots indicate all observed execution time values (all in seconds) along the y-axis, including the minimal and maximal ones, as well as first and third quartiles encompassing time range achieved by half of the total executions (recall that each instrumentation version was executed 100 times). Figures also provide tables with execution time values of the different instrumentations. We discuss these figures in sequence as they correspond to the three experimental steps we discussed earlier.

While we tried to control other operating system activities, there is still a large variation in the execution times obtained (Fig. 7). This is especially true for the original executions, and is likely due to their longer execution times, which gives more chances to some operating system tasks to intervene. However, compared to the differences between minimum and maximum execution times, most execution times lie within a narrow range (so we can discard the outliers). Notice that the largest number of points for each variation is found at or near the minimum execution time. This is likely due to the fact that most of the time there were very few other processes running on the computer. The higher points probably occurred during times that the processor was handling other expensive system events we were not able to control. On average, the light instrumentation causes the program to execute slower than it would without instrumentation but much faster than with the original instrumentation, especially as the number of method calls grows, i.e., the size of the instrumented program grows: Example or PPC versus Calculator.

We compared the samples of 100 execution times obtained with the light and original versions and with the original and base versions, for all three systems: we used a one-tailed t-test and confirmed the results with the corresponding non-parametric Wilcoxon signed-rank test. All comparisons are statistically significant (p-value threshold at 0.05), with all the p-values smaller than 0.0001. The light instrumentation statistically leads to a smaller overhead than the original instrumentation.

We investigated reasons for differences in execution times between the light and the original instrumentations. Regardless of the instrumentation strategy, the instrumentation makes system calls to write to a log file. A file needs to be created and written to repeatedly, which are execution-heavy tasks (heavier than any instrumentation-related behaviour added to the programs). We will investigate other logging mechanisms than writing to a file in the future. Also, we noticed the amount of characters written into a file for our light instrumentation is higher than for original. For example, the first trace entry for Calculator light was 249 characters long, versus 175 for original. The light instrumentation is therefore not entirely “lighter” than original for executions small enough not to be negatively affected by large numbers of instrumentation calls. Indeed, even though the original instrumentation writes to a file at least twice as often as our light instrumentation, the number of times the file is written to is small so the difference is not important. We suspect that the overhead when the program is small mostly comes from creation of the trace file.

The second step of the investigation consisted in simulating the instrumentation impact on execution time for larger systems. We executed Example with command line argument values  $10^2$ ,  $10^3$ ,  $10^4$ ,  $10^5$ ,  $10^6$ , and  $10^7$ , 100 times each. Figure 8 shows the results, using a logarithmic scale on the y-axis. We only show the median execution time over the 100 executions: the standard deviation was very small. Again, the light instrumentation causes the program to execute slower than it would without instrumentation but much faster than with the original instrumentation: the light instrumentation is two times faster than the original one for  $10^6$  and  $10^7$ , 1.8 times faster for  $10^4$  and  $10^5$ . Figure 8 also shows that additional work is required to further reduce the impact of instrumentation since the light instrumentation is slower than the base execution. We believe a different tracing mechanism with fewer accesses to the disk to save smaller amounts of data should be used as we have already mentioned; one may also consider instrumenting only parts of a large program to reverse engineer.

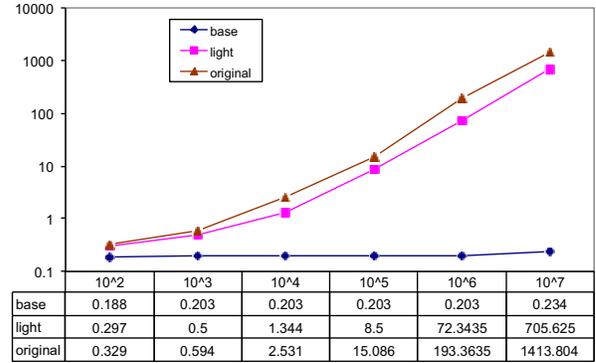


Fig.8. Median execution times (in seconds) as the number of method calls increases

In the third step we studied impact on execution time with Weka: Figure 9. Once again, boxplots are the results of 100 executions. We observe that the original instrumentation is 2.5 (resp. 3.5) times slower than the light one for the whole test suite (resp., TC4), and the light one is much slower than the base one confirming there is room for improvements, i.e., probe effect reduction. Recall from Table I that executing Example with input 800,000 triggers a very similar number of calls as Weka\_TC4. The larger overhead difference between Orig and Light for Weka\_TC4 than for Example is therefore due to Orig’s instrumentation of the control flow.

These results overall confirm that removing instrumentation of control flow structures helped reduce the probe effect, and that our aspects (before rather than around advice) also helped.

## VII. CONCLUSIONS

In this paper, we presented a hybrid technique, relying on both static (control flow) and dynamic (execution trace) information, to automatically reverse engineer scenario diagrams. Our objective was to (1) obtain scenario diagrams that are equivalent to what previous techniques can generate (i.e., sequences of messages with information on conditions and loops triggering those messages, represented under the form of the UML sequence diagram), while (2) reducing the amount of instrumentation of the bytecode and avoiding instrumenting the source code. The latter is particularly important as we do not want the instrumentation to affect too much the program behaviour to the extent that there would be a risk of not observing the right behaviour when executing the

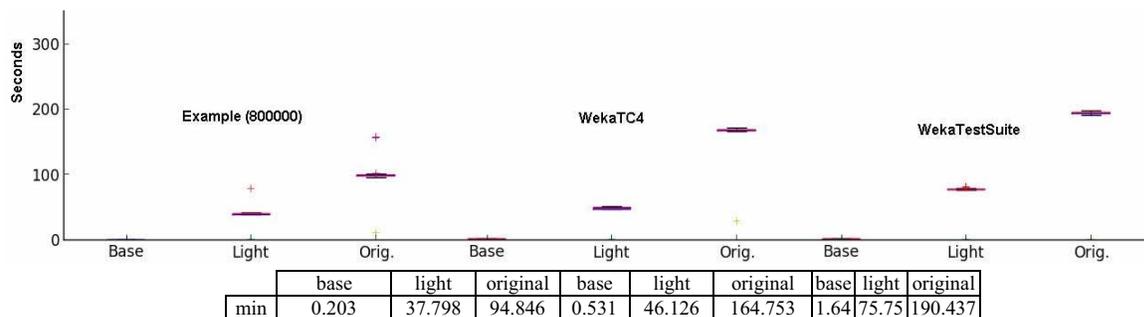


Fig.9. Execution times (in seconds) for Weka instrumentation (100 executions)

instrumented program.

We therefore tried to limit the impact of our aspects for trace generation (e.g., tracing calls rather than executions, with a before rather than an around advice). In parallel we generated simple control flow graphs of methods: we were only interested in method definitions, the method calls they trigger and the conditions under which those calls are triggered. We represented both sets of information using UML class diagrams. A model transformation then transforms an instance of the trace model and instances of the control flow model to an instance of the UML sequence diagram metamodel.

We performed several case studies that indicate that we achieved our goals: (1) although we did not discuss these results in details due to space constraints, we noted that the generated diagrams were equivalent to the ones generated by a previous technique; (2) we reduced (at least by half in large systems) the probe effect due to instrumentation.

There is room for future work. First, we intend to combine our instrumentation strategy with previous techniques to monitor network communications: either assuming RMI communications [6] or not [30]; and thread communications [6]. Second, our experimental results show we can still reduce execution time overhead by for instance considering other mechanisms than a file to collect trace information. We also intend to perform more extensive experimentations to more precisely understand what aspects of the approach hurts the most in terms of probe effect, given characteristics of the program being monitored. We also intend to combine our technique with existing trace minimization techniques (e.g., [8, 11, 12, 19, 27]). Another important challenge will be to combine several scenario diagrams and create accurate, complete sequence diagrams. There is work in the literature we can get ideas from [7] with that respect.

#### References

- [1] Ackermann C., Lindvall M. and Cleaveland R., "Recovering views of inter-system interaction behaviors," Proc. IEEE WCRE, pp. 53-61, 2009.
- [2] Alalfi M. H., Cordy J. R. and Thomas D., "Automated reverse engineering of UML sequence diagrams for dynamic web applications," Proc. IEEE ICST Workshops, pp. 287-294, 2009.
- [3] Bennett C., Myers D., Storey M.-A., German D. M., Ouellet D., Salois M. and Charland P., "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," Wiley SME, 20 (4), pp. 291-315, 2008.
- [4] Bouckaert R. R., Frank E., Hall M., Kirkby R., Reutemann P., Seewald A. and Scuse D., "WEKA Manual, version 3-7-7," User manual, 2012.
- [5] Briand L. C., Labiche Y. and Leduc J., "Tracing Distributed Systems Executions Using AspectJ," Proc. IEEE ICSM, pp. 81-90, 2005.
- [6] Briand L. C., Labiche Y. and Leduc J., "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," IEEE TSE, 32 (9), pp. 642-663, 2006.
- [7] Cornelissen B., Zaidman A., Van Deursen A., Moonen L. and Koschke R., "A Systematic Survey of Program Comprehension through Dynamic Analysis," IEEE TSE, 35 (5), pp. 684-702, 2009.
- [8] Dugerdil P. and Repond J., "Automatic generation of abstract views for legacy software comprehension," Proc. ACM ISEC, pp. 23--32, 2010.
- [9] Gradecki J. D. and Lesiecki N., Mastering AspectJ - Aspect-Oriented Programming in Java, Wiley, 2003.
- [10] Guéhéneuc Y.-G., "A reverse engineering tool for precise class diagrams," Proc. IBM CASCON, pp. 28-41, 2004.
- [11] Hamou-Lhadj A., Techniques to Simplify the Analysis of Execution Traces for Program Comprehension, Thesis, University of Ottawa, 2006
- [12] Hamou-Lhadj A. and Lethbridge T. C., "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," Proc. IEEE ICPC, pp. 181-190, 2006.
- [13] Imazeki Y. and Takada S., "Reverse engineering of sequence diagrams from framework based web applications," Proc. IASTED SEA, pp. 202-209, 2009.
- [14] Ishio T., Watanabe Y. and Inoue K., "AMIDA: a sequence diagram extracting toolkit supporting automatic phase detection," Proc. Companion of the ACM ICSE, 2008.
- [15] Koskimies K., "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs," Proc. IEEE ICSE, pp. 366-375, 1996.
- [16] Koskinen J., Kettunen M. and Systa T., "Profile-Based Approach to Support Comprehension of Software Behavior," Proc. IEEE ICPC, pp. 212-224, 2006.
- [17] Malloy B. A. and Power J. F., "Exploiting UML Dynamic Object Modeling for the Visualization of C++ Programs," Proc. ACM SOFTVIS, pp. 105-114, 2005.
- [18] Munakata S., Ishio T. and Inoue K., "OGAN: visualizing object interaction scenarios based on dynamic interaction context," Proc. IEEE ICPC, pp. 283-284, 2009.
- [19] Myers D., Storey M.-A. and Salois M., "Utilizing Debug Information to Compact Loops in Large Program Traces," Proc. IEEE CSMR, pp. 41-50, 2010.
- [20] Necula G. C., "Proof-carrying code," Proc. ACM PoPL, pp. 106--119, 1997.
- [21] Ng J. K.-Y., Guéhéneuc Y.-G. and Antoniol G., "Identification of behavioural and creational design motifs through dynamic analysis," Wiley SME, 22 (8), pp. 597-627, 2010.
- [22] Nguyen D.-P., Luu C.-T., Truong A.-H. and Radics N., "Verifying Implementation of UML Sequence Diagrams Using Java PathFinder," Proc. IEEE KSE, pp. 194-200, 2010.
- [23] Noda K., Kobayashi T., Agusa K. and Yamamoto S., "Sequence Diagram Slicing," Proc. CPS APSEC, pp. 291-298, 2009.
- [24] OMG, "UML 2.0 Superstructure Specification," Object Management Group, Final Adopted Specification ptc/2007-11-02, 2007.
- [25] Pender T., UML Bible, Wiley, 2003.
- [26] Pilskalns O., Wallace S. and Ilas F., "Runtime debugging using reverse-engineered UML," Proc. Models, LNCS vol. 4735, pp. 605-619, 2007.
- [27] Reiss S. P. and Renieris M., "Encoding Program Executions," Proc. ACM/IEEE ICSE, pp. 221-230, 2001.
- [28] Roubtsov S., Serebrenik A., Mazoyer A. and van den Brand M., "I2SD: Reverse Engineering Sequence Diagrams from Enterprise Java Beans with Interceptors," Proc. IEEE SCAM, pp. 155-164, 2011.
- [29] Serebrenik A., Roubtsov S., Roubtsova E. and van den Brand M., "Reverse engineering sequence diagrams for Enterprise JavaBeans with business method interceptors," Proc. IEEE WCRE, pp. 269-273, 2009.
- [30] Systa T., Koskimies K. and Muller H., "Shimba - An Environment for Reverse Engineering Java Software Systems," SPE, 31 (4), pp. 371-394, 2001.
- [31] Taïani F., Killijian M.-O. and Fabre J.-C., "COSMOPEN: dynamic reverse engineering on a budget. How cheap observation techniques can be used to reconstruct complex multi-level behaviour," SPE, 39 (18), pp. 1467-1514, 2009.
- [32] Wang F. Q., Ke H. J. and Liu J. B., "Towards the Reverse Engineer of UML2.0 Sequence Diagram for Procedure Blueprint," Proc. WRI WCSE, 3, pp. 118-122, 2009.
- [33] Zaidman A., Matthijssen N., Storey M.-A. and Van Deursen A., "Understanding Ajax Applications by Connecting Client and Server-Side Execution Traces," ESE, 18 (2), pp. 181-218, 2013.
- [34] Zhou Z., Wang L., Cui Z., Chen X. and Zhao J., "Jasmine: A Tool for Model-Driven Runtime Verification with UML Behavioral Models," Proc. IEEE HASE, pp. 487-490, 2008.
- [35] Ziadi T., da Silva M. A. A., Hillah L. M. and Ziane M., "A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams," Proc. ICECCS, pp. 107-116, 2011.

# An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams

Hafeez Osman\*, Michel R.V. Chaudron\*<sup>§</sup> and Peter van der Putten\*

*\*Leiden Institute of Advanced Computer Science*

*Leiden University, Leiden, the Netherlands*

*{hosman,putten}@liacs.nl*

*§Joint Department of Computer Science and Engineering*

*Chalmers University of Technology and Goteborg University*

*Gothenburg, Sweden*

*chaudron@chalmers.se*

**Abstract**—There is a range of techniques available to reverse engineer software designs from source code. However, these approaches generate highly detailed representations. The condensing of reverse engineered representations into more high-level design information would enhance the understandability of reverse engineered diagrams. This paper describes an automated approach for condensing reverse engineered diagrams into diagrams that look as if they are constructed as forward designed UML models. To this end, we propose a machine learning approach. The training set of this approach consists of a set of forward designed UML class diagrams and reverse engineered class diagrams (for the same system). Based on this training set, the method ‘learns’ to select the key classes for inclusion in the class diagrams. In this paper, we study a set of nine classification algorithms from the machine learning community and evaluate which algorithms perform best for predicting the key classes in a class diagram.

**Keywords**-Software Engineering; UML; Reverse Engineering; Machine Learning; Program Comprehension;

## I. INTRODUCTION

Up-to-date design documentation is important, not just for the initial design but also in later stages of development and in the maintenance phase. UML models created during the design phase of a software project are often poorly kept up to date during development and maintenance. As the implementation evolves, correspondence between design and implementation degrades [1]. For legacy software, faithful designs are often no longer available, while these are considered valuable for maintaining such systems.

A popular method to recover an up-to-date design of a system is reverse engineering. Reverse engineering is the process of analyzing the source code of a system to identify the systems components and their relationships and create design representations of the system at a higher level of abstraction [2]. Reverse engineering also refers to methods aimed at recovering knowledge about a software system in support of execution some of software engineering task [3]. Tool support during maintenance, re-engineering or re-architecting activities has become important to decrease the

time that software personnel spends on manual source code analysis and helps to focus attention on important program understanding issues [4]. However, current reverse engineering techniques do not yet solve this problem adequately. In particular, reverse engineered class diagrams are typically a detailed representation of the underlying source code. This makes it hard for software engineers to understand what the key elements in the software structure are [5]. Although several Computer Aided Software Engineering (CASE) tools have options to leave out several properties in a class diagram they are unable to automatically identify classes that are less important.

This paper is partially motivated by a scenario where new programmers want to join a development team. They need a starting point in order to understand the whole system before they are able to modify it. Provided with the software design, the programmer will normally browse the class design and try to synchronize the design with the source code. There is a need for programmers to know which classes in the system play important roles or can be considered as key classes in the system.

Fernández-Sáez et al. [6] found that developers experience more difficulties in finding information in reverse engineered diagrams than in forward designed diagrams and also find the level of detail in forward designed diagrams more appropriate than in reverse engineered diagrams. In order to achieve better reverse engineered representations we need to learn which information from the implemented system to include and which information to leave out. A method to assist software engineers to focus on the key classes and aspects of the design is needed. The identification of key classes can also be used to simplify a complex class diagrams or help to predict the severity of a defect in a software system.

**Research Problem.** This paper specifically aims at providing suitable classification algorithms to decide which classes should be included in a class diagram. We seek an automated approach to classify the key classes in a class diagram. The

algorithms need to be able to produce a score, not just a classification, so that a user potentially has the option to choose a particular level of abstraction for representing a reverse engineered design.

This paper focusses on using design metrics as predictors (input variables used by the classification algorithm). The advantage of using design metrics is that these can be obtained very efficiently with little effort. This fits our objective of creating a method that will be of practical use to software developers. Also, we analyse the predictive power of the predictors to know how influential each of these predictors is in the performance of the classifier.

**Contribution.** We explore several classification algorithms for predicting key classes that should be included in a class diagram. As input for this study, we use sets of source codes from open source projects with corresponding UML models that contain forward designed class diagrams. We use these class diagrams as ‘ground truth’ to validate the quality of the prediction algorithms. The methods we study will learn from the forward designed class diagrams which classes should be selected from the reverse engineered class diagrams.

In total, nine algorithms were selected for this comparison study. These algorithms will be evaluated in terms of accuracy and robustness with respect to the information that they recommend to keep in and leave out of the class diagram. The candidate set of algorithms includes: J48 Decision Tree, k-Nearest Neighbor, Logistic Regression, Naive Bayes, Decision Tables, Decision Stumps, Radial Basis Function Networks, Random Forests and Random Trees.

We have collected a diverse collection of data sets consisting of nine pairs of UML design class diagrams and associated Java source code derived from open source software projects. The number of classes in the source code of these projects ranges from 59 to 903. Out of these classes 3% to 47% was found to be included in the forward UML class diagram. The open source projects were chosen for a number of reasons. We wanted the data to be representative for the diversity and complexity of real world projects. The quality of documentation for open source projects varies widely and there is also a large variation in the ratio of classes in the forward design versus classes in the source code. In open source projects software design is not a mandatory requirement, and these projects rely on volunteers working together in a distributed fashion. Also, by using open source projects we make it easier for other researchers to reproduce or compare against our results, and develop new methods on the same data.

The paper is structured as follows: Section II discusses related research and Section III describes the research questions. Section IV explains the basics of machine learning. Section V explains the approach on how we conducted the evaluation. Section VI presents the result and section VII discusses our findings and future work. This is followed with conclusion in Section IX.

## II. RELATED WORK

The following studies are related to our research from the perspective of identifying key classes from software artefacts.

Zaidman and Demeyer [7] proposed a method for identifying key classes using web mining techniques. They used dynamic analysis of the source code as basis for the identification of key classes. For validating their method, they manually identified key classes from the software documentation. Recall and precision were used to evaluate the approach and they found that their approach was able to recall 90% of the key classes and the precision was slightly under 50%. However, dynamic analysis approaches need significant effort for collecting run-time traces.

Perin et al. [8] proposed ranking software artefacts using the PageRank algorithm. They used the Pharo Smalltalk system and Moose reengineering environment as case studies. For the Pharo Smalltalk system, they reported that their approach was able to detect 42% of the important classes (prediction based on classes) and to detect 52% of the important classes when prediction was based on methods. However, no precision result was presented for the Moose system.

Hammad et al. [9] proposed an approach to identify the critical software classes in the context of design evolution. Version (commit) history and source code were used as the input for this study. They assumed that the classes that were frequently changed in the software evolution are the classes that are critical for the system and as the result, they found only about 15% of the classes in the case studies were impacted from six design changes.

Steidl et al. [30] presented an approach to retrieve important classes of a system by using network analysis on dependency graphs. They performed an empirical study to find the best combination of centrality measurement of dependency graph. They used classes recommended by their tests projects developers as the baselines.

Zimmermann et al. [10] presented a prototype tool that applies data mining to predict and suggest further changes to the developer when a system is modified. Version management repositories of open source projects were used as input.

Singer et al. [11] developed a tool to recommend files that are potentially relevant to the file that a developer is currently viewing. This study aimed at easing navigation in source code. The tool ranks the files that are (potentially) related.

Our work also aims at identifying key classes but we explore diverse classification algorithms based on supervised machine learning. In contrast, static analysis is used for our data collection as it is easy to obtain from open source projects. Our validation consists of comparing selected classes against those actually found in the forward design.

### III. RESEARCH QUESTIONS

This section describes the research questions of this study that will be answered in section VI.

**RQ1: Which individual predictors are influential for the classification?** For each case study, we explore the predictive power of individual predictors.

**RQ2: How robust is the classification to the inclusion of categories of predictors?** We explore how the performance of the classification algorithm is influenced by partitioning the predictor-variables in different sets.

**RQ3: What are suitable classification algorithms in classifying key classes?** The candidate classification algorithms are evaluated to find the suitable algorithm(s) in classifying the key classes in a class diagram.

### IV. BASICS OF MACHINE LEARNING

This section describes the basics of machine learning terms and algorithms used in this paper.

#### A. Univariate Analysis

To measure predictive power of predictors we used the information gain with respect to the class [33]. Univariate predictive power means measuring how influential a single predictor is in prediction performance. The results of this algorithm are normally used to select the most suitable predictor for prediction purposes. Nevertheless, in our study we did not use it for predictor selection, but for an exploratory analysis of the usefulness of various predictors (in our case: class diagram metrics). In our approach, class diagram metrics were used as predictors for the prediction of key classes.

#### B. Machine Learning Classification Algorithm

Prior to making a selection of the classification algorithms, we ran exploratory experiments on a wider range of algorithms. We do not expect that there will be a single silver bullet algorithm that will outperform all others across all sets of problems. Also, we are not just interested in a single algorithm that scores a top result on a given problem, but are looking for sets of classifiers (i.e. classification algorithms) that produce robust results across domains. In this way, algorithms become more portable across problems with very different rates of inclusion of classes in designs. We also aimed for a mix of classifiers in terms of expected bias (what relationships can be captured) and variance (does the prediction change when trained on different random samples) [15]. As discussed, we wanted to use a diverse set of algorithms representative for different approaches. For example decision trees, stumps, tables and random trees or forests all divide the input space up in disjoint smaller sub spaces and make a prediction based on occurrence of positive classes in those sub spaces. K-NN and Radial Basis Functions are similar local approaches, but the sub spaces here are overlapping. In contrast, logistic regression

and Naive Bayes model parameters are estimated based on potentially large number of instances and can thus be seen as more global models. The nine classification algorithms are the following (see [16] for more explanation):

1) *Decision Table*: A Decision Table consists of rows and columns that associate a set of conditions or test with a set of action. The Waikato Environment for Knowledge Analysis (WEKA) [33] used a simple Decision Table Majority (DTM) classifier.

2) *Decision Stumps*: Decision Stumps are decision trees consisting of just a single level and split [15]. A decision stump makes a prediction based on the value of just a single input feature, and is a good baseline classifier to compare against decision trees and other classifiers, to determine what results can already be achieved with a very basic model.

3) *J48 Decision Tree (J48)*: J48 is a WEKA implementation of the C.45 decision tree algorithm. This algorithm generates a classification-decision tree for the given data set by recursive partitioning of data.

4) *k-Nearest Neighbour (k-NN)*: k-NN classification finds a group of k objects in the training set that are most similar to the test object and bases its classification on the predominance of a particular class in this neighborhood [18].

5) *Logistic Regression(LR)*: uses a linear input combination of input variables to provide an output score, which is then mapped to a probability by applying a logistic function [19].

6) *Naive Bayes*: Naive Bayes is a classification algorithm based on the Bayes rule of conditional probability. It assumes independence across the predictors.

7) *Radial Basis Function (RBF) Networks*: RBF Networks are a type of neural network. We used simple normalized Gaussian functions that each cover a part of the input space, and the output is then fed into a basic feed forward neural network [20].

8) *Random Forest*: Random Forest is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [21].

9) *Random Tree*: Random Tree algorithm builds a classification tree considering K randomly chosen predictors at each node.

#### C. Evaluation Method

For Univariate analysis, the predictors were evaluated by using the InfoGain Attribute Evaluator (InfoGain). This method produces a value which indicates the influence of a predictor in prediction performance based on the case studies. A higher value of InfoGain denotes a stronger influence of the predictor (i.e. closer to 1 is better). The evaluation of machine learning classification algorithms started with generating a confusion matrix based on applying a classification algorithm in WEKA. Table I shows an example of a confusion matrix. In this table, for the case of the

Table I  
CONFUSION MATRIX

Prediction Result		Actual Result
Y	N	Y
<i>TP</i>	<i>FN</i>	Y
<i>FP</i>	<i>TN</i>	N

Example :

Y	N	Y
11	33	Y
10	849	N

actual data is positive (Y), *TP* represents the number of correct predictions (true positive) and *FN* represents the number of incorrect predictions (false negative) done by the classification algorithms. In the case of the actual data is negative (N), *FP* represents the incorrect predictions (false positive) while *TN* represents correct predictions (true negative). The example in Table I shows that the *TP* is 11 classes while the *TN* is 849. The *FN* is 33 and the *FP* is 10. The total number of classes in this case study is 903.

The classification algorithms were evaluated using Area Under ROC (ROC means Receiver Operating Characteristics) curve [22]. The AUC shows the ability of the classification algorithms to correctly rank classes as included in the class diagram or not. The larger the ROC area, the better the classification algorithms in term of classifying classes [23]. Based on this, WEKA provides AUC calculations that ease our evaluation task. This tool presents the result as a number between 0 and 1. A value closer to 1 means a better classification result while a value close to 0.50 means the classification performs almost randomly. The algorithms need to provide reliable estimates across the score range, thus we evaluate using the AUC value rather than for instance accuracy (e.g. percentage of correct or incorrect prediction). For highly unbalanced data this also avoids the issue of favouring models that just predict the majority outcome class.

In general, this measure is better than instance accuracy such as percentage correct because for projects with very low ratios of classes in the UML class diagram compared to the source code, a classification algorithm that would exclude all classes would score high on accuracy. For instance, referring to the example in Table I, the percentage of correct prediction is 95.24% and incorrect prediction is 4.76%. It seems that the algorithm performs an excellent prediction. However, the 95.24% correct prediction is the result for overall prediction result by taking the sum of correct prediction divided by number of classes. The percentage of correct prediction for *TN* is 98.80% while the percentage of correct prediction for *TP* is 25.00%. The resulting prediction performance for *TP* is very low even though overall correct prediction is very high. The AUC value measures performance of a model over the entire range of models scores, i.e. how well it separates by changing

the score threshold of a class over the entire score range. Therefore, AUC is preferred over accuracy as a measure.

Based on early observation on our case studies, we decided the threshold for the AUC value = 0.60. This means, if the AUC value  $\geq 0.60$ , the classification algorithm is considered to be usable for prediction for our specific problem.

## V. APPROACH

This section describes the Examined Predictors and Tools, Case studies and Process.

### A. Examined Predictors and Tools

In this section, we describe i) the metrics that we used as inputs to the prediction algorithms, and ii) the tools used for this research.

1) *Examined Predictors*: We used a set of software metrics that are typically used to characterize classes in class diagrams as input to our classification algorithms. SDMetrics<sup>1</sup> is capable of computing 32 types of class diagram metrics. These metrics are divided into five categories namely Size, Coupling, Inheritance, Complexity and Diagram. These classes of metrics were selected for the following reasons: i) our earlier survey [29] [31] showed that developers use Size and Coupling as predictors for key classes, ii) experts in the area of software metrics (Briand et al. [27] and Genero et al. [24]) stated that Coupling is an important structural dimension in object-oriented design, iii) the work in [14] and [12] showed that WMC (a metric in the Size category) and CBO (a metric in the Coupling category) are influential for defect prediction. The specific set of 11 metrics used is shown in Table II.

2) *Tools*: The tools used in this study are the following:

- Reverse Engineering Tool : MagicDraw<sup>2</sup> is a system modeling tool that provides reverse engineering facilities. MagicDraw version 17.0 (academic evaluation license) was used for this study.
- Software Metrics Tool : SDMetrics is a tool that computes a large set of metrics for UML models. SDMetrics version 2.2 (academic license) was used for this study.
- Data Mining Tool : Waikato Environment for Knowledge Analysis (WEKA) is a collection of machine learning algorithms for data mining tasks [16]. It contains tools for data pre-processing, classification, clustering, and visualization. WEKA version 3.6.6 was used for this study.

### B. Case Studies

We used the following criteria for selecting case studies:

- Open source software/system project that provides implementation source code and forward design class diagram.

<sup>1</sup>www.SDMetrics.com

<sup>2</sup>www.nomagic.com

Table II  
LIST OF CLASS DIAGRAM METRICS

Metrics	Category	Description
NumAttr	Size	The number of attributes in the class.
NumOps	Size	The number of operations in the class. Also known as WMC in [13] and Number of Methods (NM) in [25].
NumPubOps	Size	The number of public operations in a class. Also known as Number of Public Methods (NPM) in [25].
Setters	Size	The number of operations with a name starting with 'set'.
Getters	Size	The number of operations with a name starting with 'get', 'is', or 'has'.
Dep_Out	Coupling (import)	The number of dependencies where the class is the client.
Dep_In	Coupling (export)	The number of dependencies where the class is the supplier.
EC_Attr	Coupling (export)	The number of times the class is externally used at attributes type. This is a version of OAEC +AAEC in [26].
IC_Attr	Coupling (import)	The number of attributes in the class having another class or interface as their type. This is a version of OAIC+AAIC in [26].
EC_Par	Coupling (export)	The number of times the class is externally used as parameter type. This is a version of OMEC+AMEC in [26].
IC_Par	Coupling (import)	The number of parameters in the class having another class or interface as their type. This is a version of OMIC+AMIC in [26].

- The amounts of classes in the implementation (source code) should be at least 50 classes.

Based on these criteria, nine open source software/systems were selected. In this projects we selected a forward UML class diagram from the documentation and then selected a matching version of the source code. The amount of classes in these case studies ranges from 59 to 903 (see Table III). The ratio between the number of classes included in the UML class diagram and the number of classes in the implementation (source code) spreads across a wide range: from 3 to 47%. This large range in characteristics of the input may be a difficulty for building a reliable classifier for our domain. For this reason, we focus on algorithms that will produce a score for a class concordant with the likelihood that it would be included in the UML diagram. This will allow a developer to vary the amount of classes included, i.e. the level of abstraction, by changing the threshold on the score.

We make the case studies used for this research available at [17] for future research and for validation of this study.

<sup>3</sup><http://argouml.tigris.org/>

<sup>4</sup><http://mars-sim.sourceforge.net/>

<sup>5</sup><http://java-player.sourceforge.net/>

<sup>6</sup><http://sourceforge.net/projects/jgap/>

<sup>7</sup><http://neuroph.sourceforge.net/>

<sup>8</sup><http://jpmc.sourceforge.net/>

<sup>9</sup><http://code.google.com/p/wro4j/>

<sup>10</sup><http://code.google.com/p/xuml-compiler/>

<sup>11</sup><http://code.google.com/p/maze-solver/>

Table III  
LIST OF CASE STUDY

No	Project	Total Classes in Source Code (S)	Total Classes in Design (D)	D:S ratio as %
1.	ArgoUML <sup>3</sup>	903	44	4.87
2.	Mars <sup>4</sup>	840	29	3.45
3.	JavaClient <sup>5</sup>	214	57	26.64
4.	JGAP <sup>6</sup>	171	18	10.52
5.	Neuroph 2.3 <sup>7</sup>	161	24	14.90
6.	JPMC <sup>8</sup>	121	24	19.83
7.	Wro4J <sup>9</sup>	87	11	12.64
8.	xUML <sup>10</sup>	84	37	44.05
9.	Maze <sup>11</sup>	59	28	47.45

### C. Process

This subsection describes the steps performed for this study. The inputs for this process are forward designs and reverse engineered class diagrams (constructed from source code of the case studies). The output of the machine learning phase is the list of key classes that can be used to condense a class diagram. The approach is illustrated in Figure 1.

1) *Data Preparation*: For data preparation, class diagram metrics were extracted from the reverse engineered class design (obtained using reverse engineering with the Magic-Draw CASE tool). Then, the information about the presence of a class in the forward design was entered in a table.

The data preparation steps are described in Table IV. We expect that there is some noise in the predictors. For instance, the getters and setters predictor completely rely on conformance of source code to naming conventions (e.g 'get', 'has'). Not all case studies have this kind of naming convention. There is also the possibility that operations that are shown at a child class are actually inherited. In this case, the operation is counted twice. To study whether this noise has significant influence, we experimented with different sets of predictors: experiment A: the full set of predictors (predictor set A), experiment B: all metrics, but excluding metrics related to getters, setters and Number of Public Operation (predictor set B), and experiment C: set of predictors that only uses Coupling metrics (predictor set C). The details of all predictor sets are shown in Table V.

2) *Data Processing and Analysis of Results*: For every training and test activity for the classification algorithm, we randomly split every single predictor set (for every case study) into 50% for the training set and the other 50% for the test set. The main reason for doing this is that the data are highly unbalanced where the number of classes in design (the 'positives') is very low compared to the number of classes in the source code. If we would have used 10 fold cross validation, it means that we use only 10% of the data for testing and 90% for training. Thus, the possibility of any positives to be included in the test data was very low, and test set error measurements would not have been reliable. To further improve reliability, we ran each experiment 10 times using different randomization.

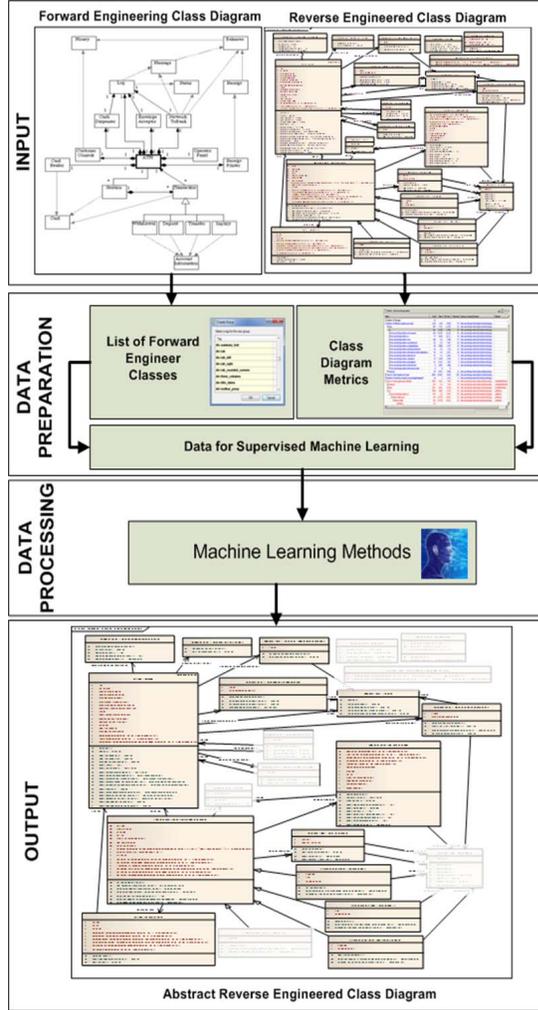


Figure 1. Design Abstraction Process

We assumed a fully automated method and that our end users have no knowledge of data mining. We also wanted to test for robustness under a selected threshold. Algorithms ran with default WEKA configuration, except for k-Nearest Neighbor, for which we used two different neighborhood size settings (1 and 5 neighbors).

## VI. EVALUATION OF RESULTS

This section presents our evaluation on i) predictive power of predictors and ii) overview of benchmark AUC results.

### A. Predictor Evaluation

This subsection presents our univariate analysis results that measure the predictive performance of each predictor using information gain.

**RQ1.** The results show the influence of a predictor for the classification algorithm. A class diagram metric is considered to be influential for prediction when the value of

Table IV  
DATA PREPARATION STEPS

No	Preparation Step	Description
1.	List all the classes that appear in the UML design-class diagram	Input to get the class in design vs. implementation ratio
2.	Reverse engineer the source code into a class diagram using MagicDraw. Save the class diagram in XML Metadata Interchange (XMI) file format	To get the design from the source code prepared for the metrics tool input
3.	Calculate the software metrics on the reverse engineered class diagram using SDMetrics and save in CSV format	Class diagram metrics calculation and data mining input preparation
4.	Manually remove external library classes and runtime classes from the list	To extract only developed classes in the source code
5.	Merge the software metrics information from the source code and the classes in the forward design	To map between classes in design and classes from software metrics obtained from the source code
6.	Amend the CSV file by adding the information of "In Design" properties (N for not presented in Design Document, Y for presented in Design Document)	Add the information about the class in design in the software metrics information
7.	Remove software metrics properties that show no significant information (data cleanup) present an overall data summary in plot graph	To extract only the selected independent variable (class diagram metrics) and present the summary of data

Table V  
PREDICTOR SETS

No	Predictor	Predictor set A	Predictor set B	Predictor set C
1	NumAttr	Yes	Yes	No
2	NumOps	Yes	Yes	No
3	NumPubOps	Yes	No	No
4	Setters	Yes	No	No
5	Getters	Yes	No	No
6	Dep_out	Yes	Yes	Yes
7	Dep_In	Yes	Yes	Yes
8	EC_Attr	Yes	Yes	Yes
9	IC_Attr	Yes	Yes	Yes
10	EC_Par	Yes	Yes	Yes
11	IC_Par	Yes	Yes	Yes

Project	NumAttr	NumOps	NumPubOps	Setters	Getters	Dep_out	Dep_In	EC_Attr	IC_Attr	EC_Par	IC_Par
ArgoUML	0.000	0.000	0.000	0.000	0.000	0.024	0.000	0.000	0.000	0.000	0.000
Mars	0.000	0.013	0.017	0.011	0.025	0.000	0.047	0.037	0.000	0.031	0.000
JavaClient	0.093	0.048	0.044	0.000	0.050	0.215	0.093	0.000	0.183	0.092	0.225
JGAP	0.073	0.056	0.000	0.078	0.000	0.047	0.000	0.000	0.000	0.058	0.000
Neuroph	0.000	0.054	0.062	0.000	0.000	0.000	0.084	0.000	0.000	0.106	0.000
IPMC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.859	0.000	0.000
Wro4J	0.000	0.000	0.000	0.000	0.000	0.000	0.212	0.111	0.000	0.196	0.000
xUML	0.168	0.281	0.281	0.306	0.147	0.240	0.000	0.000	0.085	0.000	0.506
Maze	0.000	0.000	0.000	0.000	0.000	0.000	0.171	0.178	0.000	0.125	0.000

Figure 2. Univariate Predictor Performance (Information gain)

the InfoGain Attribute Evaluator is greater than 0. Figure 2 shows that out of eleven class diagram metrics used in this study, nine of them influenced the prediction in the JavaClient project while xUML and Mars have eight and seven influential class diagram metrics. On the other

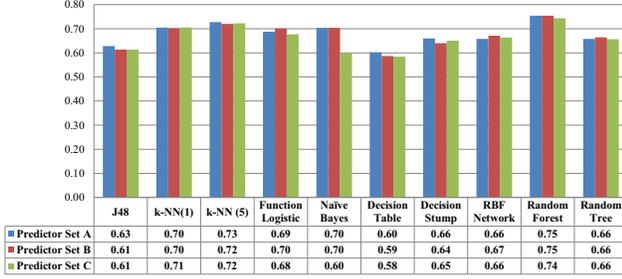


Figure 3. Average AUC Score for Every Data set

hand, ArgoUML and JPMC have only one influential class diagram metric. Figure 2 also shows that the Coupling category metrics are influential for every case study. For all cases, at least one of the Coupling metrics is listed as influential for prediction. This means that class diagram metrics categorized in Coupling (i.e. IC\_Par, EC\_Par, IC\_Attr, EC\_Attr, Dep\_In and Dep\_out) have a strong influence on prediction ability. If we compare Coupling metrics with Size metrics (i.e. NumAttr, NumOps, NumPubOps, Getters, Setters) we find that only five case studies list at least one of the Size-metrics as influential predictor. EC\_Par is the most influential class diagram metrics because it is listed as influential in prediction for six out of nine case studies.

**RQ2.** We have studied the predictors through three different experiments (as defined in Table V). Figure 3 shows the average AUCs of classification algorithms for all experiments. We expected to see a large difference in prediction performance among the three experiments. However, there is not much difference in prediction performance as we can see in Figure 3 the difference in average AUC is only  $\pm 0.02$ . From this figure, we found out that the performances slightly degrade for experiment C but the amount of degradation is not very significant. This means, even though the number of predictors in experiment C is smaller than in experiments A and B, the set of predictors is still reliable for prediction purposes. This shows that the Coupling category strongly influences the prediction performance.

### B. Benchmark Scoring Results

**RQ3.** The classification algorithms were evaluated by measuring the average and standard deviation of the AUC over ten runs for each predictor set. Table VI shows an example of results for experiment C. We have highlighted those cells that contain very weak classification results, i.e.  $AUC < 0.60$ . Note that an AUC of 0.50 means that the classifier produces completely random result. We decided a value of AUC of 0.60 or higher indicates a useful algorithm. This means, the classification algorithms that are able to produce this score for almost all case studies for all experiments are considered suitable for classifying key classes.

After running all the experiments, we found that the Random Forest and K-Nearest Neighbor (k-NN(5)) algorithms

Table VI  
RESULTS FOR PREDICTOR SET C

Project	J48	k-NN(1)	k-NN(5)	Function Logistic	Naive Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
ArgoUML	0.50	0.69	0.69	0.54	0.50	0.50	0.55	0.50	0.64	0.60
	0.00	0.04	0.05	0.05	0.00	0.00	0.07	0.07	0.04	0.03
Mars	0.53	0.69	0.75	0.61	0.73	0.52	0.70	0.58	0.73	0.61
	0.06	0.03	0.05	0.05	0.09	0.05	0.07	0.16	0.09	0.08
JavaClient	0.76	0.83	0.86	0.81	0.82	0.78	0.75	0.80	0.86	0.81
	0.09	0.03	0.04	0.05	0.02	0.07	0.06	0.04	0.04	0.05
JGAP	0.54	0.60	0.62	0.67	0.51	0.51	0.59	0.65	0.72	0.60
	0.07	0.05	0.07	0.12	0.02	0.02	0.04	0.10	0.10	0.17
Neuroph	0.61	0.79	0.82	0.71	0.53	0.56	0.63	0.72	0.78	0.68
	0.14	0.06	0.06	0.10	0.07	0.09	0.08	0.16	0.09	0.08
JPMC	0.54	0.66	0.67	0.69	0.50	0.50	0.58	0.61	0.69	0.59
	0.08	0.06	0.06	0.08	0.00	0.01	0.03	0.06	0.09	0.08
Wro4j	0.63	0.70	0.68	0.77	0.62	0.62	0.70	0.69	0.74	0.68
	0.18	0.09	0.19	0.16	0.13	0.12	0.13	0.21	0.14	0.14
xUML	0.74	0.78	0.77	0.69	0.62	0.69	0.72	0.82	0.83	0.75
	0.06	0.07	0.06	0.12	0.11	0.10	0.06	0.04	0.06	0.06
Maze	0.67	0.61	0.64	0.60	0.57	0.58	0.63	0.60	0.70	0.59
	0.07	0.10	0.14	0.11	0.08	0.07	0.06	0.10	0.10	0.08
Average	0.61	0.71	0.72	0.68	0.60	0.58	0.65	0.66	0.74	0.66
	0.09	0.08	0.08	0.08	0.11	0.10	0.07	0.10	0.07	0.08

Note : The first row for each predictor set is the average AUC, the second row lists the standard deviation. Cells with  $AUC < 0.60$  are highlighted.

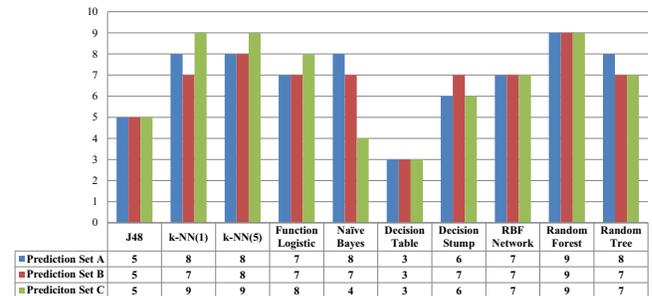


Figure 4. AUC Score  $\geq 0.60$

perform the best in classifying the key classes in a class diagram in terms of overall AUC as well as robustness over various predictor sets.

Figure 4 shows the prediction performance of all selected classification algorithm. This figure illustrates the number of case studies (for each predictor set) in which the classification algorithm produces an AUC score greater than 0.60. Random Forest and k-NN(5) perform the best prediction where both classification algorithm produced AUC score above 0.60 for at least 8 case studies for all datasets. Meanwhile, Random Tree, Function Logistic, RBF Network and Decision Stump perform less robust across all predictor sets. These classification algorithms performed reasonably well. They produced an AUC above the threshold for 6

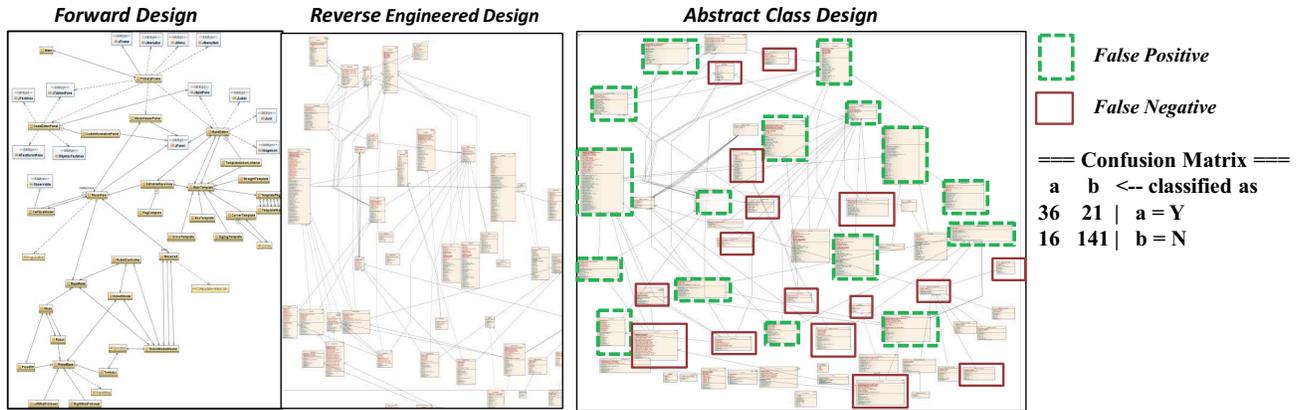


Figure 5. Application of Random Forest Classification Algorithm

to 8 case studies. Naive Bayes performed well for case studies using predictor set A and predictor set B but not using predictor set C. J48 and Decision Table appear not to be suitable to be used in these case studies, given the low number of results with  $AUC \geq 0.60$  (between 3 to 5). In addition, Figure 3 shows that the performance for Naive Bayes degrades tremendously when predictor set C is used for training even though the average AUC score for predictor set A and predictor set B are equal 0.70. The average AUC score of more than 0.72 for Random Forest and k-Nearest Neighbor (k-NN(5)) shows their suitability for all predictor sets.

Figure 5 illustrates the application of our method. In particular, it applies the Random Forest classification algorithm to the JavaClient case study. As result, a confusion matrix was generated. It shows that the total number of classes is 214 with 57 of the classes in the forward design. The generated confusion matrix shows that 36 out of 57 classes are correctly predicted as should be present in the class diagrams. Also 141 out of 157 classes are correctly predicted as should be omitted from the abstract class diagram. On the other hand, there are 21 false negatives (predicted as leave out, but should be included) and 16 classes that are false positives (predicted as ‘include’, but should not be included).

## VII. DISCUSSION AND FUTURE WORK

With this result, we can conclude that class diagram metrics from the Coupling and Size category can be good predictors for key classes in class diagrams. In summary, there are three class diagram metrics that should be considered as influential predictors: Export Coupling Parameter (EC\_Par), Dependency In (Dep\_In) and Number of Operation (NumOps). This means, the higher value of these metrics in a class may indicate that this class can be a candidate of an important class. Thus, newcomers in the project may use this information to manually predict the key classes when joining project. This finding is consistent

with [29] and [31] where the Number of Operation and Relationship (related to coupling) are the elements that are most software developer looked at in order to find the important classes in a class diagram.

The results show that k-NN(5) and Random Forest are suitable classification algorithms in this study. We took a step forward by exploring this classification algorithm by applying the algorithm individually to several case studies. As a result, some of the predicted True Positive in algorithm k-NN(5) are predicted False Positive in Random Forest and vice versa. We compared all the result manually from those two algorithms applied to several case studies and some of true and false results are different. The possibility to enhance this prediction power is by combining those classification algorithms to achieve the best result. Given the unbalanced data, the algorithms were not able to produce high AUC scores.

Basically, this study is expected to discover suitable classification algorithms which could provide a rank score concordant with the likelihood for classes to be included in the UML class diagram. Based on this result, we are able to produce an approach for ranking classes for importance. This will allow the software engineer to generate a UML diagram at different levels of detail. To construct the abstraction of the class diagrams, the software engineer may apply the abstraction of relationship in class diagrams as presented by Egyed [28].

This study was an early experimental benchmark, and we see a number of ways to extend this work. Alternative input parameters for predicting the key classes in a class diagram could be investigated. This could include the use of other type of design metrics for example based on (semantics of) the names of classes, methods and predictors. There are also possibilities to use source code metrics such as Line of Code (LOC) and Lines of Comments as additional predictors for the classification algorithms. Moreover, we could look at identification of features as unit of inclusion or exclusion in

the UML class diagram. Also, more extensive benchmarking should take place, for instance by learning models on one problem and testing it on another, or testing out an ensemble approach that combines classification algorithms. Specific approaches exist to better transfer knowledge across different problems, such as transfer learning.

Another approach to deal with limited availability of ‘ground truth’-data for validation is to use a semi supervised or interactive approach, where a user first selects some limited top level classes, then the system learns and recommends further classes to be included, and the user responds by confirming or rejecting recommendations. Building an interactive application may also help to guide future research.

In terms of predictive performance, it could be interesting to compare the result of this study with other approaches. This study uses the classes in the forward design as the ground truth. In version history mining the classes that are frequently changing are seen as candidates for key classes [9]. It is also interesting to compare our approach with other works that apply different algorithm such as HITS web mining (used in [7]), network analysis on dependency graphs (used in [30]) and PageRank [8], and provide guidelines in which cases which approach would be preferred, or to create hybrid approaches.

#### VIII. THREATS TO VALIDITY

This study assumed that all the classes that existed in the forward designs were the important classes. There is a possibility that some of these classes were not important or not the key classes of the system. Also, there is a possibility that the forward design used is too ‘old’ or in other words obsolete. Feedbacks from the system developer may enhance the accuracy of these key classes from forward design. However, collecting the feedback requires more effort.

The input of this study is dependent on the reverse engineered class diagram constructed by the MagicDraw CASE tools. As mentioned by Osman [5], there are several weakness of CASE tools’ reverse engineering features. This weakness may influence the accuracy of the class diagram metrics calculation. There is higher risk for large system that the CASE tool may leave out several information of some classes.

We only cover nine opensource case studies. Based on the amount of classes we can consider that the case studies represent of small to medium size project. The result may differ if we include large systems in our case studies. However, to get large open source systems that have forward design is really difficult.

#### IX. CONCLUSION

In this paper, we propose an approach for condensing reverse engineered class diagram by selecting the key classes in it. We study how well machine learning techniques

perform in selecting the key classes in a class diagram by using supervised learning methods. The machine learning algorithms are trained on a set of open source projects. These projects contain a forward design class diagram which is used as a reference for validating the quality of the condensation. Given the unbalanced nature of the data, Area under ROC curve is recommended as a performance evaluator for these algorithms.

This paper evaluates the influential predictors in classifying key classes and also compares various machine learning classification algorithms on nine case studies derived from open source software projects, to identify candidate algorithms with the most accurate as well as robust behavior across predictor sets. We discovered Export Coupling Parameter, Dependency In and Number of Operation are the most influential predictors for predicting key classes in a class diagram. On these predictor sets, Random Forest and k-Nearest Neighbor provided the best results. For all listed case studies, the Random Forest method scores an AUC above 0.64 and the average AUCs for every prediction set is 0.74. These algorithms are able to produce a predictive score that can be used to rank important classes by relative importance. Based on this class-ranking information, a tool can be developed that provides views of reverse engineered class diagrams at different levels of abstraction. In this was, developers may generate multiple levels of class diagram abstractions, ranging from highly detailed class diagram (equal to source code) to abstract class diagram (satisfying architect’s preference for high level views). In a broader perspective, this approach supports both the Bottom-Up and also the Top-Down approach for understanding of programs [32].

Indeed, based on the selected case studies, this approach could not produce 100% correct prediction of key classes in a class diagram. Hence, further research is needed to find complementary explanatory variables. We expect better results by taking the meaning of classes into account.

Finding the set of projects suitable for this study was a very time consuming task. This set can now be used by the scientific community as a benchmark for further studies.

#### ACKNOWLEDGMENT

This work is partly supported by Public Service Department of Malaysia. We would like to thank the CASE tools and software metrics providers for the licenses.

#### REFERENCES

- [1] A. Nugroho, M. R. V. Chaudron, “A Survey of the Practice of Design - Code Correspondence amongst Professional Software Engineers,” Proc. of the 1st Intl. Symp. on Empirical Softw. Eng. and Measurement(ESEM 2007), Sep. 2007, pp.467-469.
- [2] E.J. Chikofsky and J.H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Softw., vol. 294, Jan. 1990, pp. 13-17.

- [3] P. Tonella, M. Torchiano, B.D. Bois and T. Systä. "Empirical Studies in Reverse Engineering : State of the Art and Future Trends", *Empirical Softw. Eng.*, vol. 12, Oct. 2007, pp.551-571.
- [4] B. Bellay and H. Gall, "A Comparison of Four Reverse Engineering Tools," *Proc. of the 4th Working Conf. on Reverse Eng. (WCRE)*, Oct. 1997, pp. 2-11.
- [5] H. Osman, M.R.V. Chaudron, "Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Model," *GSTF Journal on Computing*, Vol. 2, Apr 2012, pp. 193-201.
- [6] A.M. Fernández-Sáez, M.R.V. Chaudron, M. Genero, and I. Ramos, "Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?," *Proc. of the 17th Intl. Conf. on Evaluation and Assessment in Softw. Eng. (EASE '13)*, 2013, pp. 60-71.
- [7] A. Zaidman, S. Demeyer, "Automatic Identification of Key Classes in a Software System using Webmining Techniques ," *J. Softw. Maint. Evol.: Res. Pract.*, John Wiley & Sons, vol. 20, pp. 387-417.
- [8] F. Perin, L. Renggli, and J. Ressia. "Ranking software artifacts." 4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010), 2010.
- [9] M. Hammad, M.L. Collard and J.I. Maletic, "Measuring Class Importance in the Context of Design Evolution," *IEEE 18th Intl. Conf. Prog. Comprehension (ICPC)*, July 2010, pp. 148-151.
- [10] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, Andreas, "Mining Version Histories to Guide Software Changes," *Proc. of the 26th Intl. Conf. on Softw. Eng. (ICSE '04)*, 2004, pp. 563-572.
- [11] J. Singer, "NavTracks: Supporting Navigation in Software," *Proc. of the 21st IEEE Intl. Conf. on Softw. Maint. (ICSM'05)*, 2005, pp. 325-334.
- [12] T. Gyimothy, R. Ferenc and I. Siket, "Empirical Validation of Object-Oriented metrics on Open Source Software for Fault Prediction," *IEEE Trans. Softw. Eng.*, vol.31, Oct. 2005.
- [13] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. on Softw. Eng.*, vol. 20, Jun 1994, pp. 476-493.
- [14] Z. Yuming and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Trans. on Softw. Eng.*, vol.32, Oct. 2006, pp. 771-798.
- [15] P.v.d. Putten and M. van Someren, "A Bias-Variance Analysis of a Real World Learning Problem: The CoIL Challenge 2000," *Kluwer Academic Publishers*, Oct. 2004, vol. 57, pp. 177-195.
- [16] I.H. Witten, E. Frank and M.A. Hall, "Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)," *Morgan Kaufmann*, Jan. 2011
- [17] Datasets, <http://www.liacs.nl/~hosman/DataSets.rar>
- [18] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G.J. McLachlan, A.F.M. Ng, B. Liu, P.S. Yu, Z.-H. Zhou, M. Steinbach, D.J. Hand, and D. Steinberg, "Top 10 Algorithms in Data Mining," *Knowl. Inf. Syst.*, vol. 14, Dec. 2007, pp. 1-37.
- [19] A. Field, *Discovering Statistics Using SPSS Third edition*, SAGE Pub., 2009
- [20] M. Orr, "Introduction to radial basis function networks. Tech. report, Institute for Adaptive and Neural Computation," *Edinburgh Univ.*, 1996, <http://www.anc.ed.ac.uk/#mjo/rbf.html>.
- [21] L. Breiman, "Random Forests," *Mach. Learn.*, Kluwer Academic Publishers, vol. 45, Oct. 2001, pp. 532.
- [22] J.A. Hanley and B.J McNeil. *The Meaning and use of Area Under a Receiver Operating Characteristic(ROC) Curve*, *Diagnostic Radiology*, vol. 143, Apr. 1982
- [23] A. Dallal and L.C. Briand, "A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes," *ACM Trans. Softw. Eng. Methodol.*, vol.21, Mar.2012.
- [24] M. Genero, M. Piattini and C. Calero, "A Survey of Metrics for UML Class Diagram," *Journal of Object Technology*, vol. 4, Nov. 2005, pp. 59-92.
- [25] A. Lake, and C.R. Cook, "Use of Factor Analysis to Develop OOP Software Complexity Metrics," *Tech. Report*, Oregon State Univ., 1994.
- [26] L. Briand, W. Melo and P. Devanbu, "An Investigation into Coupling Measures for C++," *Proc. of the 19th Intl. Conf. on Softw. Eng. (ICSE '97)*, ACM, 1997, pp. 412-421.
- [27] L.C. Briand, J. Wüst, S.V. Ikonovskii and H. Lounis, "Investigating Quality Factors in Object-oriented Designs: an Industrial Case Study," *Proc. of the 21st Intl. Conf. on Softw. Eng. (ICSE '99)*, ACM, 1999, pp. 345-354.
- [28] A. Egyed, "Automated Abstraction of Class Diagrams," *ACM Trans. on Softw. Eng. and Methodol.*, vol. 11, Oct. 2002, pp. 449- 491.
- [29] H. Osman, D.R. Stikkorum, A.v. Zadelhoff and M.R.V. Chaudron, "UML Simplification : What is in the developers mind?," *Proc. of the 2nd Workshop on Experiences and Empirical Studies in Softw. Modelling (EESMOD '12)*, ACM, 2012.
- [30] D. Steidl, B. Hummel and E. Juergens, "Using Network Analysis for Recommendation of Central Software Classes," *Proc. of the 19th Working Conf. on Reverse Eng. (WCRE)*, IEEE, Oct. 2012, pp. 93-102.
- [31] H. Osman, A.v. Zadelhoff and M.R.V. Chaudron, "UML Class Diagram Simplification : A Survey for Improving Reverse Engineered Class Diagram Comprehension," *Proc. of the 1st Intl. Conf. on Model-Driven Eng. and Softw. Dev. (MODEL-SWARD 2013)*, Feb 2013, pp. 291-296.
- [32] M.-A.D. Storey, F.D. Fracchia, and H.A. Mueller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization," *Proc. of the 5th Intl. Workshop on Prog. Comprehension (WPC '97)*. IEEE, 1997
- [33] Waikato Environment for Knowledge Analysis (WEKA), <http://www.cs.waikato.ac.nz/ml/weka/>

# Output-oriented Refactoring in PHP-based Dynamic Web Applications

Hoan Anh Nguyen, Hung Viet Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen  
 Electrical and Computer Engineering Department  
 Iowa State University  
 Email: {hoan,hungnv,tung,tien}@iastate.edu

**Abstract**—Refactoring is crucial in the development process of traditional programs as well as advanced Web applications. In a dynamic Web application, multiple versions of client code in HTML and JavaScript are dynamically generated from server-side code at run time for different usage scenarios. Toward understanding refactoring for dynamic Web code, we conducted an empirical study on several PHP-based Web applications. We found that Web developers perform a new type of refactoring that is specific to PHP-based dynamic Web code and pertain to output client-side code. After such a refactoring, the server-side code is more compact and modular with less amount of embedded and inline client-side HTML/JS code, or produces more standard-conforming client-side code. However, the corresponding output client-side code of the server code before and after the refactoring provides the same external behavior. We call it *output-oriented refactoring*. Our finding in the study motivates us to build WebDyn, an automatic tool for dynamicalizing refactorings. When performing on a portion of server-side code (which might contain both PHP and embedded/inline HTML/JS code), WebDyn detects the repeated and varied parts in that code portion and produces dynamic PHP code that creates the same client-side code. Our empirical evaluation on several projects showed WebDyn’s accuracy in such automated refactorings.

**Keywords**—refactoring; output-oriented; dynamic Web

## I. INTRODUCTION

Refactoring refers to source code restructuring for better software quality. It is important during the development of traditional programs as well as advanced Web applications. In a dynamic Web program, multiple versions of client-side code in HTML and JavaScript (JS) are dynamically generated at run time from server code written in a host language, e.g. PHP, for different usage scenarios. Client-side program entities are often embedded in string literals and/or computed via several statements in server-side code.

While several existing tools support refactoring on the traditional programs, there is still limited automated refactoring support for dynamic Web code. Toward studying refactoring in dynamic Web code, we have conducted an empirical study on four PHP-based Web applications. We manually checked a total of 2,664 revisions. We have found that developers have performed a special kind of refactoring that is very specific to dynamic Web programs. After such a refactoring, the server-side code is more compact and modular with less amount of embedded and inline client-side HTML/JS code, or produces more standard-conforming client-side code. However, the corresponding output client-side code of the

server code before and after the refactoring provides the same external behavior. For example, they often replace a portion of HTML/JS code embedded or inlined within PHP server code with new PHP code that produces the same client-side code. They also replace a long fragment of server code in both PHP and inline HTML/JS with more dynamic code in PHP that creates the same client-side code. Another popular type of refactoring occurs when developers refactor client-side code that is *embedded* within PHP strings. For example, to make their HTML/JS code conform to Web code standards [25], developers change a tag name or add/delete opening/closing tags in HTML code embedded in a PHP string. We call those operations *output-oriented refactorings*.

In total, we found 11 output-oriented refactoring operations, which are classified into 5 categories depending on their purposes: 1) dynamicalization (e.g. replacing inline HTML/JS code with a PHP fragment or function), 2) re-structuring server-side and client-side code (e.g. extracting and moving server code and inline JS code), 3) renaming embedded HTML/JS elements, 4) standardizing embedded HTML code (e.g. adding proper tags), and 5) refactoring for separation of concerns (e.g. separating JS code for control logic from HTML code for presentation). Our finding calls for automated tools to support output-oriented refactorings.

We also introduce WebDyn, a refactoring tool that supports dynamicalization refactorings. After a user selects a code portion in a PHP file (which might contain both PHP and embedded HTML/JS code), it will check the pre-condition of the dynamicalization refactoring such as whether it is syntactically correct and contains repetitive code. If the pre-condition holds, WebDyn will analyze the selected portion of code in order to partition and parameterize it. Then, it produces the resulting dynamic PHP code based on the detected partition and its parameterization. In our prior work [16], [17], we have developed a tool for embedded code renaming and standardization refactorings. Refactorings for re-structuring server and client code and for separations of concerns will be parts of our future work. Our empirical evaluation on real-world projects showed that WebDyn achieves 100% accuracy in automatic dynamicalization refactorings. Our key contributions include:

1. An empirical study that motivates tool support for output-oriented refactoring operations in dynamic Web applications,
2. WebDyn, an automated output-oriented refactoring tool,
3. An empirical evaluation to show WebDyn’s accuracy.

```

a) File modules/projects/addedit.php at rev 93
210 <select name="StartMM_int" size="1">
211 <OPTION VALUE="1">
  <?php if(@date("m", $start_date) == 1){?>
    selected<?php ?>>Jan
212 <OPTION VALUE="2">
  <?php if(@date("m", $start_date) == 2){?>
    selected<?php ?>>Feb
...
221 <OPTION VALUE="11">
  <?php if(@date("m", $start_date) == 11){?>
    selected<?php ?>>Nov
222 <OPTION VALUE="12">
  <?php if(@date("m", $start_date) == 12){?>
    selected<?php ?>>Dec
223 </select>

b) File modules/projects/addedit.php at rev 94
58 $months = array("Jan", "Feb", "Mar", "Apr",
  "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
...
208 <?php echo arraySelect($months,
  "StartMM_int", 'size=1',
  @date("m", $start_date)); ?>

c) File includes/main_function.php
36 function arraySelect(&arr, $name, $attrs,
  $selected) {
37     reset($arr);
38     $s = "<select name=\"$name\" $attrs>";
39     while(list($k, $v) = each($arr)) {
40         $s .= '<option value="' . $k . "'
  ($k == $selected ? ' selected' : '')
  . '>' . $v;
41     }
42     $s .= "</select>";
43     return $s;
44 }

```

Fig. 1. Making embedded HTML code become dynamic

## II. MOTIVATING EXAMPLES

Let us describe a few examples motivating our approach. The first example is from a Web page in dotProject [23]. The page allows users to add or edit the information on a new or existing project such as its name, start date, target/actual end date, target/actual budget, status, and description.

At revision 93, the server-side PHP code that generates the drop-down list for selecting the start month is shown in Figure 1a (lines 210-223). The drop-down list is specified via the HTML `<select>` element. Each choice of month in the list is specified via an HTML `<option>` element (lines 211-222). The attribute value contains the month data (1, 2, etc.) that will be submitted to the server. The text following the opening `<OPTION>` tag (Jan, Feb, etc.) will be displayed in the users' browsers.

In dotProject, drop-down lists are widely used and appear in many places for selecting different types of values such as days, months and years. Since the HTML structures for drop-down lists are standard, the PHP code portions generating them are very similar from place to place. Thus, function `arraySelect` was introduced to modularize the code generating drop-down lists (Figure 1c). This function takes as input an array, a name of the `<select>` tag, other attributes of the `<select>`, and a default selected value in the drop-down list. It will produce an HTML `<select>` element having the given name and attributes, and containing a drop-down list of `<option>`'s whose values and texts are the (index, value) pairs in the input array.

At revision 94, the code generating the `<select>` element for the start month in Figure 1 was replaced by a call to the function `arraySelect` (line 208, Figure 1b). The *static, inline HTML code portions* in the list of `<option>` elements at lines 211-222 (Figure 1a) were *replaced* by the *PHP code* with an array initialization at line 58 of Figure 1b and a while loop in the function `arraySelect` (lines 39-41, Figure 1c). The echo statements are used to print the string values to the output stream, which will be parsed by users' browsers. The resulting drop-down list is still the same, however, the corresponding HTML code is now **dynamically generated** via PHP.

This illustrates a popular and yet specific refactoring in dynamic Web applications in which a portion of static client-side code is replaced with a portion of server-side code that *dy-*

```

a) File index.php rev 22 to 23
19 <script ...> ...
32 function MailSelection() {
  ...
53 location.href =
  "mailto:" + addresses;
54 }
  ...
  getMailer()
55 </script>
  ...
169 echo "<td>
  <a href='mailto:$email'>...

b) File include/mailer.inc.php rev 23
2 $mailers['standard']="mailto:";
3 $mailers['gmail']="mail.google...";
4 $mailers['yahoo']="mail.yahoo...";
5 $mailers['hotmail']="hotmail...";
6 function getMailer() {
7     global $mailers;
8     if(isset($mailers[getPref('mailer')]))
9         return $mailers[getPref('mailer')];
10    else
11        return "mailto:";
12 }

```

Fig. 2. Replacing embedded client-side code with a function call

*namically generates* the same corresponding client-side code. We call this kind of refactoring **dynamicalizing**. Figure 1 shows such a dynamicalizing change to the HTML code for the start month. We also observed the similar dynamicalizing changes to the HTML code for the start day and year, and the end day/month/year (not shown).

Another kind of change at the revision 94 is that the tag name `OPTION` and attribute name `VALUE` were changed to lowercase. Since HTML is not case-sensitive, this does not change the semantic of the generated client code. However, it is recommended by W3C [25] that the tag names and attribute names should be in lowercase. Thus, this type of change makes the code conform to the standard while maintaining the same semantic. We call it *standardization refactoring*.

Our second example is taken from the main page of Address Book project [24]. This page uses `mailto` URI schema to create a `mailto` clickable link/button that allows users to activate the local mail client for composing a new message. At revision 23, to support other popular mailers, the function `getMailer` (line 6, Figure 2b) was introduced to get the mailer corresponding to the current configuration. Then, all the embedded client code "mailto:" were replaced with the calls to function `getMailer`.

**Discussions.** In a dynamic Web application, the client-side code is dynamically generated from the server-side code. The content of the client-side code in HTML/JS is embedded in PHP string literals and is `print'ed/echo'ed` to the output stream via executing a sequence of PHP statements. In addition to refactoring PHP code as in single-language code, developers also perform refactoring operations that pertain to the output client-side HTML/JS code. When a developer wants to refactor such client-side code, (s)he either (1) modifies/re-structures the corresponding generating server-side PHP code, or (2) directly modifies the values of PHP strings.

The examples of case (1) include dynamicalizing of static HTML code (Figure 1) and replacing a portion of embedded client-side code with a function call (Figure 2). The program structure and elements before and after the refactoring could be very different, e.g. a part of an inline HTML becomes a while loop with several echo and if statements. However, the corresponding output client-side code provides the same external behavior. These types of refactoring are specific to dynamic Web programming with server- and client-side code.

The refactoring that replaces embedded code portion with a function in Figure 2 is different in nature from the traditional method extraction. First, any portion of inline HTML/JS code (e.g. a portion of a PHP string literal) can be made dynamic regardless of whether it is syntactically well-formed, since it is the output of the execution on PHP code. In contrast, method extraction on a single-language program must be performed only on syntactically correct portions of code. Second, in traditional method extraction, the extracted code is mostly unchanged in the new method. In Web programs, the body of the new function has different code structures than the embedded code portion as in Figure 2, yet still produces the same output.

An example of case (2) is the capitalization of the tag and attribute names of the HTML `<option>` elements as in Figure 1. Since they are embedded as substrings in PHP literals, a refactoring tool must consider the syntax and semantics of the corresponding client code, which might appear as valid HTML code only at run time.

In brief, before and after such refactoring, the corresponding output client-side code of the server code provides the same external behavior. We call them **output-oriented refactoring**.

### III. IMPORTANT CONCEPTS

The above examples illustrate the following important concepts used in our work on dynamic Web code refactoring.

*Definition 1: A **Server Code Fragment** is a continuous sequence of well-formed program elements in the server code that is executed on the server side.*

The lines 38-42 of Figure 1c form a server code fragment. All PHP statements in this fragment are well-formed: a while loop and the statements in its body, and two assignments.

*Definition 2: A **Client Code Fragment** is a continuous sequence of well-formed program elements in the generated client-side code that is executed on the client’s browser(s).*

For example, the generated code of the HTML `<select>` element from Figure 1a is an HTML client code fragment.

*Definition 3: An **Embedded Client Code Fragment** is a portion of client code embedded in a string literal or in an inline client code fragment within server code.*

For example, PHP code for the opening tag of the HTML `<select>` element at line 210 in Figure 1a is an embedded client code fragment since it is a portion of the inline HTML code in the server page `addedit.php`. The sub-strings `'select'` and `'name'` at line 38 of Figure 1c are embedded client code fragments since they are portions of a string literal written in PHP code. From the definition and examples, note that 1) an embedded client code fragment is embedded in server code, and 2) it might contain a syntactically ill-formed program element in HTML or JS, i.e. any portion of embedded client code can be an embedded client code fragment, regardless of its well-formedness with respect to HTML and JS languages.

*Definition 4: A **Source Slice**  $S$  is a list of server code fragments and/or embedded client code fragments that generate a client code fragment (i.e. a continuous portion of client code). The client code that  $S$  generates is denoted by  $Output(S)$ .*

TABLE I  
SUBJECT SYSTEMS

System	Tot. Revs	Ch. Revs	Files	LOC	Avg.
AddressBook	467	1-467	103	19K	184
DotProject	5,591	1-530	769	201K	261
PhpFusion	2,713	1,800-2,100	795	91K	114
PostfixAdmin	1,367	1-1,367	252	50K	198

We use  $Output(S) \simeq Output(R)$  (“equivalent”) to denote that the client code of  $S$  and  $R$  has the same behavior.

A source slice consists of one or multiple fragments. In line 211 of Figure 1a, the HTML code for the `<OPTION>` element is generated by a source slice that consists of three fragments: an inline HTML (`<OPTION VALUE="1">`), an `if` statement, and another inline HTML (`Jan`). A source slice is not necessarily continuous. For example, lines 58 and 208 in Figure 1b form a source slice since they produce an HTML element `<select>` that is the same as the output of the source slice in Figure 1a.

*Definition 5: An **Embedded Identifier** of a client-side program element is a location in the server code corresponding to the definition or a reference of that element in the client-side code. When the context is clear, it is called either **embedded declaration** or **embedded reference**.*

We are interested in three kinds of client-side elements: HTML elements, JS functions and variables; all can be defined and referred to via names. For instance, `StartMM_int` (line 210, Figure 1b) is an embedded identifier since it is used to generate the name of an HTML `<select>` element.

### IV. AN EMPIRICAL STUDY

To learn more on output-oriented refactoring, we conducted an empirical study on several real-world Web applications. We aimed to answer the following research questions:

- Q1. What types of output-oriented refactoring operations were performed in a dynamic Web application?
- Q2. What is the frequency of each refactoring operation?

#### A. Subject Systems and Experimental Setup

We collected many revisions of four open-source PHP-based Web applications from `sourceforge.net` (Table I). Column `Tot. Revs` shows the number of revisions in the history of each subject system. The last three columns show the numbers of PHP server files and lines of code, and the average numbers of lines of code per file for the latest revision of a system. For each system, we used TortoiseSVN differencing tool to examine all changes that were made between consecutive revisions during its evolution. Of all changes, we manually identified those that are output-oriented refactoring operations, in which server code is modified but the respective client code of the PHP code portions involved in the change provides the same behavior. In total, we manually checked 2,664 revisions.

#### B. Categories of Output-oriented Refactoring Operations

In our result, we were able to confirm the existence of 11 output-oriented refactoring operations in PHP-based dynamic Web applications: after such a refactoring, the corresponding

output client code provides the same external behavior. We classified them into 5 categories depending on their purposes. Let us describe them first and their numbers in the systems.

**1. Dynamicalization Refactoring Operations.** The refactoring operations of this category change the way client-side code is generated, thus, could improve the maintainability of an application. Importantly, the output client-side code has the same behavior. For example, one could replace the inline HTML code by a portion of PHP code that generates the same HTML code. Or, one could consolidate the server-side duplicate code by adding a parameterized function and replacing the portions of such duplicate code by the invocations of that function. The followings will define those refactoring operations.

*Refactoring 1: Dynamicalizing refactoring operation replaces a source slice containing repeated code (i.e. generating similar client code fragments) by another source slice that generates the same output, however, with less embedded code.*

**Input:** A code portion  $S$  in the server side  
**Pre:**  $\text{isSourceSlice}(S) \wedge \text{hasRepetition}(S)$   
**Output:** A source slice  $T$  is added,  $S$  is removed  
**Post:**  $\text{Output}(T) = \text{Output}(S) \wedge \text{lessInlineCode}(T, S)$

An example of this refactoring is in Figure 1: twelve similar code fragments for generating the `<options>`s for 12 months are replaced by one while loop at lines 39-41 of Figure 1c. *Dynamicalization* refactoring is used to improve code maintainability. For example, in Figure 1a, if one wanted to add the missing closing tag `</option>`s, (s)he would have to touch 12 lines of code. However, in Figure 1c, this task can be done by simply concatenating `'</option>'` to the end of line 40.

*Refactoring 2: Replacing with Function Call replaces embedded code by a function call generating the same output.*

**Input:** A server-side code portion  $C$  and function  $F$   
**Pre:**  $\text{isEmbeddedCode}(C) \wedge \text{exists}(F)$   
**Output:** A call to function  $F$  is added,  $C$  is removed  
**Post:**  $\text{Output}(F) = C$

The pre-condition for this operation is that the selected portion  $C$  in the server code must be an embedded client code fragment (Definition 3) and function  $F$  exists. After this refactoring, a call to the function  $F$  is created to replace  $C$ .

This refactoring is similar to the traditional Method Extraction, however it differs in nature as explained in Section II.

**2. Extracting and Moving Refactoring Operations.** Developers generally use the refactoring of this category to re-structure the server code. To keep the external behavior unchanged, those refactorings are performed in a way that their output client-side code is the same after the refactorings.

*Refactoring 3: Server Code Extraction refactoring extracts a source slice in a server file into a new server file which is dynamically included from the old file at run time.*

**Input:** A server file  $F$  and a source slice  $S$   
**Pre:**  $\text{isSourceSlice}(S) \wedge (S \in F)$   
**Output:** A newly extracted file  $G$  and a changed file  $F'$   
**Post:**  $(\text{Output}(F') = \text{Output}(F)) \wedge (\text{Output}(G) = \text{Output}(S)) \wedge (G \supset S) \wedge (F' \text{ does not contain code in } G \text{ but has an include/require to } G)$

The post condition  $\text{Output}(F') = \text{Output}(F)$  is specified to

keep the generated client-side code, i.e. the external behavior of  $F$  to the users, unchanged.  $\text{Output}(G) = \text{Output}(S)$  guarantees that  $G$  is the file containing the extracted code  $S$ .

*Refactoring 4: JS Code Moving moves inline JS code within a `<script>` element (a JS embedded client-side code fragment) from one place to another one in the same file.*

**Input:** A selected `<script>` element  $J$  containing a piece of JS code, and a file  $S$  containing  $J$   
**Pre:**  $(J \in S) \wedge \text{containJSCode}(J)$   
**Output:**  $S$  becomes  $S'$  containing  $J$  at another place  
**Post:**  $\text{Output}(S) = \text{Output}(S')$

While server code extraction is used to re-organize server code itself, developers also want to re-structure client code that is embedded within the server one to improve software modularity. This JS code moving refactoring is used to re-structure JS control logic code in client-side code, e.g. moving JS control code closer to its associated HTML elements.

**3. Renaming Refactoring Operations.** Developers change the names of program elements to make them more suitable. Besides the renaming of PHP variables, we found the renaming of the embedded identifiers of HTML/JS elements including HTML elements (form, input), and JS variables or functions. This section explains only renaming of embedded HTML/JS elements since they are specific to dynamic Web code.

For example, one might want to change the name of the `<select>` element at line 210 in Figure 1 from `StartMM_int` to `StartMM`. In Figure 1a, this task involves parsing the embedded HTML code to recognize the value of attribute name in tag `<select>`. In Figure 1b, it also involves parsing the body of function `arraySelect` to understand that the value passed to the second parameter is the name of tag `<select>`.

*Refactoring 5: Client-side Element Renaming changes the name of a client-side program element and updates its embedded identifiers within the server code.*

**Input:** The client-side element  $o$  and a name  $n$   
**Pre:**  $\text{clientElement}(o) \wedge (\neg \exists o': o.type = o'.type, o.scope = o'.scope, o.name = n)$   
**Output:** The client-side element  $o$  has new name  $n$   
**Post:** All embedded identifiers of  $o$  are renamed to  $n$

**4. Standardization Refactoring Operations.** Developers use this category of refactorings to make the embedded client code conform to important coding standards/guidelines for Web languages. For example, a HTML page should conform to the W3C syntactical rules [25]. These refactorings make changes to embedded or inline client-side code within server code.

*Refactoring 6: Lowercase HTML Tag/Attribute Name changes the HTML tag name to lower-case.*

**Input:** An HTML tag/attribute name  $T$   
**Pre:**  $(\text{isHTMLTag}(T) \vee \text{isHTMLAttribute}(T)) \wedge \text{hasAnUppercaseChar}(T)$   
**Output:** A new HTML tag/attribute name  $N$   
**Post:**  $N = \text{lowercase}(T)$

An example of this refactoring is shown in Figure 1. The uppercase names of the tag `OPTION` and attribute `VALUE` in Figure 1a were changed to lowercase at line 40 of Figure 1c.

*Refactoring 7: Quoting HTML Attribute Value adds the enclosing quotation marks to an HTML attribute's value.*

**Input:** An HTML attribute value  $V$   
**Pre:**  $\text{isHTMLAttr}(T) \wedge (\neg T.\text{startWith}(\text{"'"}) \vee \neg T.\text{endWith}(\text{"'"}))$   
**Output:** A new HTML attribute value  $N$   
**Post:**  $N = \text{"}V\text{"}$

*Refactoring 8: Encoding Special Character replaces a special character in the textual content of HTML code with an encoded control string defined by HTML standard.*

**Input:** A special character  $c$   
**Pre:**  $\text{isSpecialChar}(c) \wedge \text{isInATextElement}(c)$   
**Output:** A string  $s$   
**Post:**  $s = \text{encodeString}(c)$

*Refactoring 9: Adding Closing Tag refactoring adds a proper HTML closing tag for a given opening tag.*

**Input:** An opening tag  $T$   
**Pre:**  $\neg \exists C: C = \text{isClosingTag}(T)$   
**Output:** A closing tag  $C$  is added  
**Post:**  $C = \text{isClosingTag}(T)$

For example, all opening `<OPTION>s` in Figure 1a missed closing `</OPTION>s`. They were fixed later in revision 530. Popular browsers can tolerate those syntactical errors. However, such missing could cause serious errors [16].

**5. Concern Separation Refactoring Operations.** General design guidelines recommend the principle of *separation of concerns*. In dynamic Web applications, the HTML code for *presentation content* (e.g. page structure or user interface elements) is intermixed with the JS code for *control logic* (e.g. input data validation or event handling), or intermixed with the CSS code for *presentation style* (e.g. color, font, size, etc), which violates that principle. We found several cases where developers made changes to separate those three concerns.

*Refactoring 10: Replacing HTML Style with CSS refactoring operation replaces the use of attribute style of an HTML element by that of an attribute class referring to the corresponding style defined in Cascading Style Sheets (CSS).*

**Input:** The style attribute  $s$  of an HTML element  
**Pre:**  $s.\text{style} \neq \text{""}$   
**Output:** The class attribute  $c$  of that HTML element  
**Post:**  $c.\text{class} = \text{css\_class} \wedge \text{isSamePres}(s, \text{css\_class})$

An example of this operation is: `<p style = "font-family:arial; color:red; font-size:20px;"> A paragraph.</p>`. A new CSS class is created to specify the font and color, and referred from the HTML code as in `<p class="new-css-class">...</p>`.

*Refactoring 11: Extracting Inline Event Handler refactoring operation extracts the JS code for event handling from the HTML code into a separate JS function.*

**Input:**  $\text{isEventHandler}(E) \wedge \text{isJSCode}(J) \wedge (E = \text{"}J\text{"})$   
**Pre:**  $J \neq \text{""}$   
**Output:** A JS function  $F$  and a declaration that  $F$  is the handler for the event  $E$   
**Post:**  $(J \in F) \wedge (E = \text{"}F\text{"})$

For example, a HTML form is defined as

```
1 <form name="search" onSubmit="if (document.search.keys.value != ")
  return true; return false ;" >...
```

To separate the HTML code for presentation from JS code for control logic, one can extract that event handling code into a separate function check and designate it as an event handler: `document.search.addEventListener('submit', check)`. The code is now more extensible as `check` can be used for other events.

After a refactoring in Categories 4 and 5, we have  $\text{Output}(S) \simeq \text{Output}(T)$  (i.e.,  $S$  and  $T$ , the code before and after a refactoring, are equivalent, but not necessarily the same).

### C. Results for Output-oriented Refactoring

In Table II, the columns from 1-11 show eleven types of output-oriented refactoring operations. For each type, we show both the counting number and percentage, as well as the total number. As seen, the most common category is Dynamicalization, which accounts for 65.2% of all output-oriented refactoring operations that were performed. In this category, Type 1—Dynamicalizing accounts for 62% of all the refactorings. Renaming and standardizing refactoring operations are also common. For example, in `AddressBook`, 17% of all refactorings are Quoting HTML attribute value (Refactoring 7); and in `PostfixAdmin`, 60% of the refactorings are Client-side Element Renaming (Refactoring 5).

**Threats to Validity.** We manually checked the changes in the subject systems, thus, human errors could occur. The chosen systems and revisions might not be representative. Thus, some output-oriented refactorings exist, however, were not found.

**Implication.** This result has an important implication. It suggests the need of automated tool support for this special output-oriented type of Web code refactoring.

To address that, we have developed `WebDyn`, an automatic refactoring tool that supports dynamicalization refactorings. Refactorings for Categories 2 and 5 will be parts of our future work, while our prior work [16], [17], supports renaming for embedded entities and standardization refactorings.

## V. OUTPUT APPROXIMATION AND PARSING

To support dynamicalization refactorings, `WebDyn` needs to determine the output of any portion of PHP server code and understand the semantics of the client-side HTML/JS code in that output. To determine the output of a PHP server page, we use `PhpSync` [16] to symbolically run a PHP program to create a tree-based representation, called *D-model*, which approximates all possible textual outputs of the program.

Figure 3 shows an example of a D-model, which represents the output of the PHP function `arraySelect` in Figure 1c. The `Concat` root node of the D-model represents a concatenation of multiple sub-strings, in which three are static string values (represented by string literal nodes), two are unresolvable values (represented by symbolic nodes), and one is a string value computed via a loop (represented by the `Repeat` node). D-model also contains `Select` nodes to represent alternative versions of a string output value. For example, the `Select` in this case represents two possible choices of having attribute selected in the tag `option` or not. During symbolically executing the PHP program, `PhpSync` records the mapping between each D-model node and its corresponding location in the PHP code.

TABLE II  
REFACTORING OPERATIONS FOUND IN THE SUBJECT SYSTEMS

System	Sum	Dynamicalization		Extract/Move		Rename	Standardizing				Concept Sep.	
		1	2	3	4	5	6	7	8	9	10	11
AddressBook (AB)	35	17	2	1	1	0	2	6	2	0	4	0
		49%	6%	3%	3%	0%	6%	17%	6%	0%	11%	0%
DotProject (DP)	130	80	5	4	0	12	21	1	0	1	5	1
		62%	4%	3%	1%	9%	16%	1%	0%	1%	4%	1%
PhpFusion (PF)	43	37	0	0	0	4	0	2	0	0	0	0
		86%	0%	0%	0%	9%	0%	5%	0%	0%	0%	0%
PostfixAdmin (PA)	10	1	0	1	0	6	0	0	2	0	0	0
		10%	0%	10%	0%	60%	0%	0%	20%	0%	0%	0%
All Systems	218	135	7	6	1	22	23	9	4	1	9	1
		62%	3.2%	2.8%	0.5%	10.1%	10.6%	4.1%	1.8%	0.5%	4.1%	0.5%

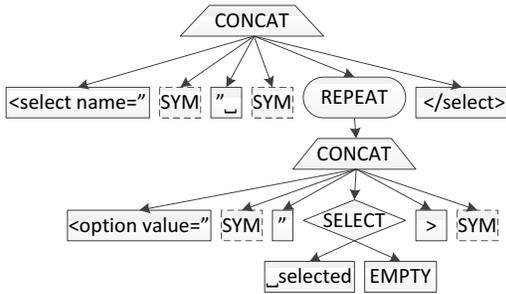


Fig. 3. D-model for the output of function arraySelect in Figure 1c

This mapping will be used to find the location(s) in the server code that generate a given portion of client-side code. Details on D-model and its node types can be found in [16].

After building the D-model, WebDyn uses a customized HTML parser [17] to identify HTML/JS elements in the output client code. This parser takes as input the D-model of given PHP code and produces the list of all HTML elements (with the attribute names and values) embedded in the literal nodes of the D-model. Those elements include the JS code contained in HTML `<script>` elements or event handlers (e.g. `onclick`, `onload`), which in turn are parsed by Eclipse’s JS parser.

## VI. DYNAMICALIZING REFACTORIZING ALGORITHM

**Overview.** Dynamicalization aims to reduce the repetitiveness of a source slice by transforming it into code with a loop that contains less embedded/inline code. Figure 4 shows our tool running on the example in Figure 1. As seen, the original source slice defines 12 repeated HTML options. Those option tags have the same structure including an embedded HTML code portion, a PHP code portion, and a text element. In those definitions, all are repeated, except for three places: the option values (ranging from “1” to “12”), the right operands of the comparisons in the PHP if statements (running from 1 to 12), and the text elements (“Jan”, “Feb”, etc). As seen, these 12 option tags are refactored into a for loop in which those values are produced via the iteration variable `$_index1` of that loop. The text elements, which are not computable from `$_index1`, are placed in an external array and retrieved using `$_index1`.

As seen in this example, WebDyn assumes the source slice under refactoring generate multiple similar HTML elements (e.g. the options in a selection box, the items in a list, the cells in a table). Each element is generated by a partition of the source slice. Those partitions have the same structure, and differ only at some places. Thus, the main task of WebDyn is to determine those partitions and the different places, and then to replace those partitions by the code with a loop and parameters to account for those differences.

Figure 5 illustrates the dynamicalization algorithm with four key steps. First, it parses the PHP source file *Fl* into an AST *T* (line 2) and transforms *T* so that embedded client-side code is replaced by PHP echo statements for easier processing later (line 3). Then, it checks whether the code portion *S* that the user selects is a source slice or not (line 4). If it is, WebDyn extracts the forest in *T* that corresponds to *S* (line 5) and performs the task of partitioning and parameterizing (line 6). The result of that task is transformed into a for loop and additional statements (e.g. a declaration of the parameter array) (line 8), and is then unparsed into the source code in place of the original *S* (line 9). Let us describe those steps in details.

**1) Step 1: Parsing and Transforming PHP code at the Token Level: (lines 11-19)** The purpose of this step has two folds. First, it facilitates the detection of repetitive code at the HTML token level during its partitioning and parameterization. Second, it supports the cases where the users make a code selection that covers parts of a string of a PHP literal node.

To achieve that, after parsing the PHP source file *Fl* into an AST tree *T*, WebDyn transforms *T* by replacing each literal node *l* of *T* (which might contain embedded client-side code) with a sequence of PHP echo statements (lines 11-19). To detect the repetition at the code token level, WebDyn tokenizes the content of the string literal in *l* (lines 16-17) and creates an echo statement for each token (line 18).

Let us take an example. Figure 7 shows the subtree for the following PHP code (the same as line 211, Figure 1):

```
<OPTION VALUE="1" <?php if(date("m", $start_date)==1){?>
selected<?php }?>>Jan
```

In Figure 7, the embedded code “<OPTION VALUE=“1”” is the content of a literal node in the AST, which is tokenized

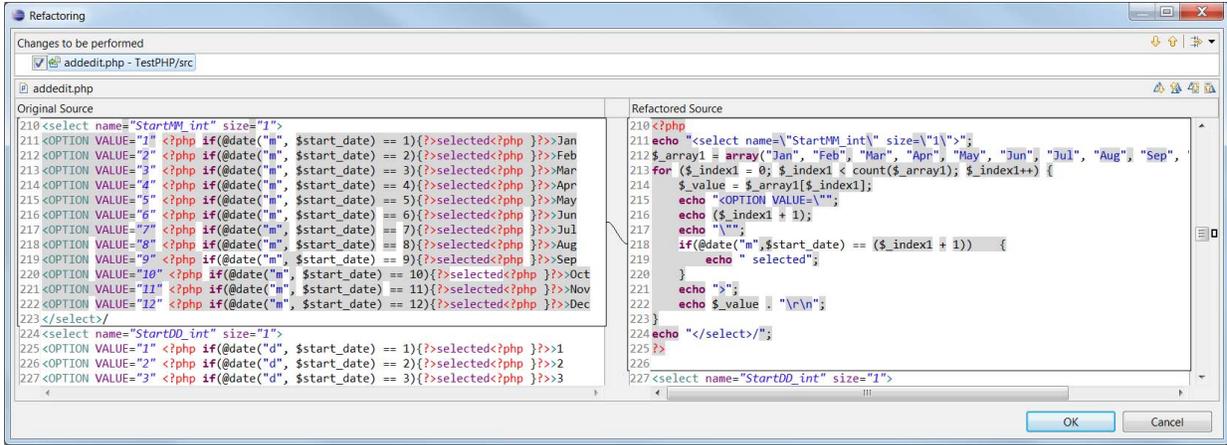


Fig. 4. WebDyn: Dynamicalization Refactoring Tool

```

1 function Dynamicalize(Program P, File Fl, Selection S)
2   AST T = BuildAST(Fl)
3   AST Tt = Transform(T, P)
4   if (!isSourceSlice(Tt, S)) return "Invalid selection"
5   AstForest F = GetAST(Tt, S)
6   Template template = PartitionAndParameterize(F)
7   if (template = EmptyTemplate) return "No repetition in selection"
8   ASTForest FD = GenAST(template)
9   FillInCode(Fl, S, FD)
10
11 function Transform(AST T, Program P)
12   Dmodel D = BuildDmodel(T, P)
13   HtmlTokenSequence H = Parse(D)
14   for each echo statement e in T
15     List<Literal> L
16     for each literal node l under e
17       L.Add(tokenize(l, H))
18     List<EchoStatement> S = CreateEcho(L)
19     Replace e with L
20
21 function isSourceSlice(AST T, Selection S)
22   Integer startChar = GetFirstNonWhiteSpaceChar(S)
23   Integer endChar = GetLastNonWhiteSpaceChar(S)
24   AstNode startNode = GetHighestAstNodeStartsAt(T, startChar)
25   AstNode endNode = GetHighestAstNodeEndsAt(T, endChar)
26   if (startNode.parent = endNode.parent) return true else return false
27
28 function GenAST(Template t)
29   List<AstNode> Placeholders = list of placeholder nodes in t.forest
30   Create an initialization of an array a
31   for i = 1 to t.numberofRepetitions
32     Create an array a1 holding all replacement values for repetition i
33     for j = 1 to Placeholders.size
34       a1.elements.Add(Placeholders[j].replacements[i])
35     a.elements.Add(a1)
36   for i = 1 to Placeholders.size
37     Replace Placeholders[i] with an array access expression a1[i]
38   Create a for statement f iterating a with a block containing t.forest
39   return a and f

```

Fig. 5. Output-oriented Dynamicalization Refactoring (1)

into six code tokens (e.g. “<”, “OPTION”, “VALUE”, etc). In Figure 8, six echo statements are created in the AST and their subtrees replace the original ones for the HTML code.

**2) Step 2: Validating User Selection as a Source Slice: (lines 21-26)** First, the leading/trailing whitespaces and comments in the user’s selection is discarded. If the remaining content  $S$  corresponds to a consecutive sequence of PHP

statements, then  $S$  is a valid source slice. To do this validation, WebDyn checks whether the starting and ending AST nodes of the remaining content of  $S$  have the same parent node.

**3) Step 3. Partitioning and Parameterizing Source Slices: (Figure 6)** In this step, WebDyn takes a sequence of consecutive AST sub-trees (a forest  $F$ ) and partition it into sub-sequences of AST sub-trees that have the same structure. It searches for the partition solution with the highest score. A score is defined as the product of the number of partitions and the number of dynamicalized tokens in embedded code (lines 45-46). The rationale is that we want to have refactored code with less embedded code, thus, we favor more partitions.

Since the repetition is often on the declarations of HTML elements, each partition will start at an opening HTML tag. Thus, in the selected AST forest, WebDyn first determines the node locations of all opening HTML tags (line 3). It processes each group of the tags of the same type individually (lines 4-7). For each group, it maintains a list  $L$  of the locations of the opening tags. A partition solution can be represented by a list of locations called *pivots*, which is a sub-list of  $L$ .

WebDyn searches for the best sub-list using a hill-climbing strategy (lines 17-34). That is, it stops when the current partition solution does not have higher score than the current best solution (lines 19-23). During searching, for each location  $Loc$  in  $L$ , it has two choices: marking it as a pivot, i.e. the start of a partition (lines 30-32) or not (lines 33-34).  $Loc$  is a pivot when either it corresponds to the first partition or its newly formed partition matches to the first partition (lines 24-29).

Two partitions are considered matched (lines 36-37) if their alignment score computed via a standard sequence alignment method is higher than a threshold. Function `Align` (lines 39-43) computes the alignment score of two sub-trees.

When a partition is selected (lines 27-28), its parameterized template and the prior partitions are updated (function `Update`). The leaf nodes with the values different from those of the corresponding nodes in the previous partitions are marked as place holders (line 53). Nodes with no correspondences in previous partitions are also marked as place holders (line 57).

```

1 function PartitionAndParameterize(AstForest F)
2 global Template template = EmptyTemplate
3 Map<Name, List<Integer>> OpenTagLocs = GetOpenTags(F)
4 for each tag name n
5 List<Integer> L = OpenTagLocs[n]
6 Stack Pivots.Push(0)
7 PartitionAndParameterize(F, L, Pivots, EmptyTemplate, 0)
8 return template
9
10 function GetOpenTags(AstForest F)
11 Map<Name, List<Integer>> OpenTagLocs
12 for i = 0 to F.size - 1
13 if AST F[i] is an echo statement  $\wedge$  F[i] contains opening tag t
14 OpenTagLocs[t.name].Add(i)
15 return OpenTagLocs
16
17 function PartitionAndParameterize(AstForest F, List<Integer> L,
18 Stack<Integer> Pivots, Template t, Integer loc)
19 if (loc = L.size - 1)
20 if (t.numberOfRepetitions > 1  $\wedge$  Score(t)  $\geq$  Score(template))
21 template = t
22 else stop;
23 else
24 Boolean matched = true
25 if (Pivots.size = 1)
26 t = new Template(F.SubForest(L[0], L[Pivots[loc+1]]))
27 if Match(t.forest, F.SubForest(L[Pivots.peek()], L[Pivots[loc+1]])  $\wedge$ 
28 Update(t, F.SubForest(L[Pivots.Peek()], L[Pivots[loc+1]]))
29 t.numberOfRepetitions++
30 else matched = false
31 if matched
32 Pivots.Push(loc + 1);
33 Partition(F, L, Pivots, t, loc + 1)
34 Pivots.Pop()
35 Partition(F, L, Pivots, t, loc + 1)
36
37 function Match(AstForest F1, AstForest F2)
38 return Align(F1, F2)  $\geq$  threshold
39
40 function Align(AstNode n1, AstNode n2)
41 if n1.type  $\neq$  n2.type return 0;
42 if n1 is an echo statement
43 if n1.expressions = n2.expressions return 1 else return 0.5
44 return 0.5 + Align(n1.childrenForest, n2.childrenForest)
45
46 function Score(Template t)
47 return t.numberOfStaticClientToken * t.numberOfRepetitions
48
49 function Update(Template t, AstForest F2)
50 AstForest F1 = t.forest
51 for each matched pair of AST nodes n1 in F1 and n2 in F2
52 n1.replacements.Add(n2)
53 if n1.isLeafNode
54 if n1.value  $\neq$  n2.value, Mark n1 as a placeholder AST node
55 else return Update(n1.childrenForest, n2.childrenForest)
56 for each unmatched AST node n1 in F1
57 if n1 is not an expression return false;
58 Mark n1 as a placeholder node
59 n1.replacements.Add(null)
60 for each unmatched AST node n2 in F2
61 if n2 is not an expression return false;
62 Add a placeholder subtree rooting at n1 for n2 in F1
63 for i = 0 to t.numberOfRepetitions n1.replacements.Add(null)
64 n1.replacements.Add(n2)
65 return true

```

Fig. 6. Output-oriented Dynamicalization Refactoring (2)

#### 4) Step 4. Generating Dynamicalization Source Slice:

With the best partition, WebDyn generates the corresponding PHP code. The generated code has an array to store different strings at the place holders. Then, it generates a loop iterating through that array and produces the code based on the shared template among the partitions. If the place holders have the values made from the iterating variables, it will use them.

Currently, WebDyn parameterizes only expressions.

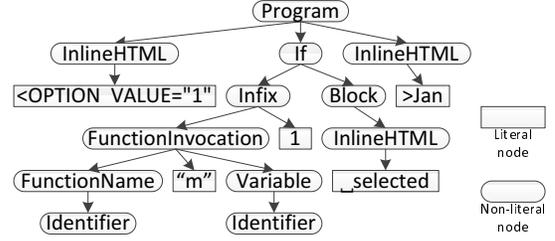


Fig. 7. Abstract Syntax Tree for the HTML <OPTION> element

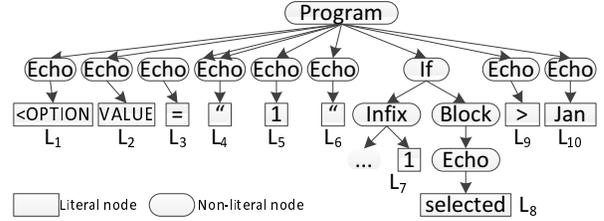


Fig. 8. Transformed Abstract Syntax Tree for HTML <OPTION> element

## VII. EMPIRICAL EVALUATION

This section presents our empirical evaluation on WebDyn's accuracy. We checked out the initial revisions of 5 open-source PHP systems (Table III). Four of them had been used in our exploratory study in different ranges of revisions. We chose the early revisions so that there would be more candidate code for refactoring. We wrote an evaluation tool to automatically select blocks of code for dynamicalizing refactoring in each project. For each starting line, the tool started with the maximum selection of 20 lines and requested WebDyn for refactoring. If the selection was not a valid source slice, it tried the selection with one line smaller until the code block was valid for refactoring or no selected line left. We recorded all those invalid selections. When the selection was valid, WebDyn would try to refactor to either produce the refactored code or notify that the source slice has no repeated embedded client code. In the former case, we manually checked if the transformed code was syntactically correct and generated the equivalent client code. In the latter, we also manually verified if the selected code had no repetition to refactor. In this way, we could exercise the tool with different scenarios: invalid selections, selections with no embedded client code or with no repeated client code, and selections which could be dynamicalized.

### A. Accuracy in Dynamicalization Refactoring

Table III shows the results on WebDyn's accuracy. Under column Selection, column  $\overline{Emb}$  shows the number of selections containing no embedded client code, column  $\overline{Repeat}$  shows that of the selections with no repetition, and column Dyn shows the number of dynamicalizable selections. Since a dynamicalizable block can appear in multiple selections, the selection sets corresponding to columns  $\overline{Emb}$ ,  $\overline{Repeat.Total}$ , and Dyn.Total are not exclusive. We manually checked all results in all cases. However, since the number of *non-dynamicalizable*

TABLE III  
WEBDYN'S DYNAMICALIZATION REFACTORING ACCURACY

System	F. KLOC		Selection					Dyn		
			Total		<i>Emb</i>		<i>Repeat</i>			
			(K)	(K)	Total	Prec.	Total			Prec.
AddressBook	17	1.6	10	9	265	429	97%	5	100%	
DotProject	61	7.9	35	31	1,345	2,237	100%	28	100%	
LimeSurvey	17	4.7	43	40	1,162	1,151	100%	6	100%	
PhpFusion	146	17.0	166	152	4,943	3,922	96%	70	100%	
PostfixAdmin	50	6.9	45	42	3,240	201	100%	2	100%	
Average							99%		100%	

selections is large (1,000-10,000), we randomly chose 100 samples for each of the three categories for manually checking.

As seen, for the dynamicalizable selections, WebDyn has 100% precision. All refactored code has the equivalent client-side code. For the selections that are invalid or have no embedded code, it also achieves 100%. For the selections with no repetition, it achieves 100% for 3 systems and 99% on average for all 5 systems. The high accuracy in non-dynamicalizable cases reflects high recall since WebDyn misses very few dynamicalizable ones. The missing cases are due to the approximation in PhpSync and our repetition detection algorithm.

### B. Characteristics of Refactored Code

To better understand the characteristics of the dynamicalized code, we further analyzed their locations in the code, sizes, the level of repetition and the amount of code that was refactored. The result is shown in Table IV. For all systems, the dynamicalizable code appears in many files. In four of them, almost 30% of the files has such code. However, the size of such code is usually small, mostly around 2-4 lines. PhpFusion has the largest one (17 LOCs) and also has the highest number of large code. The number in column Repetitions is the number of (repeated) partitions that had been parameterized. Most dynamicalization cases have two partitions. AddressBook has one case of 26 partitions, which will be shown in Section VII-C.

To measure the amount of code that was refactored, we computed the Dynamicalization ratio, which is the ratio between the amount of embedded code after and before refactoring. The lower this number is, the more dynamic the selected code can be. In most cases, this ratio is 40%-50% which is highly co-related with the dominance of two repetitions. PhpFusion has the maximum of 5 partitions, and this ratio can be as low as 5%. This is possible thanks to our mapping of the different values among partitions to the iteration variable of the loop.

### C. Case Studies

**Case Study 1.** Figure 9 shows a case in AddressBook. The hyperlinks (enclosed by the anchor tags) share most of their HTML code except for their href attributes ('a', 'b' ... 'z') and inner texts ('A', 'B' ... 'Z'). WebDyn was able to recognize this repetition and dynamicalize it to remove the duplicated code. The code is not only shorter but also more readable. More importantly, it could be easier to maintain. For changing the

#### a) Original code (AddressBook/index.php):

```
echo "<a style='font-size:75%' href='$link=a'>A</a> | ...
| <a style='font-size:75%' href='$link=z'>Z</a> |";
```

#### b) Dynamicalized code:

```
$_array1 = array(array("a", "A"), array("b", "B"), ..., array("z", "Z"));
for ($_index1 = 0; $_index1 < count($_array1); $_index1++) {
    $_value = $_array1[$_index1];
    echo "<a style='font-size:75%' href='$link=$_value[0]'>$_value[1]</a> |
    ";
}
```

Fig. 9. Dynamicalization in AddressBook

#### a) Original code (LimeSurvey/admin/dataentry.php):

```
1 echo "$setfont<INPUT TYPE='...' NAME='...' VALUE='Y'";
2 if ($idrow[$i] == "Y") {echo " CHECKED";} echo ">Yes&nbsp;";
3 echo "$setfont<INPUT TYPE='...' NAME='...' VALUE='N'";
4 if ($idrow[$i] == "N") {echo " CHECKED";} echo ">No&nbsp;";
```

#### b) Dynamicalized code:

```
1 $_array1 = array(array("Y", "Yes", $setfont), array("N", "No", ""));
2 for ($_index1 = 0; $_index1 < count($_array1); $_index1++) {
3     $_value = $_array1[$_index1];
4     echo "<INPUT TYPE='...' NAME='...' VALUE='";
5     echo "" VALUE='"; echo $_value[0]; echo """;
6     if ($idrow[$i] == $_value[0]) {echo " CHECKED";}
7     echo ">$_value[1]&nbsp;"; echo $_value[2];
8 }
```

Fig. 10. Dynamicalization in LimeSurvey

font-size, one would just edit a token in the dynamicalized code while (s)he would have to edit 26 places in the original code.

**Case Study 2.** Figure 10 displays another case in LimeSurvey. The original PHP code is highly intermixed with its embedded HTML code. However, WebDyn was still able to identify two code fragments sharing the same structure (lines 1-2, and lines 3-4, Figure 10a). Though the result is correct, it might be different from the result of manual refactoring where the repeated code could start from the variable at the beginning of the echo statement. This difference is caused by our current algorithm that always starts a partition with an opening tag.

**Threats to Validity.** The result was checked manually. It has inherent threats from PhpSync for D-model building and parsing. Five selected subject systems might not be representative.

## VIII. RELATED WORK

Refactoring is an important part of software maintenance. Mens *et al.* [13] provide a comprehensive survey on refactoring. However, there are not many approaches addressing refactorings for dynamic, multi-lingual Web code.

There is research on refactoring for Web applications at coarser-grained levels. Cabot and Gomez [2] introduce a catalog of refactoring changes for the navigational structure of a Web application. Rossi *et al.* [20] and Garrido *et al.* [8] propose methods to refactor on a Web architectural model. Ping and Kontogiannis [18] refactor Web sites via clustering pages and inter-links toward control-centric architecture. Ricca and Tonella [19] define code structuring for Web languages,

TABLE IV  
WEBDYN'S DYNAMICALIZATION REFACTORING RESULT

System	Dyn	F.	%F.	LOC	%LOC	Lines					Repetitions					Dynamicalization ratio							
						1	2	4	8	16	More	2	3	4	5	More	5%	10%	20%	30%	40%	50%	More
AddressBook	5	5	29%	30	1.8%	1	1	1	0	2	0	3	1	0	0	1	1	0	0	0	1	1	2
DotProject	28	28	46%	113	1.4%	0	15	6	5	2	0	21	6	1	0	0	0	0	1	4	4	16	3
LimeSurvey	6	6	35%	16	0.3%	0	5	0	1	0	0	6	0	0	0	0	0	0	0	2	4	0	0
PhpFusion	70	70	48%	345	2.0%	2	28	13	16	10	1	53	10	3	4	0	1	0	7	12	21	26	3
PostfixAdmin	2	2	4%	4	0.1%	0	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	2	0

but they focus on coarse-grained changes. Di Lucca *et al.* [4], [5] develop Page Control Flow Graph (PCFG). In PCFG, in which a node represents a sequence of unconditionally executed statements, and an edge represents the control flow among statements. They also develop WARE [5], a dynamic analysis framework for reverse engineering of a UML-like diagram among coarse-grained Web entities. Graunke *et al.* [10] introduce a Web CFG that takes into account both programs and their display elements with user interactions. DSketch [3] supports finding dependencies among Web entities.

In comparison, WebDyn has key differences with those approaches. First, the above approaches do not analyze *dynamic, embedded* code and its relation to the host program. Second, their goal is more toward program comprehension and architectural recovery. They focus more on linking structure among Web pages, rather than fine-grained Web code refactoring.

Minamide [14]'s string analyzer approximates the output of a PHP program via a context-free grammar. It requires a regular expression describing the input. Based on that, Wang *et al.* [21] computed the constant strings visible from the browser for translation. WebDyn uses D-model [16] to approximate the output of PHP code via symbolic execution, without the input. Our prior tool [17] supports renaming for embedded entities. It also uses D-model and has a HTML parser to produce program tokens in embedded languages HTML and JS. We use both D-model and that parser to support WebDyn's refactorings.

Research on refactoring recovery for single-language programs has been extensive [6], [22], [1], [9]. Our work is related to research in original analysis during software evolution [12], [9], [11], [7]. It is also related to clone detection, especially for Web code. Muhammad *et al.* [15] use Island grammar for such detection. We use PhpSync to approximate the output.

## IX. CONCLUSIONS

This paper presents an empirical study on several PHP-based Web applications. We found that after an *output-oriented refactoring*, server code is more dynamic, but the corresponding output client-side code provides the same behavior. We also develop WebDyn, a method/tool to support dynamicalization refactorings. Our empirical evaluation on several dynamic Web applications shows that WebDyn can achieve 100% accuracy in such refactorings. In future work, we plan to conduct a controlled experiment on human subjects to study the usability of WebDyn and improve it to match better with developers' intention. We will measure other code quality metrics such as code readability/maintainability.

## ACKNOWLEDGMENT

This project is funded in part by US National Science Foundation (NSF) TUES-0737029, CCF-1018600, and CNS-1223828 awards.

## REFERENCES

- [1] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In IWPSE '04. IEEE CS, 2004.
- [2] J. Cabot and C. Gómez. A catalogue of refactorings for navigation models. In ICWE, pages 75–85, 2008.
- [3] B. Cossette and R. J. Walker. DSketch: lightweight, adaptable dependency analysis. In FSE '10, pages 297–306. ACM, 2010.
- [4] G. A. Di Lucca and M. Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In WSE'05.
- [5] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering web applications: the WARE approach. *J. Softw. Maint. Evol.*, 16:71–101, Jan 2004.
- [6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In ECOOP'06.
- [7] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.
- [8] A. Garrido, G. Rossi, and D. Distanto. Model refactoring in web applications. In WSE'07, pages 89–96. IEEE CS, 2007.
- [9] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, 2005.
- [10] P. Graunke, R. B. Fidler, S. Krishnamurthi, and M. Felleisen. Modeling Web interactions. In ESOP'03, pages 238–252. Springer-Verlag.
- [11] M. Kim and D. Notkin. Discovering and representing systematic code changes. In ICSE'09, pages 309–319. IEEE CS, 2009.
- [12] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In WCRES'05.
- [13] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [14] Y. Minamide. Static approximation of dynamically generated web pages. In WWW '05, pages 432–441. ACM, 2005.
- [15] T. Muhammad, M.F. Zibrán, Y. Yamamoto, C.K. Roy. Near-Miss Clone Patterns in Web applications: An empirical study with industrial systems. In CCECE'13.
- [16] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In ASE'11. ACM, 2011.
- [17] H. Nguyen, H. Nguyen, T. Nguyen and T. Nguyen BabelRef: Detection and Renaming Tool for Cross-Language Program Entities in Dynamic Web Applications. ICSE'12, IEEE CS, 2012.
- [18] Y. Ping and K. Kontogiannis. Refactoring web sites to the controller-centric architecture. In CSMR'04. IEEE CS, 2004.
- [19] F. Ricca and P. Tonella. Program transformation for web application restructuring. *Web Eng.: Principles and Techniques*, XI:242–260, 2005.
- [20] G. Rossi, M. Urbietta, J. Ginzburg, D. Distanto, A. Garrido. Refactoring to rich internet applications a model-driven approach. In ICWE'08.
- [21] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In FSE'10. ACM, 2010.
- [22] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In ASE'06. pages 231–240. IEEE CS, 2006.
- [23] dotproject. <http://www.dotproject.net/>.
- [24] PHP Address Book. <http://sourceforge.net/projects/php-addressbook/>.
- [25] "Why Validate." <http://validator.w3.org/docs/why.html>, W3C.

# On the Automation of Dependency-Breaking Refactorings in Java

Syed Muhammad Ali Shah, Jens Dietrich, Catherine McCartin  
 School of Engineering and Advanced Technology  
 Massey University, Palmerston North, New Zealand  
 Email: {m.a.shah, j.b.dietrich, c.m.mccartin}@massey.ac.nz

**Abstract**—The architecture and design of object-oriented systems is often described in terms of dependencies between program artefacts like classes, packages and libraries. In particular, there are many heuristics that mandate that certain dependencies should be avoided, and there are increasingly popular frameworks that focus on actively managed dependencies between artefacts, such as the Spring framework and the OSGi dynamic module system. However, empirical studies have shown that many real-world Java programs are riddled with dependency related problems. The question arises whether it is possible to automatically refurbish Java programs by removing unwanted dependencies without affecting the functionality of the program. We present an algorithm and a proof-of-concept implementation that does this. Our approach uses several refactorings: move class, type generalisation, service locator, and inlining and is validated on the Qualitas Corpus set of Java programs.

## I. INTRODUCTION

The dependency graph is a simple yet effective model to describe the design and architecture of object-oriented programs. In this model, artefacts like classes and packages are represented by nodes, and their relationships are represented by edges. Using a dependency graph, design issues can be studied through metrics or patterns. Martin's classical study on dependencies [1], and advice on avoiding cyclic dependencies, going back to the classical work of Parnas [2] and Stevens, Myers and Constantine [3] are examples for this.

Unfortunately, mainstream programming languages have little or no built-in support to manage dependencies. For instance, the Java compiler merely prevents cyclic dependencies in the inheritance graph. On the other hand, language features, such as inner classes in Java, cause "hidden" dependencies through implicit references.

Many anomalies in dependency graphs that have been related to poor software quality attributes, in particular maintenance, are surprisingly common in real world programs. Broader empirical studies conducted recently have revealed that most real-world programs are riddled with dependency-related problems [4], [5] often expressed in terms of antipatterns such as cyclic dependencies (of several types) [2], [5], subclass knowledge [6] or diamond inheritance [7]. On the other hand, there is now a new generation of application development frameworks that try to manage dependencies more actively. Usually this means to avoid dependencies

being hard coded in the source code of programs, and to move to runtime composition, where dependencies are established (artefacts are "wired") at runtime, and runtime environments then reason about these dependencies. One particular advantage of this approach is that it makes it easier to evolve systems. Examples of technologies in this category are dependency injection frameworks like Spring [8] and Guice [9], and dynamic component models like OSGi [10] and its extensions, such as the Eclipse extension registry and declarative services. Managing dependencies is directly related to modularity. Modularity is based on the idea of organising software by composing it from somewhat independent parts. This implies certain dependency constraints. If there are too many inter-module dependencies, this independence, and therefore modularity itself, is lost.

This raises a question: is it possible to refactor applications into a more modular structure by removing dependencies? The very notion of refactoring implies that this should not change the behaviour of the program. In practice, there are several techniques that can be used to achieve this. For instance, a dependency between packages might be caused by a class that is misplaced, and moving this class will remove this dependency. Type generalisation can often be used to remove dependencies that originate from inappropriate type declarations. Inlining, service locators, and dependency injection are other techniques that can be used. These techniques are widely used by software engineers to *manually* correct dependency-related problems.

The question we are interested in is whether these techniques are also suitable for automation. A major hurdle is the fact that there are *many* dependencies: the class libraries of the Java Runtime Environment (JRE) version 1.7.0, for instance, contains 217973 inter-class, 10072 inter-package and 22 inter-jar dependencies<sup>1</sup>. So, the problem becomes, *which* dependencies should be removed. This is a problem we have addressed in our previous work [11]. The idea is to compute the instances of structural antipatterns [5] in the dependency graph using graph queries, and then to use a scoring algorithm that ranks dependencies based on their participation in these antipatterns. This is inspired by ideas used in other popular graph scoring algorithms like page rank [12] and betweenness centrality [13]. The advantage of this

<sup>1</sup>Massey Architecture Explorer <http://goo.gl/V5Gf5>

method is that it is possible to dramatically reduce the numbers of structural antipattern instances by removing a relatively small number of edges with high scores. We therefore regard these edges as *high impact refactoring opportunities*. If we can execute the actual refactorings to remove these dependencies, these refactorings will become actual *high-impact refactorings*.

We do not expect that removing or reorganising dependencies is always possible. For instance, to replace a reference to a type A by a reference to its supertype B we have to check whether the part of the interface of A that is used by a client is also part of B's interface, i.e., certain preconditions must be satisfied before refactorings can be performed. Even if preconditions succeed, the refactored code might not be correct or might not improve the quality criteria. This implies that refactorings also have to be safeguarded by postconditions such as successful compilation. The question we try to answer here is: *to what extent* can refactorings be automated. This paper extends work we have published previously on the removal of dependencies using two particular refactoring techniques: move refactorings [14] and the introduction of service locators [15]. The contributions of this paper are as follows: (1) we present a refactoring engine implemented using the Eclipse refactoring engine that uses several refactoring techniques: move class, service locator, type generalisation and inlining. (2) we conduct a comprehensive empirical study using the Qualitas Corpus data set to find out how many of the critical edges computed using the algorithm from [11] can be removed.

The rest of this paper is organised as follows. We discuss related work in section 2. Next we discuss a set of antipatterns and categories of dependencies in sections 3 and 4 respectively. Then we present the various refactoring techniques we use and the refactoring engine we have implemented in sections 5 and 6 respectively. Section 7 presents a brief overview of the refactoring engine algorithm. Section 8 contains the validation of our approach on the Qualitas Corpus [16], followed by a conclusion and a discussion of future work in section 9.

## II. RELATED WORK

### A. Search Based Automated Refactorings

In the literature, search based techniques have been used to automatically refactor existing systems [17], [18], [19]. These techniques are mainly applied on code and class level whereas our approach focuses on improving the overall structure of existing systems without changing the external behaviour.

### B. Type Generalisation

Several tools and techniques exist that aid developers to use more abstract types. For example, Mayer et al. have developed a refactoring tool that detects code smells and executes refactorings on the source code level [20]. In this tool the supertype hierarchy is displayed to the user in the form of a lattice. The user can choose

a relevant refactoring among multiple refactoring suggestions. This process involves human interaction and cannot be fully automated.

Streckenbach and Snelting have developed the KABA refactoring tool, which uses split classes and move members refactorings [21]. A drawback of this tool is that it cannot modify a program's source code. However, it can be used to modify the bytecode of the program. Bach et al. have developed an Eclipse plugin, which finds better fitting types in programs [22]. This plugin looks for all variable declarations, field declarations, method parameter types and method return types to compute valid supertypes. Once valid supertypes are computed, this plugin generates warnings in the Problem View of Eclipse. Quick fixes are associated with each variable declaration to automatically re-declare a variable with an abstract type. The selection of an abstract type for re-declaring a variable must be done manually. Steimann et al. have presented a study showing that, in several large Java projects, one out of four variables was declared through its interface [23]. The authors have defined a metric suite related to interface utilisation in object-oriented programs and have proposed refactorings for a better utilisation of interfaces in programs. The metrics definition is very vague and no implementation exists to validate the effectiveness of the approach.

### C. Service Locators

Martin has defined the Dependency Inversion Principle (DIP), which states that "high level modules should not depend upon low level modules. Both should depend upon abstractions" [1]. This means a class should depend on an abstract type rather than on a concrete class. However, these abstract types still have to be instantiated using concrete classes. There are several ways to instantiate a concrete class and pass it to the client class exhibiting DIP. One possible solution to locate and utilise implementation classes is a J2EE pattern called service locator<sup>2</sup>. Fowler has discussed the benefits of using service locators to avoid instantiation problems [24]. He suggests to create a service locator, which has knowledge about a service and its implementations and to use the service locator along with a registry to locate and instantiate implementation types. Shah et al. have used the service locator pattern to achieve modularity in object-oriented programs [15].

### D. Static Members Inlining

The inlining technique is primarily used by the Java compiler for the optimization of program execution, it is also known as constant folding [25]. This type of inlining is limited to static and final values and string constants, where calls to constants are replaced with their values by the compiler. Inlining is also used as a refactoring technique where fields, methods and classes are inlined into a client class for the purpose of optimisation or for removing a dependency. For example, in order to

---

<sup>2</sup><http://oracle.com/technetwork/java/servicelocator-137181.html>

remove a dependency, Feathers has suggested a refactoring called *extract and override call* in which the method signature of the invoked method (causing a dependency relation) is copied into the client class and a subclass of the client implements and returns a mock value for testing [26]. Similarly, the move method refactoring can be used to break the dependency between two classes [27].

Object inlining is a similar technique, where a dependency to a class is removed by inlining its members [28], [29]. By using this technique calls to new object creations can be minimised. This minimisation of new creations may improve the performance of a system [30]. Object inlining may significantly change the meaning of classes. It may also result in increasing the complexity of source code. Due to these reasons we restrict to inlining static members, where we move the required static members of the target class to the source class. As an alternative to break new object creation dependencies, we use service locators.

### E. Move Class

The move class refactoring has been used to re-modularise programs, improve modularisation of existing programs, and remove antipatterns. For example, Seng et al. have developed an algorithm to re-modularise programs by improving subsystem decomposition [31]. The authors have used a fitness function based on coupling, cohesion, complexity, cycles and bottlenecks. They treated the subsystem decomposition as a search problem and used a genetic algorithm to identify move class refactoring opportunities. Their results show improvement in some metrics but not the coupling and cohesion of individual packages. Their work is related to software re-modularisation as their solution generates new packages and suggests classes to move to them.

Hautus has suggested the PAKage STructure Analysis (PASTA) metric to measure the degree of cycles between packages in a program dependency graph [32]. This metric can be used to identify undesirable dependencies between packages in a program. When these dependencies are removed, the package structure becomes acyclic. The author has suggested the move class refactoring as a solution to get rid of package cycles. However, his approach does not identify classes that should be moved to other packages in order to remove cycles. Abdeen et al. have used the move class refactoring to remove cyclic dependencies on the graph level [33]. Shah et al. have also used the move class refactoring to remove tangles from Java programs on the source code level [14].

## III. DEPENDENCY SCORING

We have used a set of antipatterns, representing design flaws, to select critical dependencies in a program. The antipattern count is also used as a metric to assess results of refactorings. We score dependencies by the number of antipattern instances this edge depends on. A detailed discussion on these antipatterns is given in

our previous work [5]. The following four antipatterns are used in our approach:

### A. Circular Dependencies between Packages (SCD)

The dependency cycle between packages is a variation of cycle between modules discussed by [3]. Packages that are involved in circular dependencies become less maintainable and are difficult to reuse [34]. Empirical studies have shown that circular dependencies between classes and packages are very common in open source Java programs [35], [5].

### B. Subtype Knowledge (STK)

In the subtype knowledge (STK) antipattern [6] a supertype either directly or indirectly uses its subtype. This implies that we cannot use super and subtypes in isolation.

### C. Abstraction Without Decoupling (AWD)

In the abstraction without decoupling (AWD) antipattern, a client depends on a service (supertype) and an implementation of the service (subtype) at the same time.

### D. Degenerated Inheritance (DEGINH)

The degenerated inheritance (DEGINH) antipattern [7] appears in a program when there are multiple inheritance paths from a subtype to its supertype. In Java programs, this can be introduced by the use of multiple inheritance through interfaces. This antipattern creates duplication in the program structure and makes it difficult to separate subtypes from their supertypes.

## IV. DEPENDENCY CATEGORIES

Dependencies can be classified into different types. Our classification of dependencies is derived from earlier works on dependencies [36], [4]. Listing 1 shows different types of dependencies in a Java program. In this listing class A depends on other types B, C, D, E, F, G, H and System. In this context, we call the dependent class A the **source type**, and the rest of dependency classes **target types**.

---

```

1 public class A extends B implements C {
2     private D object = new E();
3     public F m(G obj) throws H {
4         System.out.println(obj.toString());
5         return obj.getF(); } }

```

---

**Listing 1:** Java source code creating dependencies

We can broadly classify a dependency relationship between a source and a target type into the following nine categories:

- 1) Variable Declaration (VD): The target type is used to declare a variable or a field, for example,  $A \rightarrow_{uses} D$ .

Dependency Category	Refactoring
Variable Declaration (VD) Method Parameter Type (MPT) Method Return Type (MRT) Method Exception Type (MET)	Type generalisation
Constructor Invocation (CI) Static Member Invocation (SMI)	Service Locators Static Members Inlining
Extends (EX) Implements (IM) Other	Move Class

**TABLE I:** Dependency categories and their default respective refactorings

- 2) Method Return Type (MRT): The target type is used as a return type of a method in the source type, for example,  $A \rightarrow_{uses} F$ .
- 3) Method Parameter Type (MPT): In this case the target type is used as a method parameter in the source type, for example,  $A \rightarrow_{uses} G$ .
- 4) Method Exception Type (MET): The target type is used as an exception type using *throws* keyword, for example,  $A \rightarrow_{uses} H$ .
- 5) Constructor Invocation (CI): A target type constructor is invoked through *new* keyword, for example,  $A \rightarrow_{uses} E$ .
- 6) Static Member Invocation (SMI): When a static member of a class (field or method) is invoked on the target type, for example,  $A \rightarrow_{uses} System$ .
- 7) Superclass (SC): The target type is used as a supertype through the *extends* keyword, for example,  $A \rightarrow_{extends} B$ .
- 8) Interface (IN): The target type is used as an interface through *implements* keyword, for example,  $A \rightarrow_{implements} C$ .
- 9) Other: In this category, references to a class such as *B.class*, *object instanceof E* and cast expressions are included.

Table I shows the default refactoring for each dependency category. This classification can be altered while performing a refactoring. For example, if a dependency edge is caused by multiple dependency categories (such as, SMI, VD, MRT etc.) then initially class level refactorings are attempted. If pre or postconditions of class level refactorings fail then package level refactoring (move class) is attempted.

## V. REFACTORING TECHNIQUES

### A. Type Generalisation

The declaration of variables with abstract types is considered good programming practice [37]. However, we find that, in practice, abstract types are rarely used to declare variables and fields in a class [38]. The type generalisation refactoring can be used along with the service locator pattern to decouple classes. In our approach, we apply type generalisation to the following three dependency categories: VD, MPT and MRT. Using

the type generalisation technique, we reorganise dependencies rather than removing them.

### B. Service Locator

The service locator pattern can also be used to address the decoupling problem. In this pattern, service implementation classes are decoupled from their client classes. Fowler describes the service locator pattern as a registry that is used to look up instances of implementation classes [24]. This pattern is particularly useful when there exist alternative service implementations and they need to be replaced or upgraded. We use this technique to break CI dependencies.

According to Fowler, both service locators and dependency injection can be used to break dependencies from the client class, however, there is a subtle difference between the two techniques. In the former, the client class pulls a required service from a registry, while in the latter the required service is pushed (injected) into the client class.

### C. Static Member Inlining

In the source code of programs, some dependencies occur due to the invocation of static members<sup>3</sup> of the target class. In some cases, static methods are used to declare utility functions that are self-contained, i.e. they don't use a class's data or methods. Therefore, to break a dependency of type SMI, a possible solution is to inline the target static member into the source class.

In order to remove static member dependency, the move refactoring is used to move the required static member to the source class and a delegate or proxy to the moved member is created in the target class. In this way, the dependency between the source and the target class is inverted. This is not an invasive technique, as it does not require updating clients of the static member.

This refactoring can be recursive, i.e., when a method is moved, all its dependent members inside the target class must be moved. Similarly, moving a static field could also be recursive due to initialisation of the field. For example, the following code requires to move the field and its dependencies. `public static Factory instance = createInstance();` It is also possible to move the static initialisation block, which initialises the required field. This refactoring technique breaks SMI dependencies.

### Example

Listing 2 shows a simple example of static method inlining. In this example the static method `printMessage` is moved into the source class `S` and the invocation statement is modified to refer to the new location. In the target class `T`, a proxy method is created, which now calls the moved static method in the class `S`.

<sup>3</sup>We refer to public methods and fields as members of the class.

---

```

1 //Before refactoring
2 public class S { ...
3     public void m() {
4         T.printMessage(message ); } ... }
5 public class T { ...
6     public static void printMessage(String m) {
7         System.out.println(m); } }
8 //After refactoring
9 public class S { ...
10    public void m() {
11        printMessage(message ); }
12    public static void printMessage(String m) {
13        System.out.println(m); } ... }
14 public class T { ...
15    public static void printMessage(String m) {
16        S.printMessage(m); } }

```

---

**Listing 2:** Static member inlining example

#### D. Move Class

The move class refactoring is commonly used to break dependency cycles between packages [39]. This refactoring is rather simple and robust tool support exists to automate it. Modern IDEs such as Eclipse, Netbeans, and IntelliJ IDEA have built-in support for the move refactoring. These tools automatically fix references in other classes affected by the move class refactoring.

Another advantage of the move class refactoring is that it can be completely simulated on the model level. This provides the ability to evaluate the impact of refactoring without changing the source code of programs. For example, on the dependency graph level, where classes are represented as vertices and edges represent relationship between classes, a move class refactoring can be performed by changing package labels on respective vertices.

## VI. REFACTORING ALGORITHM

An overview of the refactoring algorithm is given below:

- 1) Build the dependency graph from the bytecode of a program.
- 2) Use the Guery engine (ver 1.3.5) to compute the set of SCD, STK, AWD and DEGINH instances [11].
- 3) Compute a list of edges (class dependencies) sorted by score ranking based on their participation in all types of antipattern instances.
- 4) Parse the program's source code into ASTs.
- 5) Check preconditions to determine whether a dependency can be removed or not.
- 6) If the preconditions are satisfied, apply the refactoring on the program's ASTs. Otherwise try the next high-scored edge.
- 7) Evaluate postconditions to check whether the applied refactoring has introduced any errors.
- 8) If postconditions are satisfied, update the program's source code. Otherwise, rollback the ASTs to their previous states.

- 9) Repeat the process until all antipatterns instances are removed or a certain number of iterations are performed (*MAX* is 50 in our case).

#### A. The Dependency Graph

The first step of the algorithm starts with building the dependency graph of a program. In the dependency graph, each reference type in a program (class, interface, enum etc) is represented as a vertex. Additional properties such as namespace (package name), visibility and abstractness are represented as labels on vertices. The dependency graph does not contain method or attribute level information of a class. Edges represent relationships between classes and are labelled with relationship types. Three supported relationship types are *uses*, *extends*, and *implements*. The relationship types *extends* and *implements* represent inheritance while the relationship type *uses* represents all other types of references between classes such as type references in methods and fields. The process of building the dependency graph is explained in [5].

#### B. Scoring

In the second step, the algorithm computes instances of all four types of antipatterns. We have used a scoring function [11], which assigns a high score to edges that participate in a large number of antipattern instances. Scoring associates a number with each edge indicating in how many antipattern instances this edge occurs. This means that the removal of a high-scored edge should likely remove multiple antipattern instances. We expect that this will result in high-impact refactorings.

#### C. Parsing Source Code

After the identification of critical edges (class dependencies), we need to analyse the source code to verify whether removing or reorganising these dependencies is possible or not. For this purpose, we extract the Abstract Syntax Tree (AST) of the required classes. We have used the JDeodorant API<sup>4</sup> based on Eclipse Java Development Tool (JDT) to parse a program's source code. This API provides utility methods that assist in performing static code analysis such as obtaining incoming and outgoing references of a method.

#### D. Preconditions

1) *Generalize Refactoring Preconditions:* The type of a variable, field, method parameter or return type cannot be generalized if the class members invoked on that variable or field are not part of any supertype's interface. Consider the following example, where a method parameter type is generalised. In this case, Class A can invoke the method `size()` on both types: `java.util.Vector` and `java.util.Collection`.

---

<sup>4</sup><http://www.jdeodorant.org>

---

```

1 //Before refactoring
2 class A {
3     public int foo(java.util.Vector nums) {
4         int size = nums.size(); ... } }
5 //After refactoring
6 class A {
7     public int foo(java.util.Collection nums) {
8         int size = nums.size(); ... } }

```

---

The above refactoring would not work if the class A invoked `nums.get(i)` in the method body of `foo()`. In this case, replacing `java.util.Vector` with its supertype `java.util.Collection` is not possible because the method `get(int i)` is not a part of `java.util.Collection` interface.

In a similar way, we cannot generalise a type when a reference to the target type is leaked. For example, in the above code we have a method invocation `new OtherClass().check(nums)` in the method body, then we do not know which part of the interface of `Vector` is used in `OtherClass`. While it is possible to follow this references in principle, the existing *Generalize Declared Type* refactoring does not support this.

Second precondition for this refactoring is bound to the type of program being refactored. If we are trying to refactor a program which is used (e.g. as a library) in different contexts, this precondition should be enabled. Stand alone applications that are not used programmatically by other programs do not require this precondition. This precondition states that if a target type is used as the declaration type of a public field or used as a return type of a public method, we can't generalise that target type. Doing so may break the external client code that is not visible during refactoring. In the context of the work described in this paper, we apply our refactorings on a large dataset where it is hard to differentiate between the two types of programs. Therefore for the current experiments, we keep this precondition disabled.

2) *Inline Refactoring Preconditions*: This precondition checks the method body of the target method and confirms that there does not exist a new instance creation of the target type. The following example explains the situation. In this example, even if we move the `staticMethod` to class A, the dependency on class B will still exist.

---

```

1 class A {
2     void m() { B.staticMethod(); } }
3 class B {
4     public static void staticMethod() {
5         B b = new B(); } }

```

---

3) *Move Class Preconditions*: A class is not moved if its visibility is restricted to the package level and it is referenced by other classes within the original package. The move refactoring is not performed where the target package contains a class with the same name. If the class to be moved is an inner class, its compilation unit (all types within the class) is moved. Note that there is no restriction for a class (which was moved to another

package) to be moved back to the original package as long as it satisfies pre and postconditions.

#### E. Postconditions

The first postcondition states that the program should compile successfully after the refactoring. For example, moving a non-public class to another package results in compilation error, if the class is referenced in the original package. The second postcondition is related to the instance count before and after refactoring. The instance count after refactoring should be less than or equal to the instance count before refactoring. A refactoring is only considered successful when it passes these two postconditions. If either of the postcondition fails, the whole refactoring is rolled back.

## VII. REFACTORING IMPLEMENTATION - CARE PLUGIN

The algorithm is implemented as an Eclipse plugin called CARE (Code and Architectural Refactoring Environment)<sup>5</sup>. We have used four refactorings discussed in Section V to deal with dependencies. In order to implement refactorings, we have used the Eclipse Language Toolkit (LTK). This toolkit is a refactoring framework that provides access to the Eclipse IDE's code level refactorings, such as move refactoring, pull up / push down methods. The base class for all refactorings is `org.eclipse.ltk.core.refactoring.Refactoring`. This is an abstract class and all refactorings inherit methods from this class. The `Refactoring` class performs two actions: checks several conditions to make sure that the refactoring can be safely executed and creates a `Change` object which represents the source code modifications.

In our approach, we have extended the base `Refactoring` class and created another abstract class `CarRefactoring` as shown in figure 1. This class performs three major actions:

- 1) It checks a set of preconditions before applying a refactoring. It calls `checkAllConditions()` method from the superclass `Refactoring`.
- 2) If the preconditions succeed, refactorings are performed tentatively on the source code. This step involves calling `createChange()` method of the `Refactoring` class and then calling the `perform()` method on the change object.
- 3) Postconditions are checked to determine whether or not refactorings were performed successfully. If postconditions pass, the refactoring process is considered successful, otherwise the code level modifications are rolled back.

#### A. Generalize Refactoring

The *Generalize* refactoring is built on top of the standard Eclipse refactoring *Generalize Declared Type*, implemented via the `ChangeTypeRefactoring` class. In

---

<sup>5</sup><http://code.google.com/p/care>

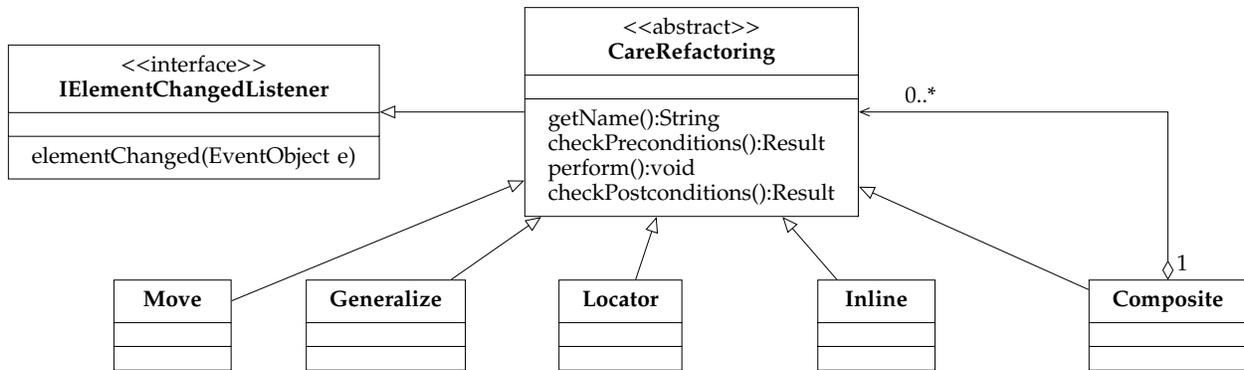


Fig. 1: Class diagram of refactorings

the *Generalize Declared Type* refactoring, we have to manually select a declaration type and see whether this can be replaced with one of its supertypes. The *Generalize* refactoring *automatically* replaces all target type declaration elements within a class with a specific compatible supertype.

The *ChangeTypeRefactoring* class computes a list of all possible supertypes with which a declaration type can be replaced. This class supports type generalisation on declaration of fields and variables (VD), method parameters (MPT) and method return types (MRT), but not on method exception types (MET). The MET type of dependency is currently not supported.

In a source class, a dependency to the target type T may occur as VD, MPT and MRT simultaneously. The *generalize* refactoring must be applied to all such dependency categories in order to remove the dependency to T. If any of the three dependency types cannot be generalised, the whole refactoring fails. In our approach, we choose a common supertype and replace the target type T with that supertype for each dependency category. The following steps describe the process through which a specific supertype is selected for the target type T.

- 1) For each VD, MPT, and MRT dependency to T, compute the most general possible type and add it to a set of *PossibleSuperTypes* in no particular order. This is done using the standard Eclipse refactoring class *ChangeTypeRefactoring*.
- 2) Next, a common supertype is selected from the set *PossibleSuperTypes*. This supertype must be the least general one among the set to avoid API access problems. Due to multiple interface inheritance, a least general supertype does not always exist. In this case, the precondition fails and the refactoring is abandoned. For the purpose of choosing a least common type, the original supertype hierarchy is extracted (*AllSuperTypes*) using Eclipse JDT Core API *ITypeHierarchy*. This API returns an array of all supertypes of the current type in bottom-up order. This means `java.lang.Object` would be the last element in the list.
- 3) Iterate over *AllSuperTypes* from the first type to

the last. If the *type* exists in the set *PossibleSuperTypes*, choose this type as a supertype to replace the old type in all identified places of the source class and stop the iteration. This would select the least general type among the set of *PossibleSuperTypes*.

### B. Locator Refactoring

The *Locator* refactoring is built on top of the standard Eclipse refactoring *Introduce Factory*. The *Locator* refactoring replaces all constructor invocations of the target type in the source class with the factory pattern. This refactoring creates a factory method in the factory class, which returns an instance of the target type. A class called `registry.ServiceLocator` is created for each program. This class serves as a global factory class for the instantiation purpose. Consider the following example:

```

1 //Before Locator Refactoring
2 public class A {
3     private B bObj = null;
4     public void m() {
5         bObj = new BImpl(); } ... }
6 //After Locator Refactoring
7 public class A {
8     private B bObj = null;
9     public void m() {
10        bObj = registry.ServiceLocator.
11            createBImpl(); } ... }
  
```

The implementation of the service locator refactoring can be further improved by using other APIs such as Java service loader or Java reflection. This will completely remove dependency from the target class and the service locator class. Therefore, while computing antipattern instances when we rebuild the dependency graph to measure the effects of the refactorings, these types are ignored.

### C. Inline Refactoring

The *Inline* refactoring is built on top of the standard Eclipse move refactoring. The *Inline* refactoring determines all static members of the target class and moves

those members to the source class. Delegate methods are created in the target class for moved static methods. Static members to be inlined can be recursive. For example, a static method calls another static method within the same class. In that case both methods will be inlined. However, the Inline refactoring does not follow references beyond the class boundary, e.g. methods invoked on supertypes.

#### D. Move Class Refactoring

The Move refactoring is built on top of the standard Eclipse refactoring `MoveRefactoring`. In this refactoring a class is moved from one package to another. First, the algorithm performs move refactoring on the dependency graph to evaluate its impact. Next, it is executed on the code level.

#### E. Composite Refactoring

A composite refactoring is a combination of individual refactorings. The standard Eclipse refactoring framework doesn't support the composite refactoring. Occasionally, it is essential to use a composite refactoring to perform a task, e.g. to remove a dependency to a type, it may be required to use several individual refactorings such as generalize, locator, and inlining. For example, `ArrayList list = new ArrayList();` can be refactored as `List list = ServiceLocator.createList();` This refactoring required the combination of type generalisation and service locator.

Since our individual refactorings are built on top of standard eclipse refactorings, each refactoring executes changes on its own working copy of the compilation unit. This makes it difficult to perform composite refactorings. The AST modifications overlap when changes in multiple working copies are committed to the original source code. In order to deal with the synchronization problem, we attached an interface `IElementChangeListener` to the `CareRefactoring` class. This listener interface receives notification about any changes made to Java elements and informs other refactorings in the queue to synchronize their working copies of the ASTs.

## VIII. RESULTS

### A. Impact of Refactorings on Instance Count Metric

In order to evaluate our approach, we have used the Qualitas Corpus version 20101126r [16]. This version of the corpus has 106 programs. We excluded 16 programs based on two criteria: size and configurability in Eclipse. The JRE-1.6 was skipped due to the large number of classes and relationships (17253, 173911 respectively). The other 15 skipped programs were plugin based, i.e. each program had multiple plugins/modules that were difficult to normalise into a single Eclipse project<sup>6</sup>. However, we believe that the resulting dataset

<sup>6</sup>The skipped programs are: eclipse-3.6, netbeans-6.9.1, exoport-v1.0.2, gt2-2.7-M3, ivatagroupware-0.11.3, jboss-5.1.0, jtopen-7.1, maven-3.0, myfaces-core-2.0.2, roller-4.0.1, spring-framework-3.0.5, struts-2.2.1, tapestry-5.1.0.5

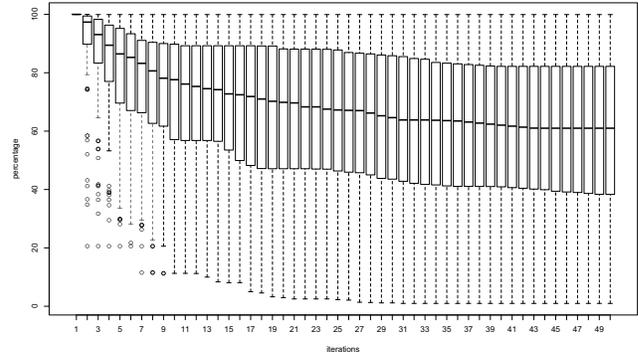


Fig. 2: Decrease in instance count after refactorings

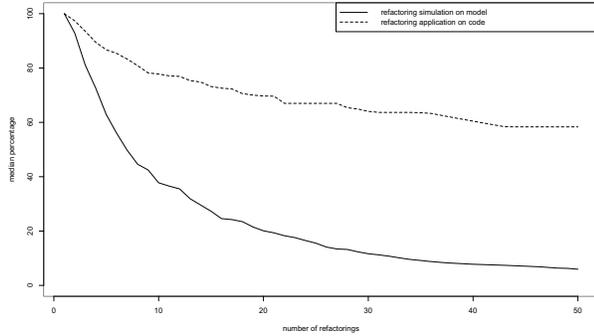
of 92 programs still has a wide variety of real-world applications sufficient for evaluation purposes. Out of those 92, one program (jasml-0.10) had zero instances of all four antipatterns.

Figure 2 shows the decrease in antipattern instances over all programs after increasing numbers of refactoring are performed. This is a boxplot chart where the box represents 50% of the observations. The horizontal line in the box represents the median. In this chart, we scale each observation by the number of original instances for its corresponding program. The value on the x-axis represents the number of refactorings executed on the code level. It is notable that after 20 refactorings the median value over all observations drops below 70%. This means, for more or less half of the programs, only 20 refactorings are required to remove 30% of instances. Similarly, only 8 refactorings are required to remove the first 20% of instances. The boxplot also shows some outliers. These include programs where the total number of antipattern instances dropped close to zero in a few initial steps, such as, jag-6.1, jsXe-04, freecs-1.3.20100406. In some programs the instance count remained unchanged, for example, jFin-DateMath-R1.0.1, fit java-1.1, and javacc-5.0.

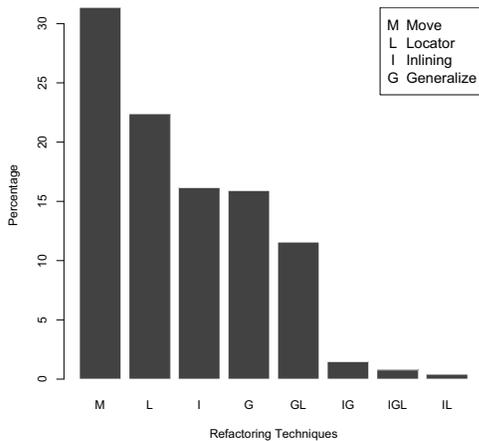
### B. Refactoring Simulation on Model vs Refactoring Application on Code

Figure 3 shows the comparison between the refactoring simulation on the graph level and the refactoring application on the code level. The two lines represent the median value. This figure allows us to assess the extent to which high impact refactorings can be automated.

There are 50 iterations and for each iteration, the refactoring on the graph level is performed by removing the highest-scored edge without checking any pre and postconditions. This is why we get a very steep curve representing the decline in the number of instances. However, not every edge can be removed on the code level, as some pre or postconditions may fail. Consequently, our algorithm iterates over a list of high-scored edges until it reaches one which passes all conditions. Therefore, the curve generated by the refactorings on the code level is less steep than graph level simulation.



**Fig. 3:** Decrease in no. of instances: comparison on model and code levels



**Fig. 4:** Refactoring types applied

This defines a lower bound of what is possible. The result of code level refactorings can be further improved by weakening some preconditions, such as by allowing move refactoring to change visibility where required or by following leaking references in case of type generalization refactoring.

### C. Refactoring Types Applied

Figure 4 shows different types of refactorings that were successfully applied across all programs. On top is the move refactoring, which is applied in 31% cases. The composite refactorings constitute to 14% of the total refactorings applied.

### D. Examples

The first critical dependency identified in Findbugs-1.3.9 is the reference from `edu.umd.cs.findbugs.ShowHelp` to `edu.umd.cs.findbugs.gui.FindBugsFrame`. This dependency is caused because the `ShowHelp` class invokes a static method `showSynopsis()` on the target type `FindBugsFrame`, as shown in the listing 3. This particular dependency is also a reference from a core package `edu.umd.cs.findbugs` to a presentation package `edu.umd.cs.findbugs.gui` and therefore violates the

design principle that logic should not depend upon presentation. This dependency was involved in 113579 SCD instances, 185 STK instances and 1811 AWD instances. Therefore, removing this dependency reduced the total number of instances from 277091 (100%) to 161517 (58%).

```

1 //Before Refactoring
2 public class ShowHelp {
3     public static void main(String[] args) { ...
4         FindBugsFrame.showSynopsis(); ...
5     } ... }

```

**Listing 3:** Source code of ShowHelp class

The refactored version of the old code causing the dependency is shown in listing 4. A delegate method is created in the target type `FindBugsFrame`, which invokes the `showSynopsis()` method in the source class.

```

1 //After Inline Refactoring
2 public class ShowHelp {
3     public static void main(String[] args) { ...
4         showSynopsis(); ... }
5     public static void showSynopsis() {
6         System.out.println("Usage: findbugs [
7             general options] [gui options]");
8     } }

```

**Listing 4:** Source code of ShowHelp after refactoring

Another example of a critical dependency was identified in `JMoney-0.4.4`. This is a reference from `net.sf.jmoney.Start` to `net.sf.jmoney.gui.MainFrame`. This dependency was involved in 57 SCD instances. In this case, our algorithm executed a move refactoring where the class `Start` was moved to the `net.sf.jmoney.gui` package, removing 26% of the total instances. The class `Start` is an entry point to run the application. This class invokes `net.sf.jmoney.gui.MainFrame` to load the application's GUI. Therefore, this class is a good candidate to be moved to the `gui` package.

## IX. CONCLUSION

In this paper, we have presented an algorithm which removes critical dependencies from Java programs. We have developed an Eclipse plugin, which automates the refactoring process. This plugin is able to perform composite refactorings on the source code level. In general, it is not possible to remove all types of dependencies automatically. However, there are certain cases where it is possible to safely remove undesired dependencies. All code level changes performed by the plugin are recorded to be reviewed by a developer of the program. In order to verify the correctness of refactored programs, we have manually tested 5 programs. All testcases produced same results before and after refactoring. The plugin computes design quality metrics (package design and strongly connected component metrics) before and after the refactoring process. This allows developers to evaluate the impact of refactorings using other standard metrics. The results presented can be further improved by weakening

preconditions, such as following leaking references, and by strengthening the implementation of basic code level refactorings, such as the move refactoring.

## REFERENCES

- [1] R. Martin, "Object oriented design quality metrics: An analysis of dependencies," Report on object analysis and design, 1994. [Online]. Available: <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>
- [2] D. L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings of the 3rd international conference on Software engineering*, ser. ICSE '78. Piscataway, NJ, USA: IEEE Press, 1978, pp. 264–277.
- [3] W. Stevens, G. Myers, and L. Constantine, *Structured design*. Upper Saddle River, NJ, USA: Yourdon Press, 1979, pp. 205–232.
- [4] H. Melton and E. Tempero, "An empirical study of cycles among classes in java," *Empirical Softw. Engg.*, vol. 12, pp. 389–415, August 2007.
- [5] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "Barriers to modularity: An empirical study to assess the potential for modularisation of java programs," in *In Proceedings QoSA'10*, 2010.
- [6] A. J. Riel, *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley, Apr. 1996.
- [7] M. Sakkinen, "Disciplined Inheritance," *ECOOP'89: Proceedings of the 1989 European Conference on Object-Oriented Programming*, vol. -, pp. 39–56, 1989.
- [8] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and D. Kopylenko, *Professional Java Development with the Spring Framework*. Birmingham, UK, UK: Wrox Press Ltd., 2005.
- [9] R. Vanbrabant, *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress, 2008.
- [10] C. Walls, *Modular Java: Creating Flexible Applications with Osgi and Spring*. Pragmatic Bookshelf, 2009.
- [11] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "On the existence of high-impact refactoring opportunities in programs," in *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ser. ACSC '12. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2012, pp. 37–48.
- [12] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.
- [13] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.
- [14] S. M. A. Shah, J. Dietrich, and C. McCartin, "Making smart moves to untangle programs," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 359–364.
- [15] S. M. A. Shah, J. Dietrich, and C. McCartin, "On the automated modularisation of java programs using service locators," in *Proceedings of the 11th international conference on Software Composition*, ser. SC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 132–147.
- [16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," *Proc. APSEC*, vol. -, pp. -, 2010.
- [17] M. O'Keefe and M. O'Kinneide, "Search-based software maintenance," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 249–260, 2006.
- [18] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1106–1113.
- [19] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '12. New York, NY, USA: ACM, 2012, pp. 49–58.
- [20] P. Mayer, A. Meissner, and F. Steimann, "A visual interface for type-related refactorings," in *WRT'07*, 2007, pp. 5–6.
- [21] M. Streckenbach and G. Snelting, "Refactoring class hierarchies with kaba," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 315–330.
- [22] M. Bach, F. Forster, and F. Steimann, "Declared type generalization checker: An eclipse plug-in for systematic programming with more general types," in *FASE 2007*, ser. LNCS, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer, 2007, pp. 117–120.
- [23] F. Steimann, W. Siberski, and T. Kuhne, "Towards the systematic use of interfaces in java programming," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, ser. PPPJ '03. New York, NY, USA: Computer Science Press, Inc., 2003, pp. 13–17.
- [24] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004. [Online]. Available: <http://martinfowler.com/articles/injection.html#InversionOfControl>
- [25] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [26] M. Feathers, *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.
- [27] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, no. RapidPosts, pp. 347–367, 2009.
- [28] J. Dolby, "Automatic inline allocation of objects," in *Proceedings of the ACM SIGPLAN 1997 conference on PLDI'97*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 7–17.
- [29] J. Dolby and A. A. Chien, "An automatic object inlining optimization and its evaluation," in *In PLDI 2000*. ACM Press, 2000, pp. 345–357.
- [30] Y. Ben Asher, T. Gal, G. Haber, and M. Zalmanovici, "Refactoring techniques for aggressive object inlining in java applications," *Automated Software Engineering*, vol. 19, pp. 97–136, 2012. [Online]. Available: <http://dx.doi.org/10.1007/s10515-011-0096-x>
- [31] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-based improvement of subsystem decompositions," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1045–1051.
- [32] E. Hautus, "Improving java software through package structure analysis," in *IASTED International Conference Software Engineering and Applications*, 2002.
- [33] H. Abdeen, S. Ducasse, H. A. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in *WCRE'09*, 2009, pp. 103–112.
- [34] H. Abdeen, S. Ducasse, D. Pollet, and I. Alloui, "Package fingerprints: A visual summary of package interface usage," *Inf. Softw. Technol.*, vol. 52, pp. 1312–1330, December 2010.
- [35] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ser. ACSC '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 35–41.
- [36] J. Lakos, *Large-scale C++ software design*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996.
- [37] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.
- [38] F. Steimann, P. Mayer, and A. Meibner, "Decoupling classes with inferred interfaces," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1404–1408.
- [39] RefactoringCatalog, "Move Class Refactoring," 1999. [Online]. Available: <http://refactoring.com/catalog/index.html>

# Reducing the Energy Consumption of Mobile Applications Behind the Scenes

Young-Woo Kwon and Eli Tilevich  
 Department of Computer Science, Virginia Tech  
 Email: {ywkwon,tilevich}@cs.vt.edu

**Abstract**—As energy efficiency has become a key consideration in the engineering of mobile applications, an increasing number of perfective maintenance tasks are concerned with optimizing energy consumption. However, optimizing a mobile application to reduce its energy consumption is non-trivial due to the highly volatile nature of mobile execution environments. Mobile applications commonly run on a variety of mobile devices over mobile networks with divergent characteristics. Therefore, no single, static energy consumption optimization is likely to yield across-the-board benefits, and may even turn to be detrimental in some scenarios. In this paper, we present a novel approach to perfective maintenance of mobile applications to reduce their energy consumption. The maintenance programmer declaratively specifies the suspected energy consumption hotspots in a mobile application. Based on this input, our approach then automatically transforms the application to enable it to offload parts of its functionality to the cloud. The offloading is highly adaptive, being driven by a runtime system that dynamically determines both the state-to-offload and its transfer mechanism based on the execution environment in place. In addition, the runtime system continuously improves its effectiveness due to a feedback-loop mechanism. Thus, our approach flexibly reduces the energy consumption of mobile applications behind the scenes. Applying our approach to third-party Android applications has shown that it can effectively reduce the overall amount of energy consumed by these applications, with the actual numbers ranging between 25% and 50%. These results indicate that our approach represents a promising direction in developing pragmatic and systematic tools for the perfective maintenance of mobile applications.

## I. INTRODUCTION

As mobile applications deliver increasingly complex functionality, mobile devices are rapidly overtaking the personal computer as a primary computing platform for an increasing number of users [6]. Because of the battery constraints of mobile devices, energy efficiency—fitting an energy budget and maximizing the utility of applications under given battery constraints—has become an important software design and maintenance consideration [14]. Traditionally concerned with performance and memory usage optimization, perfective maintenance now has to address the challenges of optimizing energy consumption, with existing perfective maintenance techniques being mostly inapplicable.

Although the field of optimizing software energy consumption is broad and diverse, existing solutions primarily focus on the hardware, operating system, and network layers. At the software layer, an energy optimization technique that has received a substantial attention from the research

community is *cloud offloading* [4], [1], [17], [8]. This optimization partitions mobile applications, so that their energy intensive functionality is executed in the cloud, without draining the battery. Existing cloud offloading techniques determine the energy intensive functionality statically, and partition mobile applications accordingly. However, to achieve maximum benefit, cloud offloading must take into consideration the hardware capacities of the mobile device running the application as well as the characteristics of the mobile network connecting the mobile device to the cloud. In other words, the offloading decisions should be made dynamically at runtime and continuously adjusted in response to the fluctuations in the mobile execution environment. Lacking this level of flexibility, existing offloading schemes are inapplicable as a general energy optimization approach for perfective maintenance.

In this paper, we present a novel offloading approach that combines the advantages of the prior state of the art both in partitioning mobile applications and in dynamically adapting mobile execution targets in response to fluctuations in network conditions. Our approach is realized as the following two major technical solutions: (1) a multi-target offloading program transformation that automatically rewrites a centralized program into a distributed program, whose local/remote distribution is determined dynamically at runtime; (2) a runtime system that determines the required local/remote distribution of the resulting distributed program based on the current execution environment. Combining these two solutions can effectively reduce the amount of energy consumed by mobile applications without having to change their source code by hand, thus optimizing them behind the scenes.

From the maintenance process perspective, our approach works as follows. The maintenance programmer marks the methods suspected of being energy consumption hotspots. Then, a series of program analysis techniques validates the programmer's input and automatically rewrites the application into a distributed application, whose local and remote parts can be flexibly determined at runtime. The flexibility in determining the distribution patterns at runtime is enabled through an elaborate checkpointing mechanism. Depending on the runtime execution environment, different portions of a program's state can be checkpointed and transferred across the network as required by the offloading strategy in place. The offloading strategy is determined by the runtime

system, whose responsibilities include: (1) managing a connection between the client and the server, (2) continuously estimating the energy consumed by the mobile device, (3) calculating which offloading strategy should be followed, (4) synchronizing the checkpointed state transferred between the server and client, and (5) ensuring resilience in the presence of failure due to network disconnections.

In our experiments, we have applied our approach to optimize the energy consumption of third-party, real-world Android applications. All the subject applications not only reduced their energy consumption, but also maintained their original performance characteristics. Our adaptive, multi-targeted cloud offloading approach can effectively reduce the amount of consumed energy. Specifically, our benchmarks and case studies demonstrate that our approach can reduce the overall energy budget of a typical mobile application by an amount ranging between 25% and 50%.

Based on our results, the technical contributions of this paper are as follows:

- 1) **A multi-target offloading program transformation** that makes it possible to postpone until the runtime the decision about which parts of the application should be executed locally or remotely.
- 2) **An adaptive cloud offloading runtime system** that determines optimal offloading strategies for the partitioned applications.
- 3) **An empirical evaluation of multi-target offloading** that shows the technique effective in reducing the energy consumed by real-world, third-party mobile applications.

The rest of this paper is structured as follows. Section II defines the problem that our approach aims at solving and introduces the concepts and technologies used in this work. Sections III and IV describe and evaluate our approach, respectively. Section VI compares our approach to the existing state of the art. Section V presents perfective maintenance guidelines for effective cloud offloading, and Section VII presents concluding remarks.

## II. PROBLEM DEFINITION AND TECHNICAL BACKGROUND

In this section, we provide a technical background both for the problem our approach is intended to solve and for the major technologies our approach uses.

### A. Problem Definition

The work presented here is motivated by an insight we have recently derived from an experimental study of the impact of distributed programming mechanisms on energy efficiency [9]. After discovering that the network conditions in place can significantly affect how much energy is spent to transfer the same data, we have created guidelines for mobile application designers to transmit data in a fashion that consumes the least amount of energy for a given mobile

network. Because transferring the same data over WiFi, 3G, or 4G networks consumes different amounts of energy, an optimal offloading mechanism should be adaptive, transferring varying amounts of state depending on the network conditions in place.

Figure 1 shows an Cloud Offloading Energy Consumption Graph (CO-ECG), a novel program analysis data structure that we introduced to model how much energy will be consumed under different offloading scenarios. The nodes of the CO-ECG represent program components, encapsulated units of functionality that can be offloaded to the cloud. Each node is labeled with an approximate amount of energy consumed by the CPU to execute the functionality of the component and its successor components in the graph. The edges represent the communication between the components, with the labels showing how much energy will be consumed by the mobile device to transmit the data between the connected components.

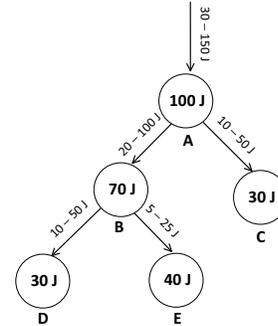


Figure 1. An cloud offloading energy consumption graph.

In this particular CO-ECG, component A consumes approximately 100 joules during a typical execution, thus becoming a viable candidate to be offloaded to the cloud. Because component A calls components C and B, which in turn calls components D and E, its energy consumption is the sum of the energy consumed by all the successor components in the graph. We assume that the energy spent on executing the offloaded functionality in the cloud is free, as it does not exhaust any battery power of the mobile device. If component A is offloaded, then transmitting the necessary data to it across the network enabling it to execute remotely would consume between 30 and 150 joules depending on the type of the network available to transmit the data. In other words, under some network conditions, offloaded execution will end up using more energy than executing component A on the mobile device. Because the type of network available is only known at runtime, the offloading decisions must be dynamic to be able to optimize the amount of consumed energy under all runtime conditions.

In this paper, we present a solution that can solve the problem discussed above. We call our solution *adaptive, multi-target cloud offloading* (AMCO). A special program

transformation creates a distributed client/server application, whose client and server functionalities are determined dynamically at runtime. As a specific example of using the CO-ECG above, when operating over a 3G network, components C and D can be offloaded, while when operating over a 4G network, only component E can be offloaded. Finally, when operating over WiFi, components A or B can be offloaded.

More specifically, marking a method as an energy hotspot creates an offloading specification, in which various portions of the call graph rooted in the marked method can be offloaded as required by the runtime conditions. Because it takes more energy to transfer data across limited networks, an optimal offloading strategy needs to trade the energy potentially saved by moving the execution to the cloud with the energy consumed by moving the data (i.e., program state) to support the offloading. Our program transformation makes it possible to offload any subgraph of the CO-ECG, while our runtime system triggers the most beneficial (i.e., saving the most battery power) offloading for the runtime execution environment in place.

### B. Technical Background

The technical concepts behind our approach include cloud offloading, energy consumption patterns, and program analysis. We describe these technologies in turn next.

1) *Cloud Offloading*: Cloud offloading is a mobile application optimization technique that makes it possible to execute the application’s energy intensive functionality in the cloud, without draining the mobile device’s battery. Cloud offloading is typically implemented as a program partitioning transformation that splits a mobile application into two parts: client running on a mobile device and server running in the cloud; all the communication between the parts is conducted via a middleware mechanism such as XML-RPC. Thus, cloud offloading is a special case of automated program partitioning—distributing a centralized program to run across the network using a compiler-based tool transform a centralized program [22] or migrating execution between different application images at the OS level [17], [2]. The promise of cloud offloading is demonstrated by the proliferation of competing approaches to this technique in the literature. CloneCloud [1] offloads execution at the thread level, while Cloudlet [17] offload at the VM level. MAUI [4] offloads through application partitioning at the method level. In our prior work on cloud offloading [8], we partition applications without destroying their ability to execute locally. All of these prior cloud offloading techniques share the goal of reducing the energy consumed by mobile devices. The approach presented in this paper adopts many of the techniques above to automatically transform mobile applications without any changes to their source code and to synchronize program states between partitions. However, our approach’s goal is to improve on the efficiency of the prior cloud offloading technique by postponing the

offloading decisions until the runtime, when a feedback-loop mechanism can determine which amount of offloading is optimal.

2) *Energy Consumption Patterns in Mobile Applications*: Network communication constitutes one of the largest sources of energy consumption in a mobile application [15], [9]. According to a recent study, network communication consumes between 10 and 50% of the total energy budget of a typical mobile application [13]. Specifically, in our prior research [9], we measured and analyzed how middleware can significantly affect a mobile application’s energy consumption. Our experiments assessed the energy consumption of passing varying volumes of data over networks with different latency/bandwidth characteristics. Then, we isolated how mobile applications consume energy to infer their common energy consumption patterns. The experimental results and systematic analysis conducted through that research inspired us to initiate the work presented in this paper.

3) *Program Analysis*: Program analysis codifies a set of techniques to infer various facts about the source code to be leveraged for optimization and transformation. Class hierarchy analysis (CHA) constructs a call graph in object-oriented languages. Dataflow analysis determines how program variables are assigned to each other [18]. Side-effect free analysis [16] determines whether a method changes the program’s heap. In this work, we use CHA to compute the functionality to offload and the program state to transfer for a given offloading. To select optimal offloading strategies, we combine dataflow and side-effect analyses. Based on the results, a bytecode enhancer then rewrites the application without changing its source code. We used Soot [23] to implement our program analysis and transformations.

## III. OFFLOADING ENERGY INTENSIVE FUNCTIONALITY

In this section, we present adaptive, multi-target cloud offloading (AMCO). We start by giving an overview of the approach and then describe its major parts in turn. We conclude with discussing the approach’s applicability and limitations.

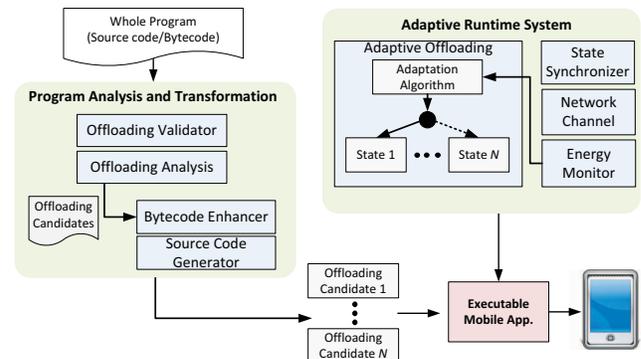


Figure 2. Overall procedure for adaptive, multi-target cloud offloading.

### A. Approach Overview

Figure 2 shows the workflow of AMCO. The approach consists of two major parts—a program analysis-guided partitioning transformation and an adaptive runtime system. The AMCO programming model is straightforward: the programmer marks the components suspected of being energy hotspots. In AMCO, components can be defined at any level of program granularity, with the smallest being individual methods and the largest a collection of packages. To mark hotspot components, AMCO provides a special Java annotation `@OffloadingCandidate`; this information can also be specified through an XML configuration file. Based on this input, an analysis engine first checks whether the specified component can be offloaded as well as any of its subcomponents (i.e., successors in the call graph). The engine also calculates the program state, to be transferred between the remote and local partitions, that would need to be transferred to offload the execution of both the entire component or of any of its subcomponents. A bytecode enhancer then generates the checkpoints that save and restore the calculated state for the entire hotspot components as well as for each of its subcomponents. At runtime, an adaptive runtime system continuously monitors the energy consumed by each offloading candidate component, broken down for each of its constituent subcomponents. Based on the estimated energy consumption, the runtime offloads those subcomponents whose cloud-based execution would save the highest amount of energy for the network conditions in place. The runtime also synchronizes remote and local states in place (while preserving the aliasing in both the local and remote heaps). Yet another responsibility of the runtime system is fault tolerance—handling temporary network disconnection and volatility.

### B. Program Analysis and Transformation

To create a program analysis heuristic that can calculate the program transformations enabling multi-target offloading, we extended our prior heuristic for static offloading [8].

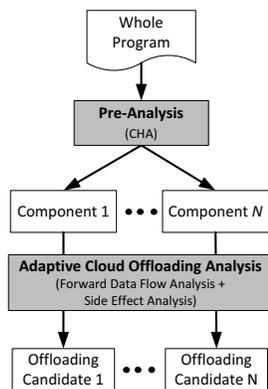


Figure 3. Program analysis for adaptive, multi-target cloud offloading.

Figure 3 shows the analysis procedure that includes constructing a call graph by using class hierarchy analysis and identifying the state-to-be-transferred or synchronized by using forward dataflow and side-effect analyses. A pre-analysis step collects all the components and subcomponents marked with `@OffloadingCandidate`, and then the main analysis identifies the state that is needed to be transferred for each offloading scenario. One technical advantage of the main analysis is that it reduces the amount of state that has to be transferred across the network. The necessity to transfer large data volumes across the network can quickly negate the energy consumption benefits afforded by remote offloading. To avoid having to transmit the entire state, our approach leverages these forward dataflow and side-effect analyses to reduce the transferred state’s size, thus rendering state transfer practical for energy optimizations. The algorithm examines assignment and invocation statements to determine whether the current state has changed during the cloud offloading.

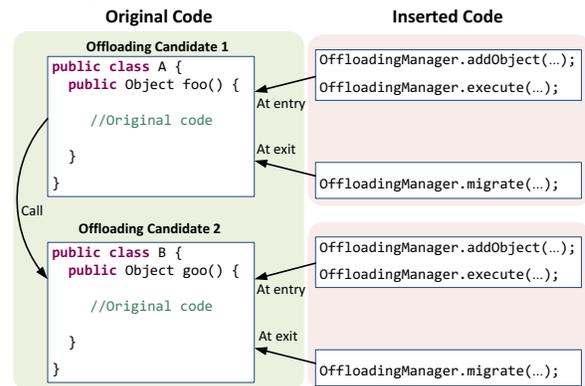


Figure 4. An example of program transformation.

Once offloading candidates are identified, the bytecode enhancer transforms them to be able to run their hotspot components and subcomponents on the client and the server as needed to realize a given offloading scenario. Then, the adaptive runtime system monitors the runtime execution environment and determines an optimal offloading candidate. Figure 4 shows how the original code of a centralized mobile application is transformed. The bytecode enhancer inserts the code that can checkpoint and restore the necessary program state at the entry and exit of the potentially offloaded methods, respectively. These methods include offloading candidates and their successors in the CO-ECG.

### C. Adaptive Runtime System

Figure 5 shows the design of the AMCO adaptive runtime system that comprises three main components: (1) cloud offloading prediction and steering, (2) energy consumption estimation, and (3) cloud offloading processing. By continuously monitoring the execution environment, the runtime system intelligently correlates the collected information to suggest optimal offloading strategies.

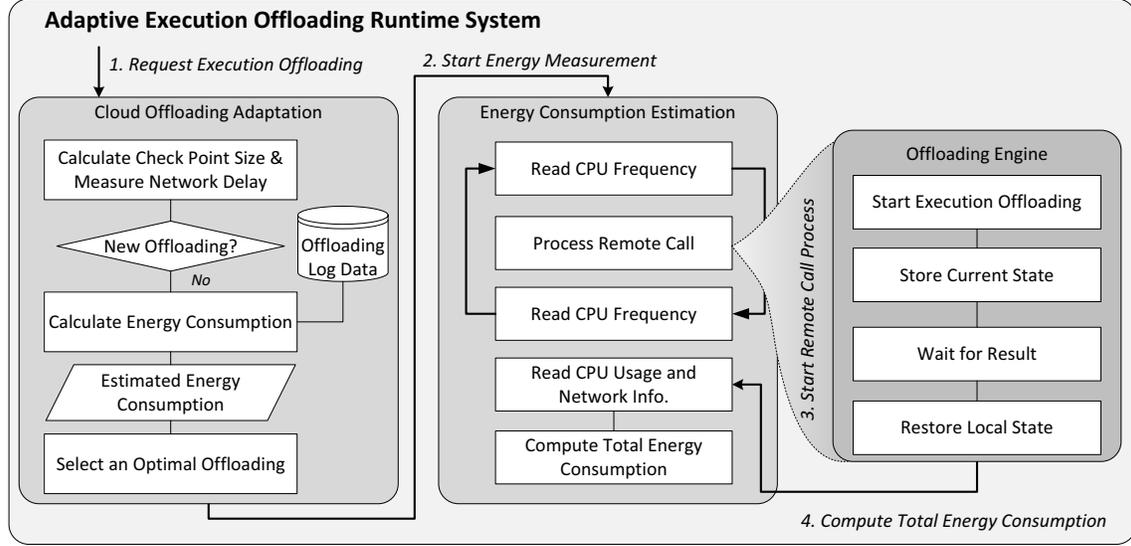


Figure 5. Process details of adaptive execution offloading.

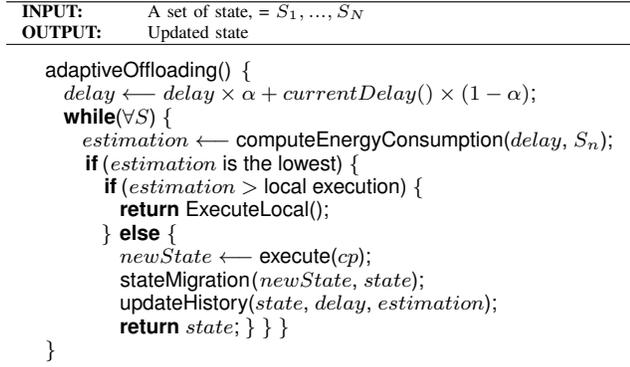


Figure 6. Adaptive, multi-target cloud offloading operation.

1) *Cloud Offloading Prediction and Steering*: Figure 6 shows the procedure for predicting and steering the multi-target cloud offloading. The module predicts the future energy consumption by analyzing the collected runtime execution environment and then selects the offloading strategy that would yield the lowest future energy consumption. First, the future energy consumption is computed using delay, which is measured by sending a probe packet to the offload server. Then, the current delay is recomputed by giving a weight to the mostly recently obtained value (i.e.,  $delay = delay \times \alpha + currentDelay() \times (1 - \alpha)$ ), thereby avoiding a delay spike as well as partially reflecting the latest delay value. Finally, the module calculates the offloading strategy to suggest by picking the smallest predicted future energy consumption from all the available offloading candidates (i.e., subgraphs of the CO-ECG).

2) *Energy Consumption Estimation*: The energy consumption estimation module estimates the energy consumption before and after the offloading operation. To estimate

the performed cloud offloading, we only compute the energy consumption by CPU and network communication as follows:

$$E = \{P_{cpu\_freq}^{active} \times (T_{cpu}^{user\_time} + T_{cpu}^{sys\_time}) + (P_{net}^{active} \times T_{net}^{active}) + (P_{net}^{idle} \times T_{net}^{idle})\} \times V$$

where  $P_{cpu\_freq}^{active}$  is the power consumed by the CPU. Modern CPUs feature speed-step, a facility that allows the clock speed of a processor to be dynamically changed by the operating system, with different levels of power consumed at each clock speed. For example, Samsung Galaxy S III's AP provides five steps, ranging from 302.4 MHz to 1512 MHz, and the amount of power consumed at each speed ranges from 55mA to 577mA.  $T_{cpu}^{user\_time}$  and  $T_{cpu}^{sys\_time}$  are user and system times taken by the offloading operation, and they are obtained by consulting the statistics provided by the operating system (e.g., `\proc\pid\stat`).  $V$  is current voltage, which is obtained by using battery-related APIs (e.g., `class BatteryStat` in Android OS).  $P_{net}^{active}$  and  $P_{net}^{idle}$  are the amount of energy that the network processor consumes during the active and idle phases, respectively. For example, the active/idle energy consumption ratio for Samsung Galaxy S III is 96 mA/0.3 mA during WiFi communication, and 250 mA/3.4 mA during mobile communication (e.g., 4G). Finally,  $T_{net}^{active}$  and  $T_{net}^{idle}$  are active and idle time periods during the cloud offloading operation, measured at runtime. These device- and execution-specific values are used to compute the amount of energy consumed during each offloading optimization.

Another important responsibility of the runtime system is to predict future energy consumption. To predict the energy that is likely to be consumed during an offloading optimization, it correlates the previously measured energy

consumption and the current execution environment. Having obtained the current values of network delay, connectivity type, CPU frequency, and voltage, the future energy consumption is computed as follows:

$$E_{est} = \{E_{cpu}^{avg} + (P_{net}^{active} \times T_{net}^{est\_active}) + (P_{net}^{idle} \times T_{net}^{est\_idle})\} \times V$$

where  $E_{est}$  is the predicted future energy consumption,  $E_{cpu}^{avg}$  is the average energy consumption of the given remote call, and  $T_{net}^{est\_active}$  is the estimated communication time, which is computed by using the offloaded data size and delay, respectively. Finally, based on the estimated communication time and prior executions, the runtime system predicts the future energy consumption. The computed energy consumption is used for determining an optimal offloading strategy.

3) *Cloud Offloading Engine*: The cloud offloading engine manages a connection between the offload server and the mobile client, synchronizes the checkpointed state, and provides resilience in the presence of failure due to network volatility. The checkpointed state is synchronized by means of copy-restore, an advanced semantics introduced into remote method call middleware with the goal of passing as parameters liked data structures (e.g., linked lists, trees, and maps) [21]. Copy-restore copies all reachable state to the server and then overwrites the client's state with the server modified data in-place. To adapt to operating over cellular networks with limited bandwidth, we modified the original copy-restore implementation to use sparse arrays, which encode **null** values space efficiently. Our implementation uses **null** values to mark the portion of the transferred state that has not been mutated during the offloaded operations.

Figure 7 demonstrates how the runtime system synchronizes the checkpointed state. Graph (a) depicts the mobile device's state to be transferred to the server. The runtime system transfers only the nodes that the analysis identified as being used by the server (nodes 2, 3, 4, and 7). Graph (b) depicts the server's state before it is synchronized with the transferred state. Nodes 2, 4, and 5 are updated with new values; node 3 is reassigned to point to node 7. Graph (c) shows the synchronized server state. In this example, the offloaded server execution assigns a new instance, node 8 to node 3, modifies nodes 3 and 5, and assigns the **null** value to node 6, with Graph (d) depicting the results. The mutated state is then transferred to the client and synchronized with its state depicted in Graph (e). Specifically, node 6 is removed, node 3 is reassigned to point to node 8, and nodes 3 and 5 are overwritten with new values. Graph (f) shows the client state after the synchronization.

#### D. Discussion

In this section, we discuss advantages and limitations of our approach, adaptive, multi-target cloud offloading.

1) *Advantages*: AMCO works with the standard, unmodified hardware/software stack; it employs bytecode engineering to transform programs and a lightweight runtime system

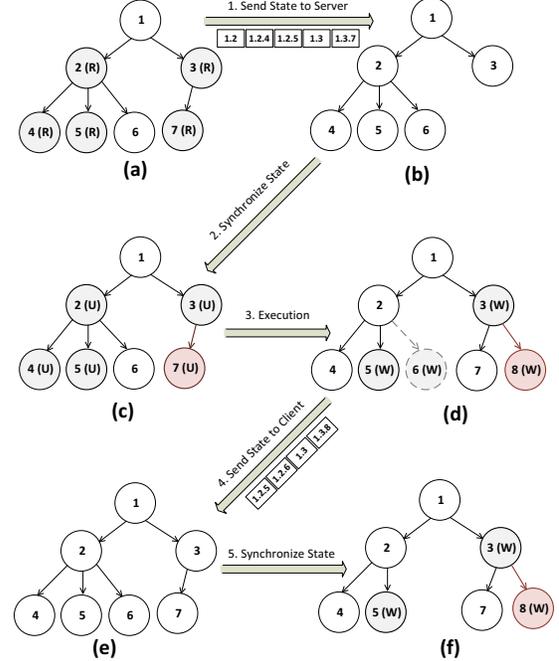


Figure 7. Procedure to synchronize two different state.

to dynamically steer and adapt the offloading. AMCO makes it possible to keep the maintained version of the mobile application's source code intact, as only the bytecode version is transformed. AMCO requires a minimal programming effort, limited to marking methods as energy hotspots. AMCO makes offloading decisions at runtime by monitoring the execution environment, thus discovering optimal offloading strategies. Finally, the AMCO offloading transformations do not preclude the mobile application from executing locally in the case of the network becoming disconnected.

2) *Limitations*: Despite its advantages, AMCO cannot be applied to optimize the energy consumption of all mobile applications. The AMCO approach is automated rather than automatic; it relies on the programmer to identify the energy-intensive methods. Offloading at the method boundaries, AMCO relies on the subject applications following the well-accepted principles of quality software design (i.e., encapsulation, modularity, loose coupling), so that the offloaded methods encapsulate distinct functionality loosely coupled from the rest of the application. The AMCO offloading model also requires that the lifetime of all the threads in the offloaded methods coincide with the methods' boundaries. That is, the offloaded methods are free to employ multiple threads, but all the threads are expected to terminate when the methods stop executing. As with all partitioning systems that rely on bytecode engineering, AMCO can only partition non-native (i.e., expressed exclusively in bytecode) state [20]. Finally, offloading can increase execution latencies, thus hurting the user experience for highly interactive applications.

#### IV. EVALUATION

We evaluated our approach through a micro benchmark and a larger case study. The results show that our runtime system reports actionable environmental information without imposing unreasonable performance and energy overheads. Also, our overall approach can effectively reduce the amount of consumed energy for well-engineered applications, with the introduced program transformations and runtime execution never causing the enhanced applications to exceed their original levels of energy consumption.

##### A. Micro Benchmark

The purpose of this micro benchmark is to understand the overhead imposed by the runtime system, whose responsibilities include monitoring the relevant fluctuations in the environment, estimating potential energy savings due to the possible offloading steps, and synchronizing heaps during the offloading.

We have based our test suite on the benchmarks originally proposed by the JavaParty project [7], which is used widely in benchmarking middleware implementations. These benchmarks comprise remote invocations with varying parameter sizes and types. Similarly, our test suite assumes that a client needs to execute some server methods, each of which takes parameters that differ in their size and type. Because the executed server methods have empty bodies, one can reasonably attribute the energy consumed during their invocation to the underlying runtime system.

1) *Experimental Setup*: The experimental setup has comprised a Motorola Droid (600 MHz CPU, 256 MB RAM, 802.11g, 3G) and Samsung Galaxy III (1.5 GHz dual-core CPU, 2 GB RAM, 802.11n, 4G) as the mobile device and Dell PC (3.0 GHz quad-core CPU, 8 GB RAM) as the offloaded server. The mobile device has communicated with the server through WiFi, 3G network, and 4G network. For the WiFi connection, we have experimented with two emulated network conditions that have the following respective round trip time (RTT) and bandwidth characteristics: 2 ms and 50 Mbps, typical for a high-end mobile network and 50 ms and 1 Mbps, typical for a medium-end mobile network. For the mobile connection, we used a 3G network for the Motorola Droid and a 4G network for the Samsung Galaxy. Table I shows the energy profiles, which are used to parameterize the runtime system.

Table I  
MANUFACTURER PROVIDED ENERGY PROFILES

	Samsung Galaxy III	Motorola Droid
CPU	1512.0 MHz: 577 mA	800.0 MHz: 280 mA
	1209.6 MHz: 408 mA	685.7 MHz: 236 mA
	907.2 MHz: 249 mA	571.4 MHz: 207 mA
	604.8 MHz: 148 mA	342.8 MHz: 165 mA
	302.4 MHz: 55 mA	228.5 MHz: 87 mA
	N/A	114.2 MHz: 66 mA
WiFi	96 mA	130 mA
	0.3 mA	4 mA
Mobile	250 mA	300 mA
	3.4 mA	3 mA

To a large degree, it is the hardware specifications of a mobile device that determine its energy consumption profile. For example, the nano-level process technology is known to reduce the amount of energy consumed by modern CPUs. Similarly, the latest network and radio chipsets have improved their energy efficiency. Therefore, device-specific characteristics play a pivotal role in estimating the energy consumed by a given mobile device.

2) *Energy Consumption Estimation*: First, we evaluated how accurately the runtime system can predict how much energy will be consumed in a given time interval. Figure 8 compares the energy consumption predicted by the runtime system and that estimated by our model based on the actual resource usage. The average error is 10.6 % and standard deviation is 21.3 %. When considering only 90 % data removing outliers, the average error is 8.5 % and standard deviation is 6.8 %. These results indicate that the runtime system can predict the future energy consumption accurately, with the discrepancies averaging 6-7%.

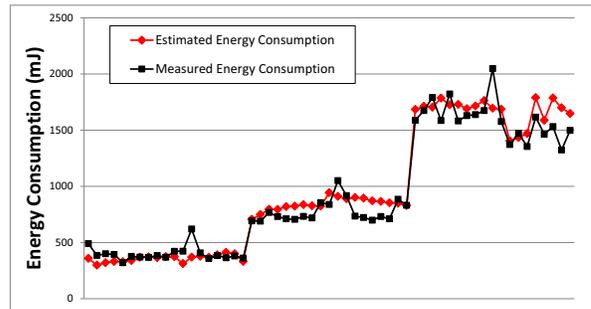


Figure 8. Energy consumption estimation.

3) *Overhead*: In this benchmark, we compared the total execution time and energy consumption of the baseline versions of the benchmarks with that of in the presence of the AMCO runtime system. The first graph of Figure 9 shows the total execution time measured on two devices. As one can see, the performance overhead is quite insignificant. In particular, the overhead for both devices never exceeds 100 ms and remains constant for all the measured data transfer sizes. The second graph shows the energy consumed by each device. As expected, the high-end device (Samsung Galaxy) consumes less energy than the low-end device (Motorola Droid). Despite the low-end device having a larger overhead than the high-end device, the actual number was 100 mJ, a negligible value in comparison with the total energy typically consumed by a mobile application.

##### B. Case Study

To determine if our approach is applicable to real-world mobile applications, we experimented with open source projects as our experimental subjects. Pocket chess<sup>1</sup> is a mobile chess game, whose AI engine, contained in

<sup>1</sup><http://code.google.com/p/pocket-chess-for-android/>

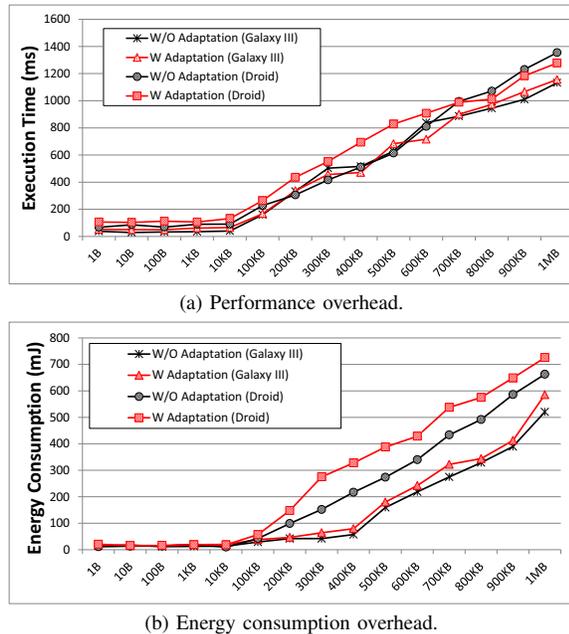


Figure 9. Performance and energy consumption overhead.

**class** SimpleEngine was marked as @OffloadCandidate. JJIL<sup>2</sup> is a face recognition application, whose recognition functionality, contained in **class** DetectHaarParam.

Figure 10 shows how the AMCO approach has reduced the amount of energy consumed by the subjects. For each subject, we present four graphs showing the amount of the energy consumed by typical, simple use cases. Specifically, for the chess application, we moved one randomly selected chess piece; for the face recognition application, we examined one image file for the presence of human faces. The use cases were performed under the following four optimization modes: (1) original centralized execution (baseline), (2) plain cloud offloading (offload the entire call tree rooted in the method marked with @OffloadCandidate), (3) same as (2) but with the transferred heap state minimized by means of program analysis, and (4) our approach, AMCO.

The optimized versions of the subject applications consumed less energy than their original and plain cloud offloading versions. A typical offloading strategy offloads to the cloud the heavy CPU processing required to calculate the next move, and transfers back only the new position for the piece to move. Because of this optimal architectural property of the chess application, its optimizations consume between 10% and 90% of the energy consumed by the original application. As the game proceeds, the optimized versions exhibit a constant rate of energy consumption, while the original version consumes an increasing amount of energy as the required AI processing intensifies. As expected, even the simplest energy optimization yields significant energy savings, without any tangible benefits afforded by using the

<sup>2</sup><http://code.google.com/p/jjil/>

AMCO adaptive runtime system.

In the case of the face recognition application, all three optimization strategies showed different levels of effectiveness. First, the plain cloud offloading approach only can save energy under the most favorable network environment (Wi-Fi). Second, optimizing the amount of transferred state shows consistent improvement in the amount of consumed energy. Third, when the runtime is instructed not to monitor the environment, the amount of consumed energy in the offloaded version actually exceeds that in the original, centralized version. Lastly, the adaptive offloading capability of our AMCO approach seems to be pivotal to saving the amount of consumed energy consistently. In particular, our AMCO approach optimizes the application to consume between 10% and 80% less energy than the original local version, as well as between 25% and 50% less energy than a state-of-the-art static cloud offloading approach [8].

### C. Threats to Validity

The experimental results above are subject to both internal and external validity threats. The internal validity is threatened by the manner in which the subject applications were run. Because the subject applications involve user interactions, their behavior and energy consumption depend on user actions (e.g., choosing a particular chess piece to move or taking a certain picture). These user interactions can heavily influence how much energy will be consumed.

The external validity is threatened by the mechanism employed by the AMCO runtime system to measure energy consumption. Rather than measuring the physical consumed energy directly, it estimates the energy consumption based on the actual resource usage information. As a result, the estimated energy consumption is likely to be less precise than those that would be measured through specialized hardware. In addition, the energy profiles that we used in the runtime system are provided by manufacturers, so that the accuracy of our measurements depends on the accuracy of the provided energy profiles.

## V. PERFECTIVE MAINTENANCE FOR EFFECTIVE CLOUD OFFLOADING

When creating the AMCO approach, we have observed a positive correlation between the desirable software engineering properties (i.e., strong modularity, high cohesion, low coupling, etc.) of the application and its amenability for the cloud offloading optimization. Based on these observations, we next present three perfective maintenance guidelines that programmer can follow to render their mobile applications better fit for effective cloud offloading optimizations.

*Applying the Split Method Refactoring:* A method is one of the earliest abstractions for separating concerns. A well-designed method should ideally contain a single coherent unit of functionality. In general, the larger the method, the less cohesive it is. As it turns out, large methods

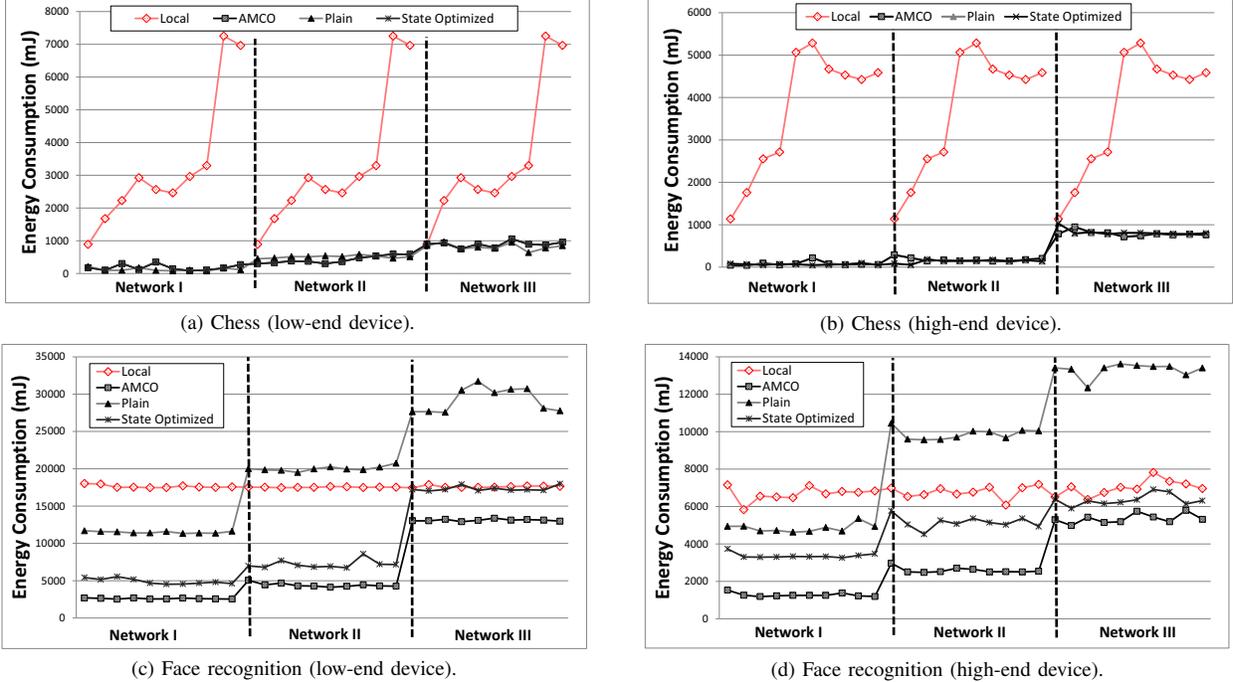


Figure 10. Energy consumption of the subject applications.

can be unwieldy as cloud offloading units, particularly in the case of fine-grained, adaptive offloading as in AMCO. In light of this observation, we recommend that the energy intensive methods marked with `@OffloadCandidate` be examined for their cohesion and then refactored if necessary. The Split Methods refactoring divides the functionality of a single method into separate methods calling each other. Smaller methods encapsulate conceptually coherent units of functionality. As a result, fine-grained, well-modularized mobile applications are not only desirable from the software comprehension perspective, but also are well amenable to perfective maintenance using cloud offloading.

*Encapsulating Native State:* Although native code usually renders the surrounding bytecode not modifiable [20], the programmer can guide the offloading tools by providing a bytecode API that accesses and synchronizes the portion of the state implemented in native code. Because native code may be impossible to analyze, it is the programmer’s responsibility to ensure that the wrapping API correctly synchronizes the native state without introducing any harmful side effects.

*Eliminating False State Sharing:* In object-oriented languages, all the member fields of a class can be accessed by all of its methods. However, the purpose of member fields is to define the state, whose purpose remains constant throughout the lifetime of the class’s objects. A common design flaw is to use the same member field across multiple class methods in different capacities. That is, rather than declaring the field locally in each method, several methods

use the same member field, a condition that we call *false state sharing*. False state sharing is a more serious design flaw that it may seem on the surface, as the problems it causes are similar to that caused by global variables in procedural languages. With respect to cloud offloading, false state sharing complicates program analysis and state synchronization, thus limiting the amount of functionality that can be effectively offloaded. Therefore, we recommend that as part of perfective maintenance for energy optimization, programmers eliminate false state sharing through refactoring as much as possible.

## VI. RELATED WORK

Extending the battery life by reducing the energy consumed by mobile applications has been the focus of multiple complimentary research efforts such as more energy-efficient operating systems (e.g., energy-efficient CPU scheduling [26], disk power managements [24], and process migration [1], network protocols [25]), VM-level [17] and application-level offloading techniques [4].

Although the majority of these efforts has focused on one particular system layer (i.e., mainly the network), a technique called a cross-layer approach effectively controls energy consumption by leveraging the information provided by multiple system layers [11]. Several programming abstractions enable effective adaptations that leverage multiple optimization strategies. The Odyssey platform [5] adapts data or computational quality to save energy consumption, so as not to exceed the available system resources. These

energy-aware adaptations can identify possible trade-offs between energy consumption and application quality, choosing an energy management strategy based on the runtime conditions. DYNAMO [11] is another middleware platform that adapts energy optimization strategies across various system layers, including applications, middleware, OS, network, and hardware, to optimize both performance and energy.

While much research has focused on system-level solutions, programming-level approaches (e.g., algorithms [12], design patterns [10], software models [19]) have received little attention in the literature. A recent language-based approach to energy-aware programming is ET [3], a new object-oriented programming language that enables the programmer to write energy-aware code.

By contrast, the AMCO focus is on perfective maintenance, even though it adopts its techniques from both programming- and system-level approaches to energy optimization. The novelty of AMCO lies in combining analysis-driven program transformation and runtime adaptation.

## VII. CONCLUSIONS

We have presented a novel perfective maintenance approach to reducing the energy consumption of mobile applications, adaptive, multi-target cloud offloading (AMCO). Our approach reduces the energy consumed by mobile applications without changing their source code by employing powerful program analysis and transformations as well as an adaptive runtime system that determines an optimal offloading strategy at runtime. We have evaluated our approach by reducing the energy consumed by micro benchmarks and third-party applications under different execution environments. These results indicate that our approach represents a promising direction in developing pragmatic and systematic tools for the perfective maintenance of mobile applications.

## ACKNOWLEDGMENTS

This research is supported by the National Science Foundation through the Grant CCF-1116565.

## REFERENCES

- [1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *6<sup>th</sup> ACM European Conference on Computer Systems*, 2011.
- [2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2<sup>nd</sup> conference on Symposium on Networked Systems Design & Implementation*, 2005.
- [3] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2012.
- [4] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *8<sup>th</sup> International Conference on Mobile Systems, Applications, and Services*, 2010.
- [5] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.
- [6] Gartner, Inc. Predicts 2013: Mobility becomes a broad-based ingredient for change, Nov. 2012.
- [7] B. Haumacher, T. Moschny, and M. Philippsen. The JavaParty project. [www.ipd.uka.de/JavaParty](http://www.ipd.uka.de/JavaParty), 2007.
- [8] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *32<sup>nd</sup> International Conference on Distributed Computing Systems*, 2012.
- [9] Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, vol. 55, no. 9, 2013.
- [10] Y. Liu. Energy-efficient synchronization through program patterns. In *1<sup>st</sup> International Workshop on Green and Sustainable Software*, 2012.
- [11] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *IEEE Journal on Selected Areas in Communications*, 25(4):722–737, 2007.
- [12] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenIT. In *1<sup>st</sup> International Workshop on Green and Sustainable Software*, Zurich, Suisse, 2012.
- [13] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *7<sup>th</sup> ACM European Conference on Computer Systems*, 2012.
- [14] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.
- [15] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *9<sup>th</sup> International Conference on Mobile Systems, Applications, and Services*, 2011.
- [16] A. Rountev. Precise identification of side-effect-free methods in Java. In *20<sup>th</sup> IEEE International Conference on Software Maintenance*, 2004.
- [17] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.
- [18] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.
- [19] C. Thompson, H. Turner, J. White, and D. Schmidt. Analyzing mobile application software power consumption via model-driven engineering. In *1<sup>st</sup> International Conference on Pervasive and Embedded Computing and Communication Systems*, 2011.
- [20] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *5th International Conference on Generative programming and Component Engineering*, pages 89–94, 2006.
- [21] E. Tilevich and Y. Smaragdakis. NRMI: Natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):174–187, 2008.
- [22] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1–40, 2009.
- [23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*, 1999.
- [24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. *ACM SIGOPS Operating Systems Review*, 36(SI):117–129, 2002.
- [25] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *21<sup>st</sup> IEEE Computer and Communications*, volume 3. IEEE, 2002.
- [26] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37(5):149–163, 2003.

# Efficient Automated Program Repair through Fault-Recorded Testing Prioritization

Yuhua Qi, Xiaoguang Mao\* and Yan Lei  
 School of Computer  
 National University of Defense Technology  
 Changsha, China  
 {yuhua.qi, xgmao, yanlei}@nudt.edu.cn

**Abstract**—Most techniques for automated program repair use test cases to validate the effectiveness of the produced patches. The validation process can be time-consuming especially when the object programs ship with either lots of test cases or some long-running test cases. To alleviate the cost for testing, we first introduce regression test prioritization insight into the area of automated program repair, and present a novel prioritization technique called FRTP with the goal of reducing the number of test case executions in the repair process. Unlike most existing prioritization techniques frequently requiring additional cost for gathering previous test executions information, FRTP iteratively extracts that information just from the repair process, and thus incurs trivial performance loss. We also built a tool called *TrpAutoRepair*, which implements our FRTP technique and has the ability of automatically repairing C programs. To evaluate *TrpAutoRepair*, we compared it with *GenProg*, a state-of-the-art tool for automated C program repair. The experiment on the 5 subject programs with 16 real-life bugs provides evidence that *TrpAutoRepair* performs at least as good as *GenProg* in term of success rate, in most cases (15/16); *TrpAutoRepair* can significantly improve the repair efficiency by reducing efficiently the test case executions when searching a valid patch in the repair process.

**Keywords**—automated program repair; test case prioritization; efficiency; automated debugging

## I. INTRODUCTION

Automated program repair is considered to be more difficult than bug finding [1], and plays an important part on providing complete automated debugging [2]. Most publications in the field focus on the problem of how to generate candidate patches for validation and deployment [3], [4], [5], [6]. When one candidate patch is produced, test cases, which serve as regression testing, are executed to confirm that the patch is valid in the sense that the patch does fix the bug and does not introduce unexpected side effects. Unfortunately, chances are that regression testing includes either a large number of test cases or some long-running test cases. For example, as reported by Rothermel et al. in [7], running one of their products against the entire test cases requires seven weeks. What is worse, automated program repair to its essence is an exercise in trial and error, meaning that there may be lots of invalid patches to be identified through testing before a valid patch is found, resulting in a significant amount of time cost is spent on testing [8, Fig. 8]. To

address this issue, in this paper we plan to optimize the patch validation process by maximizing early invalid-patch detection through prioritizing test cases, and thus speed up the whole repair process.

In the area of regression testing, test case prioritization techniques reorder test cases in order to maximize the *rate of fault detection*, a measure used to evaluate that how quickly faults are detected in the testing process [9]. And the ultimate goal of prioritization is to get such ordered test cases that shall reproduce the failures as soon as possible. In the context of automated program repair, test case prioritization can be redefined in such a way of scheduling test cases in an order that maximizes the *rate of invalid-patch detection*, a measure of how quickly invalid patches are detected in the validation testing process; higher rate means less time cost on the candidate patches validation and thus results in more efficient automated program repair. In the repair process, a candidate patch is considered to be invalid if some test case detects fault for the patched program. In this sense, test case prioritization for automated program repair is exactly equivalent to that for regression testing; and thus we can adopt existing test case prioritization techniques to improve the rate of invalid-patch detection.

Most existing test case prioritization techniques, however, require some previous test execution information, such as code coverage and estimated ability of exposing faults for each test case, to reorder test cases for subsequent runs; the process of gathering that information data may be either difficult or time-consuming [7], [9] especially for large-scale programs. For example, for large programs, techniques based on code coverage, which perform instrumentation to gather the total number of statements covered by each test case, can be too expensive and even infeasible because too much trace data (sometimes over Gb) need to be traced; techniques based on the ability of exposing faults most frequently approximate the fault-exposing-potential (FEP) of each test case by mutation analysis [7], [10], the cost of which could be very high even for some small programs. Hence, if we directly take advantage of existing regression test prioritization techniques, additional cost of gathering relevant information used to prioritize test cases is unavoidable before the repair progress, and can be computationally expensive especially for large programs. Note that although

\*Corresponding author.

some techniques presented in existing papers [11] can prioritize test cases without coverage information, they use some other static analysis, which may be time-consuming for complex programs, to guide the prioritization process.

In this paper, we first introduce regression test prioritization insight into the area of automated program repair, and present *fault-recorded test prioritization (FRTP)*, a novel technique that can effectively improve the rate of invalid-patch detection. Unlike most existing test case prioritization techniques requiring additional cost for gathering previous test executions information before the prioritization, *FRTP requires no prioritization information before the repair process, but rather iteratively extracts that information from the repair process*, which incurs trivial overhead and thus is very suitable in the context of automated program repair.

For maximum applicability of FRTP, we built a tool called *TrpAutoRepair*, which implements our FRTP technique and has the ability of automatically repairing faulty C programs. We then compared *TrpAutoRepair* with *GenProg*, a state-of-the-art tool on automated program repair, by running them to repair a set of large-scale faulty programs including regression test cases with the size ranging from 53 to 4,986. Experimental results indicate that *TrpAutoRepair* clearly requires fewer test case executions, and thus has higher repair efficiency.

In summary, this paper makes the following main contributions:

- The first attempt to introduce regression test prioritization insight into the area of automated program repair, and a description of new prioritization technique called FRTP with the goal of reducing the number of test cases execution in the repair process (Section III). FRTP requires no previous test execution information before the repair process, but rather iteratively extracts that information from the repair process itself.
- Experimental evaluation of *TrpAutoRepair*, which implements FRTP technique (Section IV), in comparison with *GenProg*, a state-of-the-art tool on automated program repair, with the same experimental context (Section V). The experimental results shows that *TrpAutoRepair* outperforms *GenProg* on both efficiency and effectiveness.

## II. BACKGROUND

This section first describes some core information on automated program repair, and then presents the definition of test case prioritization.

### A. Automated Program Repair

We limit the scope of automated program repair to automatically fixing faults in *source code*, not binary files or run-time patching. Although there is plenty of research on automated fault localization, removing most faults is still the task for developers. Hence, if we seek a full automation of

software engineering, automating the task of fixing faults is necessary.

Generally, automated program repair can be divided into three procedures: fault localization, patch generation and patch validation. When some failure occurs, suspicious faulty code snippet causing the failure should be identified by applying existing fault localization techniques such as Tarantula [12]. When the faulty code snippet (most often in terms of statements list ranked by suspiciousness) is located, the repair tools can produce many candidate patches by modifying that code snippet, according to specified repair rules based on either evolutionary computation [3], [8], [6], [13] or code-based contracts [14], [15]. To check that whether one candidate patch is valid or not, test suites are often used to assess patched program correctness.

For test suites, running the patched program against a set of test cases, which consists of negative test cases (revealing the presence of the fault) and positive test cases (characterizing the normal behaviors of object program), is used to check whether the patch is valid; a candidate patch is considered valid iff the patched program passes all the test cases. The time cost for patch validation scales with the time to run these test cases. Unfortunately, complex or critical programs most often equip with large, long-running test suites. Taking the `php` case in [3] for example, there are over 8,000 test cases; completely executing these test cases can take several minutes. To speed up the process of patch validation, one feasible solution is scheduling the test cases in an order that attempts to maximize early fault detection (if a candidate patch is really invalid), which improves the rate of invalid-patch detection; and this is also the problem that we plan to investigate in this paper.

### B. Test Case Prioritization

As described in [7] by Rothermel et al., the test case prioritization problem can be defined as follows:

*Given:*  $T$ , a test suite;  $PT$ , the set of permutations of  $T$ ;  $f$ , a function from  $PT$  to the real numbers.

*Problem:* Find  $T' \in PT$  such that  $(\forall T'') (T'' \in PT) (T'' \neq T') [f(T') \geq f(T'')]$ .

Here,  $PT$  represents the set of all possible permutations (in terms of orderings) of  $T$ , a specified test suite;  $f$  is a transition function which can evaluate an award value for any ordering of  $PT$ .

There are many goals of test case prioritization [7], such as increasing the rate of fault detection of a test suite and increasing the coverage of coverable code in the system under test at a faster rate.  $f$  describes the goal quantitatively and thus represents different quantifications for these goals. In this paper, we focus on the goal of increasing the rate of fault detection of a test suite, that is, we plan to reorder test cases of test suite to increase the likelihood of revealing faults earlier in the testing process.

In addition, two different types of test case prioritization are described in [7]: general and version-specific. For general test case prioritization, the goal is to find such a test case order  $T'$  that can be useful over a sequence of subsequent modified versions of  $P$ ; in contrast, for version-specific test case prioritization, the goal is to find such a test case order  $T'$  that is very effective for one specified modified version of  $P$ . The main difference between the two types is that the prior one can be effective on average over a succession of subsequent releases but may be less effective on one specific version; the latter one, however, has the opposite effect.

Obviously, version-specific test case prioritization, in essence, is analogous to the process of patch validation for automated program repair described in Section II-A, which validates one candidate patch by running corresponding patched program against specified test cases. For both of them, specified test cases are all tested on a specific version. However, for automated program repair, directly adopting the techniques on version-specific test case prioritization is not suitable, because version-specific test case prioritization often suffers from the problem of excessively delaying the very regression testing activities due to the cost of prioritizing. The prioritization cost can be very expensive for large-scale programs, and thus reduce the efficiency.

### C. Prioritization Information and Limitations

As described in Section II-B, prioritization information gathered from the previous test case executions is often required for most existing prioritization techniques. According to the description in [7], [16], although there are several types of prioritization information, we focus on two types which are highly correlated with our work in this paper:

*Code coverage.* The intuition for the idea is that early maximization of code coverage are more likely to increase the chance of early maximization of fault detection. Prioritization techniques based on code coverage prioritize test cases in terms of their total coverage of code components<sup>1</sup> by counting the number of components covered by each test case. If multiple test cases have the same number of code coverage, these test cases are reordered randomly. In addition, for version-specific prioritization, test cases which can cover the modified code area are considered to be more likely to reveal faults, and thus should be given more chances for early execution.

*FEP.* The intuition for the idea is that the ability of a test case to expose a fault depends not only on whether the test case achieves the coverage of the faulty area, but also on the probability that a fault in that area will cause a failure for that test case [17]. That is, code coverage is necessary but not sufficient to be used as the prioritization criterion. Mutation analysis [18], in general, is used to measure the fault-exposing-potential (FEP) of a test case. Program mutation

<sup>1</sup>The code components can be referred as to statements, functions or branches et al.

produces many different mutant versions of the program by simple syntactic changes to the program source. For each test case the award value in terms of mutation score is determined to be the ratio of mutants exposed by this test case to the total mutants [16]. Then, FEP prioritization sorts the test cases in descending order of that award value.

However, for existing prioritization techniques based on either code coverage or FEP, gathering prioritization information may be time-consuming. For example, mutation analysis used to evaluate FEP, traditionally, has been seen to be computationally expensive, though there are many techniques that try to reduce the cost [18].

In this paper, we try to address the above issue using a novel technique called fault-recorded testing prioritization (FRTP).

## III. THE FRTP TECHNIQUE

In this section we discuss our FRTP technique in detail. We first overview the interesting issue and our insight for suppressing it in Section III-A. Then, we discuss the insight and motivation of FRTP technique in Section III-B. Finally, we describe the algorithm details on the FRTP technique in Section III-C.

### A. Overview

Recently, much research focuses on the area of automated program repair, which attempts to repair a bug-aware program by modifying the program at the *source code level* with the assumption that source code is accessible. Since there is no a priori guarantee that some modification (in terms of candidate patch) actually repairs the defective program, the repair process is not deterministic, which means that lots of candidate patches are most probably produced before a really valid patch is found. As described in [8, Fig. 8], overwhelming majority of time cost is spent on patch validation including test case executions cost and compilation cost.

Unfortunately, in the process of patch validation, test case executions can be time-consuming in most cases but become more serious on complex or critical programs most of which equip with large, long-running test suites. To address the issue and speed up the repair process, we present the FRTP technique, which introduces the test case prioritization insight into the patch validation process. However, for most existing prioritization techniques, previous test execution information is often necessary for prioritizing test cases. If we directly adopt these techniques, the cost of gathering prioritization information may trade off the benefits generated by the very test case prioritization itself. For FRTP the prioritization information is not specially gathered before the repair process; rather, that information is incrementally gotten from the repair process. Hence, FRTP requires no previous test executions information before the repair process and incurs the trivial cost of gathering prioritization

information. A more detail account on FRTP is presented in the following subsections.

### B. Prioritizing Test Cases for Automated Program Repair

Having investigated the repair process, we find that the prioritization information, such as code coverage and FEP, can be extracted just from the repair process with trivial cost.

Suppose that the source code of a program  $P$  is constructed from a set of components  $C = \{c_1, c_2, \dots, c_i, \dots, c_n\}$ ; when a fault is detected, the defective component set of  $C_{sub} \subset C$  causing the fault can be identified through existing fault localization techniques. Then,  $C_{sub}$  is modified to repair the program, producing lots of candidate patches  $PT = \{pt_1, pt_2, \dots, pt_i, \dots\}$ . For each  $pt \in PT$ , run the program  $P'$ , a mutant version of  $P$  updated by  $pt$ , against a specified set of test cases  $T$ , which is used to guarantee that  $pt$  does fix the fault and does not introduce new faults. If a test case  $t \in T$  detects fault, then  $pt$  is killed by the test case and the next candidate patch will be considered; if all test cases successfully pass the testing, then the repair process is terminated and outputs  $pt$  as the valid patch for the fault.

For the above description of automated program repair, given the  $i$ th candidate patch  $pt_i \in PT$  (suppose that a valid patch has not been found before this patch): prior to  $pt_i$ , there are total  $i-1$  candidate patches having been validated; and these patches are considered invalid in the sense that some test case has detected some fault for each patch in the process of patch validation (i.e., the patch is killed by the test case). We record these test cases which have ever detected faults prior to validating  $pt_i$ , producing the set of test cases  $T_i \subseteq T$  corresponding to  $pt_i$ . Obviously,  $T_i$  meets the constraint of

$$Size(T_i) \leq Size(PT_i), PT_i = \{pt_1, pt_2, \dots, pt_{i-1}\}.$$

The  $Size$  function gives the number of one specific set. The constraint indicates that one test case  $t \in T_i$  may kill more than one candidate patches, and one candidate patch  $pt \in PT_i$  is killed by one and only one test case  $t \in T_i$ . The mapping relation between  $T_i$  and  $PT_i$  is shown in Figure 1.

In Figure 1, since each  $t \in T_i$  has ever killed some  $pt \in PT_i$  by detecting fault,  $t$  is highly correlated with the defective  $C_{sub}$  components (located by fault localization techniques) in the sense that  $t$  most probably covers the code of  $C_{sub}$  or at least covers some code highly depending on  $C_{sub}$ . Thus, compared to  $T'_i = T \setminus T_i$ , the relative complement set of  $T$  and  $T_i$ ,  $T_i$  should have higher prioritization for early execution in the patch validation process of  $pt_i$ , which is consistent with version-specific prioritization techniques based on *code coverage* (in fact, for  $pt_i$ , the program  $P'$  updated by  $pt_i$  can be regarded as one specific version of  $P$ ).

On the other hand, prior patch validation process for each  $pt \in PT_i$  to its essence can be considered as the mutation

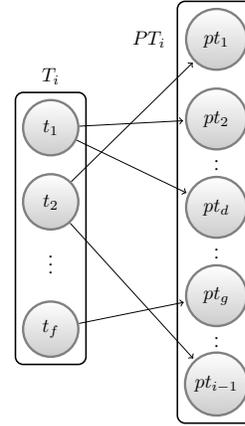


Figure 1. Mapping relation between  $T_i$  and  $PT_i$ : the directed line indicates that one test case has ever killed the corresponding candidate patch before the patch validation for  $pt_i$ .

analysis before the validation for  $pt_i$ , because the program  $P'$  updated by  $pt \in PT_i$  is actually one mutant version of  $P$  by modifying the program source code; the larger the value of  $i$  is, the more accuracy the results of mutation analysis are. Specifically in Figure 1, compared to  $t_f$  both  $t_1$  and  $t_2$  have higher FEP in terms of killing more candidate patches, and thus should be earlier executed to reduce the time cost for exposing the fault if  $pt_i$  is actually an invalid patch, which is consistent with prioritization techniques based on FEP. In the following subsection, we describe the detailed algorithm for FRTP on how to take advantage of these prioritization information to prioritize test cases.

### C. Algorithm for FRTP

Algorithm 1 shows our FRTP technique in detail. By iteratively recording the faults in the process of patch validation (hence, in term of *fault-recorded*), FRTP attempts to improve the rate of invalid-patch detection through prioritizing test cases.

Before starting to repair the defective program  $P$ , we first reconstruct the test cases  $T$  for the purpose of giving  $T$  the ability of mapping each test case  $t \in T$  to the corresponding number of candidate patches killed by  $t$ . Specifically, for each  $t \in T$ , FRTP constructs a tuple  $(t, index)$  where  $t$  is the original test case and  $index$  is a non-negative integer recording the current number of candidate patches killed by  $t$ ; in fact,  $index$  indicates the FEP ability of  $t$  and is also the main prioritization information. Then, FRTP reconstructs  $T$  from all these tuples and initializes the  $index$  value for each tuple (lines 1-3). Recall that test cases  $T$  consists of two parts — negative test cases (revealing the presence of the fault and represented by  $n_0$ ) and positive test cases (characterizing the normal behaviors of object program and represented by  $\{t_1, t_2, \dots, t_n\}$ ). Due to the natural ability of fault detection even before the repair process, on line 3 FRTP initializes the  $index$  value of  $n_0$  with 1 rather than 0 for other test cases.

To limit the search space for generating patches, on line 4 the function *FaultLocalization* preliminarily locates the

---

**Algorithm 1:** The FRTP Algorithm

---

**Input** : Defective program  $P$  and test cases  $T$

**Output:** One valid patch  $pt$

```
1 index  $\leftarrow$  0; // Initialize the index value
2  $\{n_0, t_1, t_2, \dots, t_n\} \leftarrow T$ ;
3  $T \leftarrow \{(n_0, 1)(t_1, \text{index}), (t_2, \text{index}), \dots, (t_n, \text{index})\}$ ;
4  $C_{sub} \leftarrow \text{FaultLocalization}(P, T)$ ;
5  $\text{SuccessFlag} \leftarrow \text{false}$ ;
6 repeat
7    $pt \leftarrow \text{PatchGeneration}(C_{sub})$ ;
8    $P' \leftarrow \text{Update}(P, pt)$ ;
9   for  $i \leftarrow 0$  to  $n$  do
10    //Check that whether  $pt$  is valid;
11     $(t_{index}, \text{index}) \leftarrow \text{GetTestcase}(T, i)$ ;
12    if  $\text{PatchValidation}(P', t_{index}) \neq \text{true}$ 
13      then
14         $\text{temp} \leftarrow (t_{index}, \text{index} + 1)$ ;
15         $T \leftarrow \text{Prioritize}(T, \text{temp})$ ;
16        break;
17      else if  $i = n + 1$  then
18         $\text{SuccessFlag} \leftarrow \text{true}$ ;
19      else
20        continue;
21    end
22 until  $\text{SuccessFlag} = \text{true}$ ;
23 return  $pt$ ;
```

---

defective components set  $C_{sub}$  by the analysis to trace data sets of running the program  $P$  against test cases  $T$  (a detailed account can be found in [12]). In addition, for simplicity FRTP creates the state variable  $\text{SuccessFlag}$  on line 5, which records current repair state of whether a valid patch being found.

The search begins by generating one patch  $pt$  and getting a mutant executable program  $P'$  updated by  $pt$ . The function  $\text{PatchGeneration}$  can produce one concrete patch  $pt$  by modifying  $C_{sub}$  in light of specific modification rules [3], [6], [14] (line 7 in Algorithm 1). Note that the patches produced by multiple calls to  $\text{PatchGeneration}$  are different due to the application of randomized algorithm (more details can be found in Section IV-B). Once  $pt$  is generated, the function  $\text{Update}$  is called to recompile the patched program to a executable program  $P'$  (line 8). Lines 9-21 run  $P'$  against  $T$  in order, and reorder test cases  $T$  according to the running result. Given that the  $i$ th test case for  $T$  (note that it is most probably that  $T$  has been prioritized due to prior invalid patches), line 11 calls the function  $\text{GetTestcase}$  to get the  $i$ th tuple  $(t_{index}, \text{index})$  of  $T$ ; then, the function  $\text{PatchValidation}$  runs  $P'$  against  $t_{index}$  (line 12); if a fault is detected by  $t_{index}$ , then FRTP records the fault by updating the tuple  $(t_{index}, \text{index})$  with  $(t_{index}, \text{index} + 1)$  for  $T$  (line 13), and reorders the execution order of each test case  $t \in T$

by calling the function  $\text{Prioritize}$  (line 14).  $\text{Prioritize}$  sorts the test cases (in fact, the tuples for  $T$ ) in descending order of the value of  $\text{index}$ . (When multiple test cases have the same value of  $\text{index}$ , FRTP orders them randomly.) If  $P'$  successfully passes all the test cases  $T$  (line 16), FRTP considers that a valid patch is found (line 17), and terminates the repair process immediately with the valid patch outputted (line 23).

#### IV. IMPLEMENTATION

In this section, we first describe  $\text{GenProg}$  [3], [8], a state-of-the-art tool for automated C program repair, on which we built  $\text{TrpAutoRepair}$ . Then, we present the implementation details on  $\text{TrpAutoRepair}$ .

##### A. $\text{GenProg}$

$\text{GenProg}^2$  has the ability of fixing bugs in deployed, legacy C programs without formal specifications. With the hypothesis that some missed important functionalities may occur in another position in the same program,  $\text{GenProg}$  attempts to automatically repair defective program with evolutionary computation [19].  $\text{GenProg}$  takes advantage of a modified version of genetic programming to maintain a population of variants, which are represented by the corresponding pairs of abstract syntax tree (AST) and weighted path (the results of fault localization). In the process of patch generation, a candidate patch is generated using one of generic algorithm operations: mutation and crossover; the variant is just the AST representing the program updated by the patch. Then, in the process of patch validation, testing is required:  $\text{fitness}$  is evaluated by executing test cases. If the patched program passes all the test cases,  $\text{GenProg}$  terminates the repair process and declares that a valid patch is found, otherwise  $\text{GenProg}$  goes back to the process of patch generation, and generates another patch according to the prior  $\text{fitness}$  values.

Although  $\text{GenProg}$  has successfully fixed many bugs existing among some off-the-shelf programs (e.g., python, libtiff),  $\text{GenProg}$  may suffer from the problem of time-consuming test case executions [3], which becomes more serious on complex or critical programs for which most often equip with large, long-running test suites. What is worse, for the reason that  $\text{GenProg}$  takes the fitness values, which are evaluated by running a fixed number of test cases (10% of all test cases in [3] and [20]), as the metric which serves as the guiding force behind the search for optimal or near optimal patches in the process of patch generation, thus lots of test case executions are necessary even if an invalid patch has been aware (i.e, some test case has ever detected a fault for the patched program).

<sup>2</sup>Available: <http://dijkstra.cs.virginia.edu/genprog/>

Table I  
SUBJECT PROGRAMS

Program	LOC	Test Cases	Version
libtiff	77,000	78	bug-01209c9-aaf9eb3
			bug-0860361d-1ba75257
			bug-0fb6cf7-b4158fa
			bug-10a4985-5362170
			bug-4a24508-cc79c2b
			bug-5b02179-3dfb33b
			bug-6f9fd7-73757f3
			bug-829d8c4-036d7bb
			bug-8f6338a-4c5a9ec
			bug-90d136e4-4c66680f
gmp	145,000	146	bug-d39db2b-4cd598c
			bug-ee2ce5b7-b5691a5a
python	407,000	303	bug-14166-14167
php	1,046,000	4,986	bug-69783-69784
wireshark	2,814,000	53	bug-309892-309910
			bug-37112-37111

### B. *TrpAutoRepair*

We implemented *TrpAutoRepair* on *GenProg* in OCaml language by applying our FRTP techniques.

Specifically, the suspiciousness value  $sp$  of being faulty for each statement  $s \in P$  is computed through a simple fault localization technique same to *GenProg*: a statement never visited by any test case has the  $sp$  value of 0; a statement visited only by negative test case which reproduces the failure has the high value of 1.0; a statement visited both by positive and negative test cases is given the moderate value of 0.1. For each statement  $s$ , the corresponding probability  $pb$  of  $s$  being selected is computed through the formula:  $pb = sp * mute$ , where  $mute$  represents the global mute probability set for initialization. When generating a candidate patch, *TrpAutoRepair* randomly generated each probability value  $p \in [0, 1]$  for each statement  $s$ ; *TrpAutoRepair* selects  $s_i$  for modification iff  $pb_i \geq p_i$ ; the repair rules used by *TrpAutoRepair* to modify these selected statements are also the same as that used by *GenProg*: copying a few lines of code from other parts of the program or modifying existing code (see [8]).

In fact, *TrpAutoRepair* produces candidate patches in the same way of randomly generating patches in the first generation of *GenProg* but without fitness guidance and crossover in the subsequent generations, i.e., muting the code according to both the suspiciousness and global mutation rate specified before the repair process. For the process of patch validation, we validate candidate patches in the way described in Algorithm 1.

## V. EXPERIMENTAL EVALUATION

### A. Research Questions

Our evaluation addresses the following Research Questions. In RQ1, we investigate the effectiveness of *TrpAu-*

*toRepair*. RQ2, in fact, investigates the efficiency (in term of the rate of invalid patch detection) of *TrpAutoRepair*.

- **RQ1:** How does *TrpAutoRepair* compare with *GenProg* in term of repair effectiveness?
- **RQ2:** Can *TrpAutoRepair* improve the rate of invalid-patch detection over *GenProg*? If so, how much better can *TrpAutoRepair* perform than *GenProg* on the improvement?

### B. Subject Programs

In this experiment, we selected the subject programs used in the most recent work [3] on *GenProg* as the experimental benchmarks<sup>3</sup>, all of which are written in C language and are different real-world systems with real-life bugs from different domains. For the `fbcc` program and 5 versions of `libtiff` programs, we have the compilation trouble when we try to compile these programs. We also exclude `gzip` and `lighttpd` programs, because there is too few positive test cases (no more than 10 ones) actually used in both `gzip` and `lighttpd` programs, although more test cases were listed in [3]. Too few test cases makes no sense on the optimization of testing.

For `php` which equips with over 4,000 test cases, several minutes are needed for validating only one patched program, which will result in time-consuming repair process. Thus, complete experimental evaluation on all the `php` program versions can take too much time (see [3, Table II]); the authors in [3] used Amazon’s EC2 cloud computing infrastructure including 10 trials in parallel for their experiment. Given the expensive testing computation, we randomly selected one faulty `php` version without any bias, although these `php` versions shipping with lots of test cases will give *TrpAutoRepair* more advantages. For `gmp`, `python` and `wireshark`, there exists only one version having been repaired successfully by *GenProg* in [3] for each program.

In total, Table I describes our subject programs in detail. The LOC column lists the scale of each subject program in term of lines of code, and the last two columns give the size of positive test cases and the version information. Note that for the `libtiff` program, although there are total 78 test cases listed in Table I, the concrete number of test cases used for each bug version is similar but different because not all the 78 test cases work well for every version. In addition, we assigned one negative test case for each subject program to reproduce the bug.

### C. Experimental Setup

For the purposes of comparison, we separately ran *TrpAutoRepair* and *GenProg* to repair the 5 subject programs with 16 versions described in Table I. All the experimental parameters for *GenProg* in our experiment are the same as

<sup>3</sup><https://church.cs.virginia.edu/genprog/archive/genprog-105-bugs-tarballs/>

Table II  
EXPERIMENTAL RESULTS BY *TrpAutoRepair* AND *GenProg*

Program	Approach	Mean of NTCE	Median of NTCE	Avg. Time(s) Per Repair	Success Rate	A-statistic on NTCE	p-value
libtiff-bug-01209c9-aaf9eb3	TrpAutoRepair	79	78	18.744	100%	0.940400	0.000000
	GenProg	122	109	25.764	100%		
libtiff-bug-0860361d-1ba75257	TrpAutoRepair	63	57	269.149	100%	0.787165	0.000000
	GenProg	177	119	439.535	97%		
libtiff-bug-0fb6cf7-b4158fa	TrpAutoRepair	187	170	242.009	100%	0.813636	0.000000
	GenProg	587	505	465.513	77%		
libtiff-bug-10a4985-5362170	TrpAutoRepair	51	47	42.582	100%	0.834946	0.000000
	GenProg	150	89	73.669	93%		
libtiff-bug-4a24508-cc79c2b	TrpAutoRepair	66	64	55.937	100%	0.926400	0.000000
	GenProg	115	94	88.147	100%		
libtiff-bug-5b02179-3dfb33b	TrpAutoRepair	119	103	121.409	100%	0.826053	0.000000
	GenProg	338	218	241.101	95%		
libtiff-bug-6f9f4d7-73757f3	TrpAutoRepair	68	65	66.104	100%	0.922800	0.000000
	GenProg	123	101	114.560	100%		
libtiff-bug-829d8c4-036d7bb	TrpAutoRepair	154	148	233.548	88%	0.830168	0.000000
	GenProg	540	554	539.218	73%		
libtiff-bug-8f6338a-4c5a9ec	TrpAutoRepair	47	43	25.950	100%	0.867400	0.000000
	GenProg	92	75	33.386	100%		
libtiff-bug-90d136e4-4c66680f	TrpAutoRepair	100	93	82.677	100%	0.936600	0.000000
	GenProg	349	225	132.086	100%		
libtiff-bug-d39db2b-4cd598c	TrpAutoRepair	110	99	46.467	98%	0.883716	0.000000
	GenProg	387	277	96.754	96%		
libtiff-bug-ee2ce5b7-b5691a5a	TrpAutoRepair	104	94	99.444	100%	0.926650	0.000000
	GenProg	354	238	131.874	100%		
gmp-bug-14166-14167	TrpAutoRepair	312	301	606.511	58%	0.817529	0.000590
	GenProg	663	530	472.828	12%		
python-bug-69783-69784	TrpAutoRepair	434	408	452.185	37%	0.990991	0.000000
	GenProg	2572	2318	1409.825	21%		
php-bug-309892-309910	TrpAutoRepair	4997	4994	457.486	100%	1.000000	0.000000
	GenProg	20837	16986	1440.835	97%		
wireshark-bug-37112-37111	TrpAutoRepair	167	179	2159.907	8%	0.040625	0.000204
	GenProg	630	536	1845.472	20%		

\* We separately ran *TrpAutoRepair* and *GenProg* 100 times on each of the 16 subject programs and only recorded the trials leading to a successful repair. Hence, the success rate is  $n\%$  if there are  $n$  successful trials finding successfully a valid patch; the statistics in this table are also computed according to the  $n$  successful trials.

those settings in [3]: we limited the size of the population for each generation to 40, and a maximum of 10 generations for each repair process; the global mute rate *mute* is set to 0.01. In fact, for all generations except the first generation, there are another 40 candidate patches generated due to crossover. That is, for one concrete repair process, *GenProg* can iteratively produce no more than  $40+80*10=840$  candidate patches. For *TrpAutoRepair*, we also limited the size of the population for each generation to 40, and a maximum of 10 generations for each repair process; for each generation, using random search (without crossover) total 40 candidate patches are produced in the same way of the first generation. Hence, for fair comparison, we considered that *TrpAutoRepair* (*GenProg*) failed to repair one subject

program for one repair process if the valid patch was not found within 400 candidate patches.

All the experiments ran on an Ubuntu 10.04 machine with 2.33 GHz Intel quad-core CPU and 4 GB of memory. Since randomized algorithm is applied in both *TrpAutoRepair* and *GenProg*, we statistically analyze the experimental results. Specifically, for *TrpAutoRepair* and *GenProg* we separately performed 100 trials for each program.

#### D. Experimental Results

1) **RQ1:** How does *TrpAutoRepair* compare with *GenProg* in term of repair effectiveness?

In this paper, we evaluate the repair effectiveness according to success rate measuring the difficulty of finding a valid

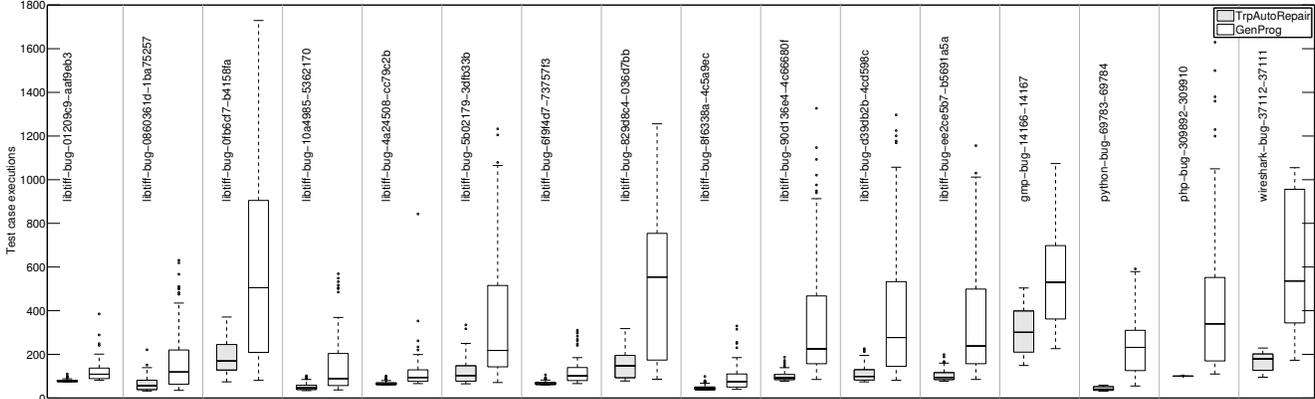


Figure 2. NTCE boxplots for experiments. Note that for the `python` and `php`, which give too large NTCE over other programs, for the ease of presentation we narrowed down the NTCE values with linear scale for the two programs.

patch. Intuitively, the higher repair effectiveness of used repair tool, the stronger ability of the tool finding a valid patch; the stronger ability indicates higher success rate of repairing successfully a faulty program. Note that we separately ran *TrpAutoRepair* and *GenProg* 100 times on each of the 16 faulty programs. Hence, the success rate is  $n\%$  if there are  $n$  successful trials.

For most programs (15 of 16 in Table II), *TrpAutoRepair* has the higher or equal success rate over *GenProg*. The higher success rate means that *TrpAutoRepair* offers the greater ability of finding a valid patch under the same condition. The relatively worse success rate indicates that fitness function evaluation used by *GenProg* does not always work well for guiding the search for optimal or near optimal patches in the patch generation step and even, sometimes (9/16), misleads the search process. We are not surprised for this experimental results. In fact, Andrea Arcuri and Lionel Briand (in their ICSE 2011 paper [21, Page 3]) in particular has presented their concern to the effectiveness of genetic programming used by *GenProg*, and pointed out that genetic programming should be compared against random search to check the effectiveness. However, in most recent work on *GenProg* [3] published in ICSE 2012, they still did not make the compare. In this sense, our experiment is the supplement to [3], and confirms the concern that genetic programming does not always work better than random search, and even worsens the patch search process in our experiment.

The possible reason for the worse repair effectiveness by *GenProg* may be tracked down from the paper [22] written by the same authors of *GenProg*: current fitness functions including that used by *GenProg* are either overly simplistic or likely to exhibit "all-or-nothing" behavior, and thus are not well correlated with true distance between an individual and the global optimum. Since chances are that imprecise fitness functions can mislead the search process, it should not be surprising that in our experimental evaluation *GenProg* has the worse repair effectiveness on many programs when

compared to *TrpAutoRepair* for which random search is used. The only exception is on `wireshark`, for which *GenProg* outperforms *TrpAutoRepair*. Having analyzed the repair process, we find that the patched programs are more likely to fail to be compiled in the initial phase of repair process, compared to other programs. Thus, we suspect the reason for the exception to be that fitness function is good at eliminating the bad patches (which fails to be compiled) and can produce more compilation-able candidate patches in the sequent repair process.

2) **RQ2:** *Can TrpAutoRepair improve the rate of invalid-patch detection over GenProg? If so, how much better can TrpAutoRepair perform than GenProg on the improvement?* To answer this question, we require a measure with which to assess and compare the rate of invalid-patch detection between *TrpAutoRepair* and *GenProg*. To quantify the goal of increasing test cases' rate of fault detection, Rothermel et al. present APFD, a metric measuring the weighted average of the percentage of faults during the test case executions [7]. Although the APFD metric is generally used for evaluating the effectiveness of various test case prioritization techniques [9], APFD is not suitable for evaluating the rate of invalid-patch detection on the area of automated program repair for the reason that the order of test cases in the repair process varies all the time (recall the line 14 in Algorithm 1), while APFD requires a fixed order before the evaluation.

In our experiment, we assess the rate of invalid-patch detection by computing the Number of Test Case Executions (NTCE) within one successful repair process; the smaller the NTCE value is, the higher the rate of invalid-patch detection is. An overview of how the two tools affected the test cases' rate of invalid-patch detection can be determined from Figure 2, which provides boxplots of the NTCE values of each successful trial for running separately *TrpAutoRepair* and *GenProg* to repair each subject program.

Figure 2 shows that, for all the 16 subject programs, *TrpAutoRepair* killed the invalid patches more rapidly (i.e.,

the lower NTCE values of both “Mean” and “Median” in Table II) than *GenProg*, which justifies that our FRTP technique can improve the rate of invalid-patch detection.

As described above, *TrpAutoRepair* improves the rate of invalid-patch detection over *GenProg* in Figure 2, we then assess the magnitude of the improvement by measuring effect size (scientific significance)—in this case, a difference in the ability to detect invalid patches. The nonparametric Vargha-Delaney *A*-test [23], which is recommended in [21] and was also used in [24], is used to evaluate the effect size in this experiment. In [23] Vargha and Delaney suggest that *A*-statistic of greater than 0.64 (or less than 0.36) is indicative of “medium” effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising “large” effect size.

Table II shows the effect size *A*-statistic, which represents the difference significance between *TrpAutoRepair* and *GenProg* on NTCE, and the *p*-value (for the rank-sum test used for computing the *A*-statistic) for each subject program. Because *A*-statistic of all the 16 programs are either greater than 0.71 or less than 0.29 with low *p*-value ( $p < 0.05$ ), it is reasonable to consider that *TrpAutoRepair* significantly improves the rate of invalid-patch detection over *GenProg* in our experiment, which also justifies the advantage of our FRTP technique.

#### E. Threats to Validity

Like many studies, the main threat is related to the possible poor generalization of the experimental results. Since we have selected 16 subject programs, chances are that the results tend to support the conclusions drawn from our experiment with some bias. Although one effective solution to minimize the experimental bias is to increase the number of subject programs, the experiment on more programs means that more expensive computation resource is necessary. To reduce the expensive computation, we plan to further optimize the repair process as we have done for earlier work [25], which will allow us to investigate an large number of subject programs in the future.

Another issue is related to the appropriateness of the evaluation criteria in our experiment. We use the nonparametric *A*-test to evaluate the effect size according to the NTCE values. The reasonableness of the adoption remains to be confirmed in the future work, though NTCE can intuitively approximate the effectiveness of the rate of invalid-patch detection.

## VI. RELATED WORK

**Automated Program Repair.** Many studies try to repair defective programs at the source code level in different ways. Guided by evolutionary computation, *GenProg* seeks to repair programs without any specifications. *AutoFix-E* [4] can generate patches for faulty programs automatically but requires for the contracts. *JAFF* [6] has the ability

of automatically fixing bugs existing in the java programs using an co-evolutionary approach; the repair effectiveness of *JAFF* is still unknown on real-world faulty softwares. In addition, there is some work [5] on fixing automatically php HTML generation Errors via string constraint solving.

Although all the above tools generate the candidate patches according to different rules, there is at least one common point: to validate these patches, they have to test the patched programs through regression testing. And none of these tools scales well to complex or critical programs shipping with either large number of test cases or long-running test cases.

**Test Case Prioritization Techniques.** Techniques for test case prioritization try to improve the rate of fault detection by scheduling test cases for testing, and are intensively studied in regression testing [9]. Based on several different coverage criteria a family of test case prioritization techniques were presented in [7]. There are also lots of work focusing on different granularity levels such as the function level [16], system model [26], the block level and method level [27]. In addition, some research work evaluated traditional techniques on test case prioritization in context of time-aware test case prioritization [28]. More details and work on Test Case Prioritization can be found in the survey [9].

## VII. CONCLUSION AND FUTURE WORK

We present the test case prioritization technique called FRTP that can assist the patch validation process by improving the rate of invalid-patch detection in the context of automated program repair. FRTP can effectively reduce the size of test case executions for exposing the invalid patches and thus improves the whole repair efficiency, especially when the object program ships with either large number of test cases or long-running test cases. Different from most existing test case prioritization techniques requiring additional cost for gathering the prioritization information before the prioritization, FRTP iteratively extracts that information by recording the faults information in the repair process, which suffers from trivial cost. We have built the tool *TrpAutoRepair*, which implements our FRTP technique and can repair programs of full automation.

We evaluated *TrpAutoRepair* against *GenProg*, a state-of-the-art tool for automated C program repair. For repair effectiveness, *TrpAutoRepair*, in most cases (15/16), outperformed *GenProg*; for repair efficiency, *TrpAutoRepair* can improve the rate of invalid-patch detection (in term of requiring much fewer test case executions to determine invalid patches) for all the subject programs in our experiment, and this improvement is statistically significant in term of “large” effect size for *A*-statistic. Complete experimental results in this paper are available at:

<http://sourceforge.net/projects/trpautorepair/files/>

In the future we plan to further speed up the repair process by applying weak recompilation and precise fault localization techniques [25], [29] to *TrpAutoRepair*. We believe that this application will make *TrpAutoRepair* more powerful in the sense that less time is taken to repair defective programs.

#### ACKNOWLEDGMENT

The authors thank W. Weimer *et al.* for their noteworthy work on *GenProg*, based on which we built the *TrpAutoRepair* system. This work was supported by the National Natural Science Foundation of China (Grant Nos. 61120106006, 91118007 and 91318301), National High Technology Research and Development Program of China (Grant Nos. 2011AA010106 and 2012AA011201).

#### REFERENCES

- [1] M. Harman, "Automated patching techniques: the fix is in technical perspective," *Communications of the ACM*, vol. 53, no. 5, pp. 108–108, 2010.
- [2] A. Zeller, "Automated debugging: Are we close," *Computer*, vol. 34, no. 11, pp. 26–31, 2001.
- [3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: fixing 55 out of 105 bugs for \$8 each," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.
- [4] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2010, pp. 61–72.
- [5] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of HTML generation errors in php applications using string constraint solving," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 277–287.
- [6] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494 – 3514, 2011.
- [7] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 10, pp. 929 –948, oct 2001.
- [8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 54 –72, 2012.
- [9] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 4, pp. 67–120, 2012.
- [10] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 8, pp. 608 –624, 2006.
- [11] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing junit test cases in absence of coverage information," in *International Conference on Software Maintenance (ICSM)*, 2009, pp. 19–28.
- [12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *International Conference on Automated Software Engineering (ASE)*, 2005, pp. 273–282.
- [13] A. Arcuri, "On the automation of fixing software bugs," in *International Conference on Software Engineering (ICSE)*, 2008, pp. 1003–1006.
- [14] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *International Conference on Automated Software Engineering (ASE)*, 2011, pp. 392 – 395.
- [15] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *International Conference on Software Engineering (ICSE)*, 2011, pp. 191–200.
- [16] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 159 –182, feb 2002.
- [17] J. Voas, "PIE: a dynamic failure-based technique," *IEEE Transactions on Software Engineering (TSE)*, vol. 18, no. 8, pp. 717 –727, aug 1992.
- [18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649 –678, 2011.
- [19] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.
- [20] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and Evolutionary Computation (GECCO)*, 2012.
- [21] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.
- [22] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Genetic and Evolutionary Computation (GECCO)*, 2010, pp. 965–972.
- [23] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
- [24] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 6, pp. 763–777, Nov. 2010.
- [25] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu, "More efficient automatic repair of large-scale programs using weak recompilation," *Science China Information Sciences*, vol. 55, no. 12, pp. 2785–2799, 2012.
- [26] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *International Conference on Software Maintenance (ICSM)*, 2008, pp. 247 –256.
- [27] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2004, pp. 113 – 124.
- [28] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 213–224.
- [29] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 191–201.

# Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names

Giuseppe Scanniello

*Dipartimento di Matematica, Informatica e Economia  
Università della Basilicata, Italy  
Email: giuseppe.scanniello@unibas.it*

Michele Risi

*Departmento di Management & Information Technology  
Università di Salerno, Italy  
Email: mrisi@unisa.it*

**Abstract**—We carried out a controlled experiment to investigate whether the use of abbreviated identifier names affects the ability of novice software developers to identify and fix faults in source code. The experiment was conducted with 49 students in Computer Science. The results of the statistical analyses indicate that there was not a significant difference to identify and to fix faults, when source code contains either abbreviated and full-word identifier names. In other words, it seems that abbreviated identifiers provide the same information as full-word identifiers on the solution domain and the implementation.

## I. INTRODUCTION

Software maintenance, testing, quality assurance, reuse, and integration are examples of activities in the software engineering field that involve existing source code [6]. To deal with these activities software engineers should have source code and design documentation [28]. The worst case scenario is that source code (program statements and source code comments) is the only available resource. However, even if design documentation is available developers consider it inadequate to comprehend source code (e.g., [21], [26], [27]). Therefore, for developers the only possible strategy is the analysis of the source code and/or its execution.

When reading source code, software engineers have two main source of information: identifier names (or simply identifiers, from here on) and comments [22], [23]. When source code is uncommented, as often happens, comprehension is almost exclusively dependent on the identifier names. Three different kind of identifiers can be used: single letters, full-words, and abbreviations. Single letter identifiers are single chars, while full-word identifiers are composed by full English words concatenated according to a given naming convention. Finally, abbreviation variants are based on the full-word variants. Compounding English words are shortened by applying any method. For example, the word *abbreviation* could be abbreviated as *abbr*, *abbrv* or *abb*. A special kind of abbreviation is the contractions. A contraction of a word is made by omitting certain letters and bringing together the first and last letters (e.g., *dr* is the contraction of the word doctor, while *prt* of print).

In this paper, we present the results of a controlled experiment aimed to investigate whether the presence of abbreviated identifiers in source code affects the ability of

novice software developers to identify and fix faults. The participants were asked to identify and fix faults in four applications implemented in C. Comments were omitted from the code the subjects viewed. Depending on the application a number of bug reports were given to the participants. The experiment was conducted in a research laboratory at the University of Basilicata with 49 Bachelor students in Computer Science.

The remainder of the paper is organized as follows. In Section II, we present the design of the controlled experiment, while the achieved results are presented and discussed in Section III and in Section IV. We highlight the threats that could affect the validity of our study in Section V. Related work is presented in Section VI, while final remarks conclude the paper.

## II. CONTROLLED EXPERIMENT

We carried out an ABBA-type experiment [34]. The experiment (denoted E-UBAS) was carried out at the University of Basilicata in January 2012 with 49 students from the Bachelor's program in Computer Science.

The experiment was carried out by following the recommendations provided by Juristo and Moreno [14], Kitchenham *et al.* [19], and Wohlin *et al.* [34] and reported according to the guidelines suggested by Jedlitschka *et al.* [13]. For replication purposes, the experiment material is available at [www2.unibas.it/gscanniello/Identifiers/](http://www2.unibas.it/gscanniello/Identifiers/).

### A. Goal

Applying the Goal Question Metric (GQM) template [4], the goal of the experiments can be defined as:

- **Analyse** abbreviated and full-word identifier names *for the purpose* of evaluating their use *with respect to* fault detection and fixing *from the point of view of* the developer *in the context of* C programs and novice software developers.

The use of GQM ensured us that important aspects were defined before the planning and the execution of the experiment took place [34].

### B. Context Selection

To reduce external validity threats, we downloaded the used software from the web. The domains of these software

can be considered a good compromise of generality and industrial application. Larger and more complex software would be difficult to use in the experiment, so increasing the risks of failure [19]. In particular, we used four software implemented in C. The following are the missions (or problem statements) of these systems:

- **Agenda**<sup>1</sup>. It provides easy to use means for keeping track of personal contacts. It allows adding a new contact by specifying name, last name, telephone number, mobile number, birthday, and email address. Existing contacts can be searched by specifying one of the fields above. All the records stored in the system can be sequentially browsed. The user can load a file containing contacts, so letting to separately manage different contact lists.
- **Financial**<sup>2</sup>. It is a command line option price calculator. It uses Black-Scholes that is a mathematical description of financial markets and derivative investment instruments. The model develops partial differential equations whose solution (i.e., the BlackScholes formula) is widely employed to price European puts and calls on non-dividend paying stocks.
- **GAS-Station**<sup>3</sup>. It is a system to manage a GAS station. It takes as input the daily price of: Petrol-Oil, Diesel, and Compressed Natural GAS. The system is able to manage receipts and bills regarding the refueling of Petrol-Oil, Diesel, and Compressed Natural GAS. The bills are stored to be successively searched, modified, and browsed.
- **Hotel-Reservation**<sup>4</sup>. It is in charge of managing room reservations for an Hotel. To reserve a room the system asks for the fiscal code number of a client and the dates for the check-in and check-out. Reservations can be modified and deleted. The system also shows the status of all the rooms of the hotel.

The source code of the systems above contained some identifiers either abbreviated (e.g., `mobNum`, `lDsk`, or `buf` in Agenda) or not. Acronyms were also used as identifiers (e.g., CNG - Compressed Natural Gas - in GAS-Station). Comments were omitted from the code the subjects viewed. One of the authors created two versions for each system chosen: one version contained abbreviated identifiers, while the other full-word identifiers. This process was founded on the dictionary available at [www.cs.tut.fi/tlt/stuff/misc/babel.html](http://www.cs.tut.fi/tlt/stuff/misc/babel.html). In Table I, we report some details on the original versions of the systems chosen. The first column shows the name of the system. The number of Line Of Code (LOCs)

<sup>1</sup>[www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=2299&lngWId=3](http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=2299&lngWId=3)

<sup>2</sup>[www.paulgriffiths.net/program/c/finance.php](http://www.paulgriffiths.net/program/c/finance.php)

<sup>3</sup>[www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=11639&lngWId=3](http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=11639&lngWId=3)

<sup>4</sup>[www.vyomworld.com/source/code.asp?id=01&l=C\\_Projects&t=HotelReservationSystem](http://www.vyomworld.com/source/code.asp?id=01&l=C_Projects&t=HotelReservationSystem)

Table I  
EXPERIMENT OBJECTS

System	LOCs	# Identifiers	Abbrev. Identif. (%)
Agenda	867	44	29.5%
GAS-Station	1021	51	37.2%
Financial	397	37	37.8%
Hotel-Reservation	511	24	50%

Table II  
FAULTS INJECTED WITH THE KIMS MUTATION OPERATORS [16], [17]

System	LCO	LOR	CFD	VRO	# faults
Agenda	2	1	1	0	4
GAS-Station	1	1	1	1	4
Financial	0	1	0	1	2
Hotel-Reservation	0	0	1	1	2

is reported in the second column and the total number of identifiers is shown in the third column. The percentage of abbreviated identifiers is reported in the last column.

We injected the same faults in each variant (with abbreviated identifiers and full-word identifiers) of a given system. The injection process was based on the Kims mutation operators [16], [17]. Some of these operators are general and can be applied to any imperative programming language, while others are specific to object-oriented languages. We used the following subset of the mutation operators for imperative programming languages:

- LCO (Literal Change Operator) - Increase/decrease numeric values or swap Boolean literals.
- LOR (Language Operator Replacement) - Replace a language operator with other legal alternatives.
- CFD (Control Flow Disrupt operator) - Disrupt normal control flow: add/remove break, continue, return.
- VRO (Variable Replacement Operator) - Replaces a variable name with other names of the same type and of the compatible types.

As for Agenda, we injected four faults applying the mutation operators: LCO, LOR, and CFD. For example, we applied the LOR operator on the statement `if(temporaryNode != NULL)`, so obtaining `if(temporaryNode == NULL)`. Details on the performed mutations are summarized in Table II. The application of a mutation operator depends on the source code of a given software. We also tried as much as possible to balance the number of faults within each software: 4 faults in Agenda and GAS-Station (the largest) and 2 in Financial and Hotel-Reservation (the smallest). In addition, we balanced the application number of the mutation operators on the size of the software. The mutated versions of the four software used in the experiment were syntactically correct.

### C. Participants

We conducted the experiment under controlled conditions using *convenience sampling* from the population of novice

software developers with *students as participants*. The participants had the following characteristics:

- They were students of a course on *Algorithms and Data Structures*. The participants had passed all the exams related to the following courses: Procedural Programming and Object Oriented Programming I. In these courses the participants studied C/C++ and Java on university problems. The participants were 21 years old on average.

The students participated in the experiments on a voluntary basis: we did not force and we did not pay them for their participation. However, we awarded the students for their participation to the experiments with a bonus towards their final mark. Before the experiment, we clearly informed the participants about that.

#### D. Variable Selection

We considered the source code with extended identifiers as the *Control Group* and the group with abbreviated identifiers as the *Treatment Group*. Therefore, the independent variable was *Method*. It is a nominal variable that could assume the following two values: ABBR (source code with abbreviations) and FULL (source code with full-word identifier names).

The direct dependent variables are:

- *Completion time* - the time which the participant spent to accomplish the experiment task.
- *Identified faults* - the number of faults the participants correctly identified.
- *Fixed faults* - the number of faults the participants correctly fixed in the source code.

To calculate the first dependent variable, we used the time (expressed in minutes) to accomplish the task, which was directly recorded by each participant. Low values for the time mean that the participants were quicker in completing the experiment task.

The variables were measured checking the source code the participants gave back. As for identified faults, we asked the participant to manually mark the statement/s containing the fault by using the following source code comments: `/* fault start */` and `/* fault end*/`. Since we knew the injected faults, the values for the variable identified faults were easily to compute. This variable assumes values between 0 and the maximum number of faults injected (e.g., four for the system Agenda). The larger the value of that variable, the better was the ability of the participant to found faults in the source code.

As for the variable fixed faults, we checked if the participant correctly modified the statement/s affected by the mutation operator. Also in this case the values for that variable were easy to compute because we had the original version of the system as the oracle. Therefore, each fault was correctly identified if and only if the statement the

participant wrote was equivalent to that in the oracle. The larger the value of the variable fixed faults, the better was the ability of the participant to remove faults. For scant relevance, we did not consider the number of wrongly identified and fixed faults in source code.

The chosen variables complement each other - one describes the efficiency of the participant and the other two the quality of dealing with faults.

We also analyzed the effect of other independent variables (also called co-factors, from here on):

**Object.** It denotes the combination system (i.e., Agenda plus Financial and Gas-Station plus Hotel-Reservation) used as the **objects** in the experiment. The effect of the Object factor should not be confounded with the main factor.

**Trial.** It denotes in which experiment trial a particular participant was exposed to ABBR and FULL. As the participants worked on two different experimental objects in two laboratory trials/runs. We analyzed whether the order might affect the results.

#### E. Hypotheses Formulation

The following three null hypotheses have been formulated and tested:

**Hn0:** There is no statistically significant difference between the completion time for ABBR and FULL.

**Hn1:** There is no statistically significant difference between the number of identified faults with ABBR and FULL.

**Hn2:** There is no statistically significant difference between the number of fixed faults with ABBR and FULL.

The hypotheses are two-tailed because we did not expect a positive nor a negative effect of the abbreviated identifiers on the experiment tasks. Even though it can be postulated that the participants in the treatment group were provided with less information, we cannot hypothesize that this concern reduces ambiguities and hinders the identification and the removal of faults in the source code.

#### F. Design of the experiment

We used a within-participants counterbalanced experimental design (see Table III). This design ensures that each participant works on different experiment objects (i.e., A+F = Agenda plus Financial and G+H = GAS-Station plus Hotel-Reservation) in two laboratory trials/runs, using ABBR or FULL each time. We opted for that design because it is particularly suitable for mitigating possible carry-over effects<sup>5</sup>. We grouped the systems in the experiment objects according to their size, the number of injected faults, and the application domain.

<sup>5</sup>If a participant is tested first under the condition X and then under the condition Y, he/she could potentially exhibit better or worse performances under the second condition.

Table III  
EXPERIMENT DESIGN

Trial	Group A	Group B	Group C	Group D
First	A+F, ABBR	A+F, FULL	G+H, ABBR	G+H, FULL
Second	G+H, FULL	G+H, ABBR	A+F, FULL	A+F, ABBR

We used the participants' average grades as blocking factor: the groups in Table III are similar to each other with respect to the number of participants with high and low average grades<sup>6</sup>. The groups A and B contained 13 students, while 12 and 11 were into the groups C and D, respectively.

### G. Experiment Tasks

We asked the participants to perform the following tasks:

- 1) *Identifying the faults.* The participants were asked to identify faults in the source code of the two experimental objects according to the design of the experiment. For example, the participants in the group A were asked to work first on the object A+F with ABBR and then on the object G+H with FULL. Regarding the object A+F, we asked the participants to work first on Agenda and then on Financial. The participants were asked to work first on GAS-Station and then on Hotel-Reservation when dealing with the object G+H. For each system, the participants were provided with its mission (or problem statement) describing its intent and a bug report for each fault. The bug reports contain both the title of the bugs (i.e., a short description) and their (long) description. An example of bug report for Agenda is shown in Figure 1. The bug report of a system is the same independently from the method (i.e., ABBR and FULL). Bug reports contained neither abbreviated nor full-word identifier names. It is also worth mentioning that the participants were not provided with any tools to perform fault localization in source code.
- 2) *Fixing the faults.* The participants were asked to remove the faults previously identified.
- 3) *Filling in the post-experiment questionnaire.* We asked the participants to fill in a post-experiment survey questionnaire. This questionnaire contained questions about: the availability of sufficient time to complete the tasks and the clarity of the experimental material and objects. The post-experiment survey questionnaire is shown in Table IV. All the statements expected closed answers according to a five-point Likert scale [25]: (1) strongly agree; (2) agree; (3) neutral; (4) disagree; and (5) strongly disagree. The goal of that

<sup>6</sup>In Italy, the exam grades are expressed as integers and assume values in between 18 and 30. The lowest grade is 18, while the highest is 30. As suggested by Abrahão *et al.* [1], we consider here the average grade as low if it is below 24/30, high otherwise.

Table IV  
POST-EXPERIMENT SURVEY QUESTIONNAIRE

Id	Question
Q1	I had enough time to perform the tasks.
Q2	The task objectives were perfectly clear to me.
Q3	The bug reports were clear to me.
Q4	The source code was correctly indented.
Q5	I found difficult to understand the abbreviated identifiers.
Q6	The presence of abbreviated identifiers made difficult the identification and the fixing of faults in source code.
Q7	There is not difference in using abbreviated or full-word identifiers while identifying and fixing faults in source code.
Q8	I found useful the experiment from the practical point of view.

questionnaire was to obtain feedback about the participants' perceptions of the experiment execution.

- 4) *Expanding abbreviations.* The participants were also asked to expand all the abbreviated identifiers in the source code they studied in the first two tasks above when using ABBR. The goal was to assess whether or not the participants correctly associated full-word identifiers to the abbreviated ones.

To perform both the laboratory trials, the participants were provided with laptops having the same hardware configuration (i.e., equipped with a 2.93 GHz i3-560 Processor with 4 GB of RAM and Windows 7). To surf, run, execute, and debug source code, we installed on each laptop Dev-C++ (available at [sourceforge.net/projects/orwelldevcpp/](http://sourceforge.net/projects/orwelldevcpp/)) an Integrated Development Environment (IDE) for C/C++. We opted for this IDE because it widely used in academic programming courses (e.g., in the procedural programming course). In addition, Dev-C++ can be considered easy to learn and to master.

The identifiers and the bug reports were in English. The complete list of all the bug reports and the source code of the four systems (both with abbreviated and full-word identifiers) are available on the web page of our experiment.

### H. Experiment operation

The participants first attended an introductory lesson in which the supervisors presented detailed instructions on the experiment. The supervisors highlighted the goal of the experiment without providing details on the experimental hypotheses. The participants were informed that the data collected in the experiments were used for research purposes and treated confidentially.

Title	Search for telephone number
Description	The search by telephone number does not produce any output. In particular, when a telephone number is provided as input and it is present in the contact list the system shows the message "no matches found".

Figure 1. A sample fault of the Agenda system

To familiarize with the experimental procedure, the participants accomplished an exercise similar to those which would appear in the laboratory trials. The software used in that exercise was the “Tower of Hanoi”. We download it from [www.paulgriffiths.net/program/c/hanoi.php](http://www.paulgriffiths.net/program/c/hanoi.php).

After the introductory lesson and training session, the participants were assigned to the groups A, B, C, and D (see Table III). No interaction was permitted among the participants, both within each laboratory trial and while passing from the first trial to the second one. No time limit for performing each of the two trials was imposed.

To carry out the experiment, the participants first received the material for the first laboratory trial. When the participants had finished, the material for the second trial was provided to them. After the completion of both the trials, we gave the participants the post-experiment questionnaire.

We asked the participants to use the following experimental procedure: (i) specifying name and start-time; (ii) identifying (and marking) the faults from the bug reports; (iii) fixing the bugs; and (iv) marking the end-time. The participants could compile and execute the applications before and after each modification performed on the source code. The debugger of Dev-C++ could be also used. We did not suggest any approach to surf, run, execute, and debug source code.

### I. Analysis Procedure

We performed the following steps:

- 1) We calculated the descriptive statistics of the dependent variables.
- 2) We tested the null hypotheses using unpaired analyses because the experimental tasks were accomplished on two different objects (see Table III). We have planned to use unpaired t-test when the data follow a normal distribution. The normality has been verified using the Shapiro-Wilk W test [30] (simply Shapiro in the following). A p-value smaller than the  $\alpha$  threshold allows us to reject the null hypothesis and to conclude that the distribution is not normal. If the data are not normally distributed, our non-parametric alternative to the unpaired t-test was the Wilcoxon rank-sum test (also known as Mann-Whitney) [9].

The chosen statistical tests analyze the presence of a significant difference between independent groups, but they do not provide any information about that difference [15]. Therefore, in the context of the parametric analyses, we used Cohen’s  $d$  [8] effect size to obtain the standardized difference between two groups. That difference can be considered: negligible ( $|d| < 0.2$ ), small ( $0.2 \leq |d| < 0.5$ ), medium ( $0.5 \leq |d| < 0.8$ ), and large ( $|d| \geq 0.8$ ) [15]. Conversely, we used the point-biserial correlation  $r$  in case of non-parametric analyses. The magnitude of the effect size measured using the point-biserial correlation is: small ( $0 < r \leq$

0.193), medium ( $0.193 < r \leq 0.456$ ), and large ( $0.456 < r \leq 0.868$ ) [15].

We also analyzed the statistical power for each test performed. The statistical power is the probability that a test will reject a null hypothesis when it is actually false. The statistical power is computed as 1 minus the Type II error (i.e.,  $\beta$ -value). This kind of error indicates that the null hypothesis is false, but the statistical test erroneously fails to reject it. In the discussion of the results, the  $\beta$ -value has to be used when a statistical test is not able to reject the null hypotheses. The higher the  $\beta$ -value, the less is the likelihood of not rejecting a null hypothesis when it is actually false.

- 3) To analyze the influence of the co-factors, we planned to use a two-way Analysis of Variance (ANOVA) [10]. To apply this test data have to be normally distributed and their variance has to be constant. The normality and the variance of the data were tested using the tests of Shapiro and Levene [24], respectively. In case these two assumptions are not verified, we would use a two-way permutation test [2], a non-parametric alternative to the two-way ANOVA test.
- 4) To graphically summarize the answers to the post-experiment survey questionnaire, we planned to use boxplots. They provide a quick visual representation to summarize the data using five numbers: the median, upper and lower quartiles, minimum and maximum values, and outliers.

In all the statistical tests, we decided (as custom) to accept a probability of 5% of committing Type-I-error [34] (i.e., the  $\alpha$  threshold is 0.05). The R environment<sup>6</sup> for statistical computing has been used in the data analyses.

## III. RESULTS

In this section, we present the data analysis following the procedure presented above.

### A. Descriptive statistics and exploratory analysis

Table V reports some descriptive statistics (i.e., median, mean, and standard deviation) of the three dependent variables on Method. A summary of the performed analyses are also shown: the p-values, the effect size, and the statistical power ( $\beta$ -values are between brackets). The observations are grouped by Method.

The descriptive statistics show any tendency in favor of neither ABBR nor FULL. In particular, we can observe that the participants achieved slightly larger values for completion time when using ABBR. As for the other two dependent variables, descriptive statistics are almost the same. In addition, the descriptive statistics suggest that the participants generally were able to fix all the correctly identified faults.

<sup>6</sup>[www.r-project.org](http://www.r-project.org)

Table V  
DESCRIPTIVE STATISTICS AND RESULTS. THE SYMBOL “\*” INDICATES THAT PARAMETRIC ANALYSES HAVE BEEN PERFORMED

Var.	FULL			ABBR		p-value	effect size d or r	statistical power	
	median	mean	st. dev.	mean	st. dev.				
Completion time*	73	74.92	32.006	77	78.02	33.503	0.64	negligible (-0.095)	0.054 ( $\beta$ - value = 0.946)
Identified faults	4	4.286	1.307	4	4.204	1.338	0.748	small ( 0.068)	0.063 ( $\beta$ - value = 0.937)
Fixed faults	4	3.837	1.297	4	3.816	1.269	0.93	small (0.149)	0.047 ( $\beta$ - value = 0.953)

### B. Effect of method

The data were normally distributed on completion time. The Shapiro test returned 0.516 and 0.881 as the p-values for ABBR and FULL, respectively. Then, we applied parametric analyses. In particular, the results suggest that Hn0 (effect of Method on completion time) cannot be rejected since the p-value is 0.64. The  $\beta$  - value is 0.946 (i.e., it is high the likelihood that the statistical test correctly fails to reject the null hypothesis). The effect size is negligible (-0.095).

As far as the other two dependent variables, we performed non-parametric analyses because the data were not normally distributed. The Mann-Whitney test failed to reject both the hypotheses Hn1 (effect of Method on identified faults) and Hn2 (effect of Method on fixed faults). The p-values are 0.748 and 0.93, respectively. For both the hypotheses the  $\beta$ -values are high: 0.937 and 0.953, respectively. The effect size is always small (i.e., 0.068 and 0.149 on the identified and fixed faults, respectively).

### C. Other factors

The results of the analysis of the co-factors is summarized in Table VI. This table reports the p-values of the statistical tests employed to verify whether or not a co-factor has any effect on each of the dependent variables. The results about the interaction between the co-factors and Method are reported as well. We could apply a two-way ANOVA on Object and Trial for completion time. As far as Object, the Shapiro test returned 0.758 and 0.999 as the p-values for A+F and G+H on ABBR, respectively. For FULL, this test returned as the p-values 0.461 for A+F and 0.695 for G+H. The Levene test returned 0.793 as the p-value for Method and 0.695 as the p-value for Object. As for Trial, the p-values obtained by applying the Shapiro test on ABBR are: 0.786 (first trial) and 0.729 (second trial). On the other hand, the p-values are 0.607 and 0.611 for the first and the second laboratory trials, respectively. The Levene test returned 0.229 as the p-value for Trial.

As for the other two dependent variables, the assumptions to use a two-way ANOVA were violated and then we applied a two-way permutation test.

1) *Object*: The results of a two-way ANOVA show that the effect of Object on completion time was not statistically significant (p-value = 0.318) and that there was not a statistically significant interaction between Method and Object (p-value = 0.688).

Table VI

RESULT OF THE ANALYSIS ON THE CO-FACTORS. THE SYMBOL “\*” INDICATES THAT PARAMETRIC ANALYSES HAVE BEEN PERFORMED

Variable	Object	Object vs. Method	Trial	Trial vs. Method
Completion time*	0.318	0.688	<b>0.002</b>	0.987
Identified faults	0.941	0.156	0.882	0.902
Fixed faults	0.882	0.119	0.36	0.921

As far as the variable identified defects, the results of the two-way permutation test indicate that the effect of Object was not statistically significant (p-value = 0.941). Also, the interaction between Object and Method was not statistically significant (p-value = 0.156). Similar results were achieved for the variable fixed faults. For Object, the p-value is equal to 0.882. For the interaction between Object and Method, the p-value is 0.119.

2) *Trial*: The results of a two-way ANOVA show that the effect of Trial on completion time was statistically significant. The p-value is 0.002 (highlighted in bold in Table VI). The descriptive statistics show that the participants spent less time to accomplish the second laboratory trial (medians: 62 vs. 81; mean 66.51 vs. 86.43). The results of the two-way ANOVA test show that there is not a statistically significant interaction between Method and Trial: the p-value is equal to 0.987.

The results of the two-way permutation test show that there was not a statistically significant effect of Trial on the variables identified faults (p-value = 0.882) and fixed faults (p-value = 0.36). The results of that test also indicate that the interaction between Trial and Object is not statistically significant. The p-values are 0.902 and 0.921 for the variables identified faults and fixed faults, respectively.

### D. Post-experiment survey questionnaire

Figure 2 shows the boxplots of the answers the participants gave to the statements of the post-experiment survey questionnaire. The participants judged adequate the time to perform the tasks (Q1). The medians is equal to 1 (strongly agree). The participants agreed on the fact that the task objectives were clear (Q2). The median is 2 (agree). For Q3 (bug report clear) and Q4 (source code correctly indented), the median is 2 (agree). The participants found difficult to: (i) understand abbreviated identifiers (Q5) and (ii) accomplish the tasks when abbreviated identifiers were present in the source code (Q6). The median is 2 (agree) for both Q5 and Q6. The participants also asserted that there

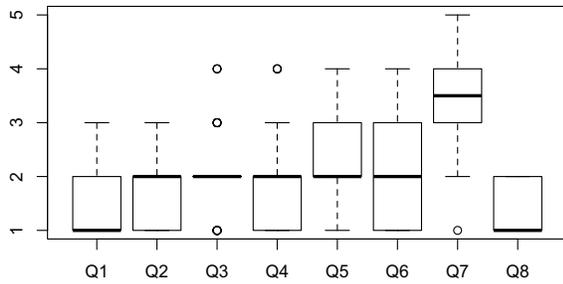


Figure 2. Boxplot of the answers to the survey questionnaire

is a difference in identifying and fixing faults when either abbreviated or full-word identifiers are in the source code (Q7). The median is 4 (disagree). The results on Q5, Q6, and Q7 contrast with the quantitative ones presented in Section III-B. This is the case where there is difference between perceived and effective complexity in the execution of a task. Finally, the answers to Q8 suggested that the participants found useful the experiment from the practical point of view. The median is 1 (strongly agree).

#### IV. DISCUSSION

Our “null-results” (i.e., the defined null hypotheses have not been rejected) are interesting because they falsify the *common sense* that a difference should be measured when dealing with and without abbreviated identifier name in the execution of comprehension tasks [23]. In practical terms, the two different representations for source code identifiers would transmit the same underlying conceptual content and its mean has been similarly inferred by the participants to our experiment when dealing with faults in source code. That is, abbreviated identifiers did not penalize the identification and the removal of faults in source code. To get some indications to explain the results above, we performed unstructured interviews with the participants after the data analyses. We observed that all the participants daily use social media, such as Facebook, Google+, and Twitter, and are comfortable with the social media slang that they prevalently use (often unconsciously) to communicate. Therefore, it could be possible that the familiarity with the social media slang, which is strongly based on abbreviations, improves the ability to infer the underlying conceptual content of abbreviated identifiers. Although this point deserves future investigations to increase our awareness in the achieved results, we believe that it delineates some research trends in the field of software engineers. The most interesting trend is: *how social media slang changes the programmers’ habit in writing and comprehending source code.*

With regard to completion time, the results showed that the participants spent more time to accomplish the assigned tasks when the source code contained abbreviated identifiers. The difference in the execution of the tasks when identifiers

are abbreviated or not is not statistically significant (see Table V). We can postulate that this difference is due to the additional time the participants needed to infer the underlying conceptual content of the abbreviated identifiers. This postulation is also corroborated by the fact that the effect of Object is not statistically significant.

We observed that the effect of Trial is statistically significant on completion time, while it is not on the other two dependent variables. This result suggests that an increasing familiarity of the participants with the kind of task and/or with the used IDE affected the time to complete the second trial, but not the ability of these participants to identify and fix faults in source code.

#### A. Implications

To judge the implications of our experiment, we adopted a perspective-based approach [3]. In particular, we based our discussion on the *practitioner/consultant* (simply *practitioner* in the following) and *researcher* perspectives [18]. The main practical implications of our study can be summarized as follows:

- The ability of participants to identify and fix faults in source code seems independent from how the identifiers are written. From the practitioner perspective, this result is relevant because source code could be written without paying attention on the names of identifiers. From the researcher perspective, it is interesting to investigate whether variations in the context (e.g., larger systems and more or less experienced developers) lead to different results. The researcher could be also interested in studying why the participants perceived more complex the execution of the tasks on source code with abbreviated identifiers (see Section III-D).
- The time to infer the underlying conceptual content of abbreviated identifiers is slightly larger than that of expanded identifiers. This result is relevant for the researcher because it is interesting to investigate how participants (independently from the experience) associate the underlying conceptual content to the identifiers. In addition, the researcher could be interested in analyzing whether writing source code with contractions/abbreviations reduces/increases the effort and cost to software development. On the other hand, the practitioner could be interested in the relation between the effort and cost to write source code with and without abbreviated identifiers and the effort and cost to identify and fix faults.
- The study is focused on desktop applications implemented in C. The researcher and the practitioner could be interested in answering the question: do the results observed hold for different kinds of software (e.g., web application) and/or implemented in different programming languages (e.g., PHP and Java)?

- The software used in the experiment was realistic enough. Although we are not sure that our findings scale to real systems, the obtained results could be true in all the cases in which fault detection is executed on small/medium sized software. Both practitioners and researchers could be interested in this concern.
- The achieved results also suggest that the task completion time and fault identification and fixing are not directly related. Our study poses the bases of future investigations in that direction. This result is interesting from the researcher and practitioner perspectives.

## V. THREATS TO VALIDITY

In the following subsections, we discuss the possible threats that could affect the validity of our experiment.

### A. Internal validity

Internal validity threats are diminished by the design of the experiments we adopted. Each group of participants worked on two different experiment objects, each containing either abbreviated or full-word identifiers.

Internal validity threat was also mitigated because the participants had a similar amount of experience with the C programming language, computer programming, and software maintenance. Furthermore, all the participants found the material, the tasks, and the goals clear, as the post-experiment survey questionnaire results showed.

Another issue concerns the exchange of information among the participants. The participants were not allowed to communicate with each other. We prevented this by monitoring them both during the laboratory trials and during the break between the two trials.

### B. External validity

The use of students could lead doubts concerning their representativeness with regard to software professionals. The tasks to be performed did not require a high level of industrial experience, so we believed that the use of students could be considered appropriate, as suggested in the literature [7] [12]. In addition, students could be more comfortable than senior software practitioners with contractions/abbreviations since they daily use social networks. This implies that the participants to our experiment could be more proficient in inferring the right meaning to abbreviated identifier with respect to senior professionals.

Working with students also implies various advantages, such as the fact that the students' prior knowledge is rather homogeneous, there is the possible availability of a large number of participants [33], and there is the chance to test experimental design and initial hypotheses [31]. An additional advantage in using students is that the cognitive complexity of the experiment objects is not hidden by the experience of the participants.

Another threat to external validity concerns the used experiment objects. The experimenters were not involved in the realization of these objects, so reducing such a possible threat. The size of the used systems could also threaten external validity as well as the language used for the identifiers. Given the context of the experiment, the familiarity of the participants with the English language could have affected the observed results. In addition, the mutation operators applied on the original versions of the chosen software systems could affect external validity, namely different faults could lead to different results.

### C. Construct validity

This kind of validity may be influenced by the used measures, the post-experiment survey questionnaire, and social threats. We used a well-known method [16], [17] to inject faults in a software and used its original version as the oracle to compute the number of identified and fixed faults. In addition, the injected faults were sufficiently complex without being too obvious. We also tried as much as possible to inject different kinds of faults. The post-experiment survey questionnaire was designed using standard forms and scales [25]. Finally, the students were not graded on the results obtained in the experiments to avoid social threats (i.e., evaluation apprehension).

### D. Conclusion validity

The used statistical tests to reject the null hypotheses could affect conclusion validity. Depending on the cases, we used unpaired parametric and non-parametric tests. A power analysis has been performed as well. Threats to the conclusion validity could be also concerned to our unstructured interviews.

## VI. RELATED WORK

Source code comprehensibility and modifiability have been largely investigated in software maintenance field through quantitative and qualitative studies (e.g., [11], [27], [29], [35]). For example, Woodfield *et al.* [35] reported on an experiment to investigate how different types of modularization and comments are related to programmers' ability to understand programs. The experimenters used eight versions of the same program. These versions were the result of the program being constructed with four types of modularization (i.e., monolithic, functional, super, and abstract data type) each with and without source code comments. The participants to this study were 48 experienced programmers. The comprehension level achieved by these participants was measured by using a questionnaire. The results indicated that those participants whose programs contained comments were able to answer more questions than those without comments. Similarly, Takang *et al.* [32] conducted an controlled experiment founded on program comprehension theories.

Three were the investigated hypotheses: (i) commented programs are more understandable; (ii) programs that contain full-word identifier names are more understandable; and (iii) the combined effect of comments and identifier names tend to enhance the understandability. The experiment was conducted on four versions of a program written in Modula-2 on two methods. The only hypothesis the authors were able to positively answer was the first. In addition, the authors also observed that the quality of identifier names is an issue that merits closer consideration and exploration in its own right. Based on this finding, our study was concerned with the use of different kinds of identifier names on fault identification and fixing.

Lawrie *et al.* [22] conducted a quantitative study with 100 programmers to investigate whether source code comprehension is affected by how identifier names are written. In particular, the authors considered identifiers of 12 functions (i.e., algorithms studied in computer science courses and functions extracted from production code) written as: single letters, abbreviations, and full-words. The comprehension achieved by the participants on these functions was assessed through comprehension questionnaires. The results showed that full-word identifiers lead to the best comprehension with respect to abbreviated identifiers, even if this difference is not statistically different. Successively, Lawrie *et al.* extended the data analysis [23]. The achieved results somewhat contrast with ours: we did not find a huge difference between the use of full-word and abbreviated identifiers. This difference in the observed results could be related to the familiarity of the participants with the social media slang. In fact, in the 2006 and 2007 (the publication years of [22], [23], respectively) social networks were not yet a part of everyday life. That is, social networks were not yet the standard communication media for young people.

Binkley *et al.* [5] presented a family of studies to investigate the impact of two identifier styles (i.e., *camel case* and *underscore*) on the time and accuracy of comprehending source code. The studies involve 150 participants with varied demographics from two different universities. The results suggested that experienced software developers appeared to be less affected by identifier style, while beginners benefited from the use of camel casing with respect to task completion time and accuracy. Several are the differences between the studies by Binkley *et al.* and that reported in this paper.

As far as qualitative studies, Roehm *et al.* [27] conducted an observational study with professional developers on the strategies they followed, the information needed, and tools used to deal with comprehension tasks. The results of that qualitative investigation showed that developers did not use tools for program comprehension and were unaware of them. Therefore, running the system, source code, and communication among developers represented the only source of information to perform comprehension tasks. This is why we provided participants with the only source code.

Ko *et al.* [20] performed a study to understand how developers gain source code comprehension and how an IDE (i.e., Eclipse) might be involved. The authors found that developers interleaved three activities: searching for relevant code both manually and using search tools; following incoming and outgoing dependencies of relevant code; and collecting code and other information. This was the reason because we decided to provide the participants in the experiment with a software tool to surf, run, execute, and debug source code.

## VII. CONCLUSION

In this paper, we present a controlled experiment to assess whether identifier names have any effect on the identification and the removal of faults in C source code. The results indicated that the difference in using abbreviated and full-word identifiers is not statistically significant with respect to: the time to complete the tasks and the number of faults the participants identified and fixed.

As future work, we are going to study the relation between the ability of the participants in correctly associating full-word identifiers to the abbreviated ones and the number of faults they correctly identified and fixed. It is subject of future research also the investigation on the relation between the familiarity of software engineers with social slang and their comprehension of source code.

## ACKNOWLEDGMENT

Special thanks to all the participants as this work would not be possible without your time. Our thanks to the Mas-similiano Di Penta for his comments and suggestions.

## REFERENCES

- [1] S. M. Abrahão, C. Gravino, E. I. Pelozo, G. Scanniello, and G. Tortora. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Trans. on Soft. Eng.*, 39(3), 2013.
- [2] R. Baker. Modern permutation test software. In E. Edgington, editor, *Randomization Tests*, Marcel Dekker, 1995.
- [3] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, L. S. Sørungård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.
- [4] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Software Eng.*, 14(6):758–773, 1988.
- [5] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.
- [6] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *Workshop on the Future of Software Engineering*, pages 326–341, 2007.

- [7] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proc. of International Symposium on Software Metrics*, pages 239–. IEEE CS Press, 2003.
- [8] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.
- [9] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd Edition, 1998.
- [10] J. L. Devore and N. Farnum. *Applied Statistics for Engineers and Scientists*. Duxbury, 1999.
- [11] C. Gravino, M. Risi, G. Scanniello, and G. Tortora. Do professional developers benefit from design pattern documentation? a replication in the context of source code comprehension. In *Proc. of International Conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 185–201. Springer, 2012.
- [12] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects: comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, Nov. 2000.
- [13] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting experiments in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer London, 2008.
- [14] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, Englewood Cliffs, NJ, 2001.
- [15] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information & Software Technology*, 49(11-12):1073–1086, 2007.
- [16] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of Java mutation operators using HAZOP. In *Proceedings of International Conference on Software & Systems Engineering and their Applications*, pages 9–10, 1999.
- [17] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. of NET.OBJECTDAYS*, pages 9–12, 2000.
- [18] B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13(1):97–121, 2008.
- [19] B. Kitchenham, S. Pflieger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. on Soft. Eng.*, 28(8):721–734, 2002.
- [20] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, Dec. 2006.
- [21] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of International Conference on Software engineering*, pages 492–501. ACM, 2006.
- [22] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What’s in a name? a study of identifiers. In *Proc. of International Conference on Program Comprehension*, pages 3–12. IEEE CS Press, 2006.
- [23] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.
- [24] H. Levene. Robust tests for equality of variances. In I. Olkin, editor, *Contributions to probability and statistics*. Stanford Univ. Press., Palo Alto, CA, 1960.
- [25] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.
- [26] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, December 2004.
- [27] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of International Conference on Software Engineering*, pages 255–265. IEEE CS Press, 2012.
- [28] G. Scanniello, A. D’Amico, C. D’Amico, and T. D’Amico. Using the Kleinberg algorithm and Vector Space Model for software system clustering. In *Proc. of International Conference on Program Comprehension*, pages 180–189. IEEE Computer Society, 2010.
- [29] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora. On the impact of UML analysis models on source code comprehensibility and modifiability. *ACM Trans. on Soft. Eng. and Meth.*, (to appear).
- [30] S. Shapiro and M. Wilk. An analysis of variance test for normality. *Biometrika*, 52(3-4):591–611, 1965.
- [31] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.
- [32] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: An experimental study. *Journal of Programming Languages*, 4(3):143–167, 1996.
- [33] J. Verelst. The influence of the level of abstraction on the evolvability of conceptual models of information systems. In *Proc. of the International Symposium on Empirical Software Engineering*, pages 17–26. IEEE CS Press, 2004.
- [34] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.
- [35] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proc. of the international conference on Software engineering*, pages 215–223. IEEE CS Press, 1981.

# DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis

Yuan Tian<sup>1</sup>, David Lo<sup>1</sup>, and Chengnian Sun<sup>2</sup>

<sup>1</sup>Singapore Management University, Singapore

<sup>2</sup>National University of Singapore, Singapore

{yuan.tian.2012,davidlo}@smu.edu.sg, suncn@comp.nus.edu.sg

**Abstract**—Bugs are prevalent. To improve software quality, developers often allow users to report bugs that they found using a bug tracking system such as Bugzilla. Users would specify among other things, a description of the bug, the component that is affected by the bug, and the severity of the bug. Based on this information, bug triagers would then assign a priority level to the reported bug. As resources are limited, bug reports would be investigated based on their priority levels. This priority assignment process however is a manual one. Could we do better? In this paper, we propose an automated approach based on machine learning that would recommend a priority level based on information available in bug reports. Our approach considers multiple factors, temporal, textual, author, related-report, severity, and product, that potentially affect the priority level of a bug report. These factors are extracted as features which are then used to train a discriminative model via a new classification algorithm that handles ordinal class labels and imbalanced data. Experiments on more than a hundred thousands bug reports from Eclipse show that we can outperform baseline approaches in terms of average F-measure by a relative improvement of 58.61%.

## I. INTRODUCTION

Due to system complexity and inadequate testing, many software systems are often released with defects. To address these defects and improve the next releases, developers need to get feedback on defects that are present in released systems. Thus, they often allow users to report such defects using bug reporting systems such as Bugzilla, Jira, or other proprietary systems. Bug reporting is a standard practice in both open source software development and closed source software development (e.g., Windows).

Although bug reporting could potentially improve software quality, the number of such reports could be too many for developers to handle. In 2005, it is reported that “everyday almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle” [2]. Thus developers often need to prioritize which bugs are to be given attention first. In Bugzilla there are 5 priority levels: P1, P2, P3, P4, and P5. P1 is the highest priority and often the software system can only be shipped if these high priority bugs are fixed. P5 on the other hand is the lowest priority and bugs assigned this priority might remain unfixed for a long period of time.

Prioritizing bugs is a manual process and is time consuming. Bug triagers need to read the information provided by users in the new bug reports, compare them with existing reports,

and decide the appropriate priority levels.

To aid bug triagers in assigning priority, we propose a new automated approach to recommend priority levels of bug reports. To do so, we leverage information available in the bug reports. Bug reports contain various information including short and long descriptions of the issues users encounter while using the software system, the products that are affected by the bugs, the dates the bugs are reported, the people that report the bugs, the estimated severity of the bugs, and many more. We would like to leverage this information to predict the priority levels of bug reports.

Our approach predicts the priority level of bug reports by considering several factors that could affect the priority of a bug report. These factors include other bug reports that are reported at the same time as the bug report (temporal), the textual content of the bug report (textual), the author of the bug report (author), related bug reports (related-report), the estimated severity of the bug (severity), and the product which the reported bug affects (product). We extract various features, e.g., the number of bugs reported in the past 7 days, etc., to capture each of these factors.

Next, we propose a new machine learning approach, in particular a new classification algorithm, to create a model from the features that could predict if a bug report should be assigned a priority level P1, P2, P3, P4, or P5. We take a training set of reports along with the priority levels. Feature values are then extracted from each data point (i.e., bug report) in this training set. The machine learning algorithm would then decide the likely priority level of a bug report whose priority level is to be predicted based on these feature values.

We propose a new framework named DRONE (PreDICTing PRiority via Multi-Faceted FactOr ANalysEs) to aid triagers in assigning priority labels to bug reports. Inside DRONE, we include our new classification engine named GRAY (ThresholdinG and Linear Regression to ClAssify Imbalanced Data) which enhances linear regression with our *thresholding approach* to handle imbalanced bug report data. Linear regression models the relationship between the values of the various features and the priority levels (which takes a value between 1 to 5) of bug reports. Linear regression considers the priority levels as ordinal values (i.e., P1 is closer to P2 than it is to P5) rather than categorical values (i.e., P1 is as different to P2 as it is to P5). It then outputs a real number given

a set of values of the different features of a new bug report. Thresholding learns a set of *thresholds* defining a set of *ranges* for the outputs of the linear regression model where each range corresponds to a priority level. Due to the data imbalance, the best set of ranges are of unequal sizes and these ranges could be effectively learned from a set of validation data points, thus addressing the data imbalance issue.

Closest to our work, is the series of work on bug report severity prediction by Menzies and Marcus [16], Lamkanfi et al. [13], [14], and our own previous work [27]. These studies predict the severity field of a bug report based on the textual content of the report. Severity however is different from priority. Severity is assigned from a user perspective while priority is assigned based on the developers' perspective. We have checked with an experienced developer from Eclipse Project Management Committee, who has fixed hundreds of bugs in Eclipse. He states that:

Severity is assigned by customers [users] while priority is provided by developers ... customer [user] reported severity does impact the developer when they assign a priority level to a bug report, but it's not the only consideration. For example, it may be a critical issue for a particular reporter that a bug is fixed but it still may not be the right thing for the eclipse team to fix.

Thus our work is different from the work on bug severity prediction. In this work, we predict the priority of bug reports by considering the temporal, textual, author, related-report, severity, and product factors of a bug report. This holistic view of a bug report is needed for us to support triagers in assigning priority levels to bug reports.

We experiment our solution on more than a hundred thousand bug reports of Eclipse that span a period of several years. We compare our approach with a baseline solution that adapt an algorithm by Menzies and Marcus [16] for bug priority prediction. Our experiments demonstrate that we can achieve 58.61% improvement on the average F-measure.

The contributions of this work are as follows:

- 1) We propose a new problem of predicting the priority of a bug given its report. Past studies on bug report analysis has only considered the problem of predicting the severity of bug reports which is an orthogonal problem.
- 2) We predict priority by considering the different factors that potentially affect the priority level of a bug report. In particular, we consider the following factors: temporal, textual, author, related-report, severity, and product.
- 3) We introduce a new machine learning framework, named DRONE, that would consider these factors and predict the priority of a bug given its report. We also propose a new classification engine, named GRAY, which is a component of DRONE, that *enhances* linear regression with *thresholding* to handle imbalanced data.
- 4) We have experimented our solution on more than a hundred thousands bug reports from Eclipse in its ability

to support developers in assigning priority levels to bug reports. The result shows that DRONE could outperform a baseline approach, built by adapting a bug report severity prediction algorithm, in terms of average F-measure, by a relative improvement of 58.61%.

The structure of this paper is as follows. In Section II, we describe preliminary information on bug report, text pre-processing, and measuring similarity of bug reports. In Section III, we describe our proposed approach. Section IV presents the result of our experiments. Next, we discuss interesting issues in Section V. Related work is presented in Section VI. Finally, we conclude and discuss future work in Section VII.

## II. PRELIMINARIES & PROBLEM DEFINITION

In this section, we first describe bug reports and bug report reporting process. Next, we present an approach to pre-process textual documents. Then, we highlight REP [22], which is a recently proposed state-of-the-art similarity measure of bug reports. Finally, we present our problem definition.

### A. Bug Reports and Reporting Process

Developers often desire feedback on defects that exist on released systems. To collect feedback, bug tracking systems are often employed. Popular bug tracking systems include Bugzilla, Jira, and other proprietary systems. Utilizing these systems, users can report issues that they find in the system and track their progress. Each reported issue is referred to as a bug report.

Each bug report contains information on how the bug could be reproduced plus other related information that could help in debugging. In a bug report, there is information on short and long descriptions of the bug, and various information on the product that is affected by the bug, the component that is affected by the bug, the estimated severity of the bug, the date that the bug is reported, and many more. All this information is commonly provided by bug reporters when they submit bug reports. We provide a description of fields in a bug report that are of interest to us in Table I.

When a new bug report is submitted into a bug tracking system, a bug triager would first investigate the fields of the bug report and potentially other reports. Based on the investigation, he or she would check the validity of the bug report and assign an appropriate priority level. Some bugs are also reported as duplicate bug reports at this point. This is possible due to the distributed nature of the bug reporting process – i.e., users from various parts of the world could encounter the same defect and create different bug reports. We show some example bug reports from Eclipse in Table II. Note that bug reports shown in the same box (e.g., 4629 and 4664) are duplicates of one another. After assigning a priority level to the bug report, bug triager would forward the bug to a developer to fix it. The developer then works on the bug and eventually comes up with a resolution.

TABLE I  
FIELDS OF INTEREST IN A BUG REPORT

Field	Description
Summary	Summarized description of a bug. Typically this summary only contains a few keywords
Description	Long description of a bug. Typically this would include information that would help in the debugging process including the reported error message, the steps to reproduce the error, etc.
Product	The product which is affected by the bug
Component	The component which is affected by the bug
Author	The author of the bug report
Severity	The estimated impact of a bug as perceived by the reporter of the bug. There are several severity labels including <code>blocker</code> , <code>critical</code> , <code>major</code> , <code>normal</code> , <code>minor</code> , and <code>trivial</code> . Aside from these severity levels, there is one additional severity level that denotes feature requests, i.e., <code>enhancement</code> . In this study, we ignore bug reports with this severity label as we focus on defects and not feature requests.
Priority	The priority of a bug to be fixed which is assigned by a bug triager. When the bug report is submitted, this field would be blank. The triager would then decide an appropriate priority level for a bug report. There are five priority levels: P1, P2, P3, P4, and P5.

TABLE II  
EXAMPLES OF BUG REPORTS FROM ECLIPSE

	ID	Summary	Product	Component	Severity	Priority
1	4629	Horizontal scroll bar appears too soon in editor (1GC32LW)	Platform	SWT	normal	P4
	4664	StyledText does not compute correct text width (1GELJXD)	Platform	SWT	normal	P2
2	4576	Thread suspend/resume errors in classes with the "same" name	JDT	Debug	normal	P1
	5083	Breakpoint not hit	JDT	Debug	normal	P1
3	4851	Print ignores print to file option (1GKXC30)	Platform	SWT	normal	P3
	5126	StyledText printing should implement "print to file"	Platform	SWT	normal	P3

### B. Text Pre-Processing

Here we present several standard pre-processing techniques to convert a textual document into a set of features. These pre-processing techniques include tokenization, stop-word removal, and stemming. We present each of them in the following paragraphs.

*Tokenization.* A textual document contains many words. Each of such words is referred to as a token. These words are separated by delimiters which could be spaces, punctuation marks, etc. Tokenization is a process to extract these tokens from a textual document by splitting the document into tokens according to the delimiters.

*Stop-Word Removal.* Not all words are equally important. There are many words that are frequently used in many documents but carry little meaning or useful information. These words are referred to as stop words. There are many of such stop words including "am", "are", "is", "I", "he", etc. These stop words need to be removed from the set of tokens extracted in the previous steps as they might affect the effectiveness of machine learning or information retrieval solutions due to their skewed distributions. We use a collection of 30 stop words and also standard abbreviations including, "I'm", "that's", etc.

*Stemming.* Words can appear in various forms; in English, various grammatical rules dictate if a root word appear in its singular, plural, present tense, past tense, future tense, or many other forms. Words originating from the same root word but are not identical with one another are semantically related. For example, there is not much difference in meaning between "write" and "writes". In the text mining and information retrieval community, stemming has been proposed to address

this issue. Stemming would try to reduce a word to its *ground* form. For example, "working", "worked", and "work" would all be reduced to "work". There are various algorithms that have been proposed to perform stemming. In this work, we use the Porter's stemming algorithm [21] to process the text as it is commonly used by many prior studies, e.g., [16], [13], [14], [29].

### C. Measuring Similarity of Bug Reports

Various techniques have been proposed to measure the similarity of bug reports. A number of techniques model a bug report as a vector of weighted tokens. Similarity of two bug reports can then be evaluated by computing the Cosine similarity of their corresponding two vectors. These include the work by Jalbert and Weimer [9], Runeson et al. [19], Wang et al. [29], etc.

The most recent approach proposed to measure the similarity of bug reports is REP which is proposed by Sun et al. [22]. Their approach extends BM25F [18] which is a state-of-the-art measure for structured document retrieval. In their proposed approach past bug reports that have been labeled as duplicate are used as training data to measure the similarity of two bug reports. Various fields of bug reports are used for comparison including the textual and non-textual contents of bug reports. We use an adapted version of REP to measure the similarity of bug reports. REP includes the comparison of the priority fields of two bug reports to measure their similarity. In our setting, we would like to predict the values of the priority field. Thus, we remove the priority field from REP's analysis as they are unknown for bug reports whose priority levels are to be predicted. We call the resultant REP,  $REP^-$ .  $REP^-$  only compares the textual (summary and description), product, and component fields of two bug reports to measure

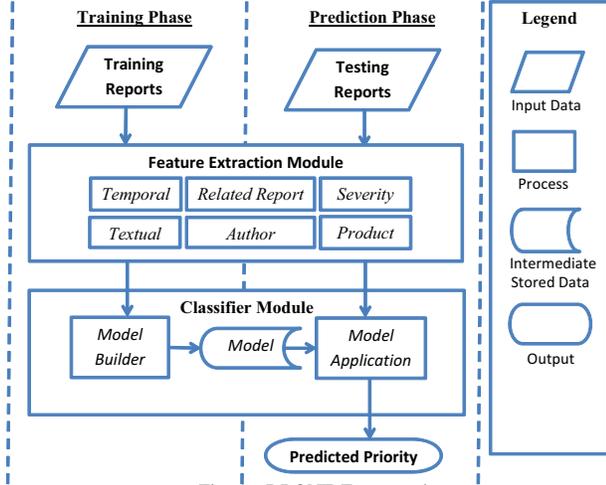


Fig. 1. DRONE Framework

their similarity.

#### D. Problem Definition

“Given a new bug report and a bug tracking system, predict the priority label of the new report as either P1, P2, P3, P4, or P5.”

### III. PROPOSED APPROACH

In this section, we describe our proposed framework. First we present the overall structure of our framework. Next, we zoom into two sub-components of the framework namely feature extraction and classification modules. In the feature extraction module, we extract various features that capture various factors that potentially affect the priority level of a bug report. In the classification module, we propose a new classification engine leveraging linear regression and thresholding to handle imbalanced data.

#### A. Overall Framework

Our framework, named DRONE (PreDICTing PRiority via Multi-Faceted FactOr ANALysEs), is illustrated in Figure 1. It runs in two phases: training and prediction. There are two main modules: feature extraction module and classification module.

In the training phase, our framework takes as input a set of bug reports with known priority labels. The feature extraction module extracts various features that capture temporal, textual, author, related-report, severity, and product factors that potentially affect the priority level of a bug report. These features are then fed to the classification module. The classification module would produce a discriminative model that could classify a bug report with unknown priority level.

In the prediction phase, our framework takes a set of bug reports whose priority levels are to be predicted. Features are first extracted from these bug reports. The model learned in the training phase is then used to predict the priority levels of the bug reports by analyzing these features.

Our framework has two placeholders: feature extraction and classification module. Various techniques could be put

into these placeholders. We describe our proposed feature extraction and classification modules in the following two subsections.

#### B. Feature Extraction Module

The goal of the feature extraction module is to characterize a bug report in several dimensions: temporal, textual, author, related-report, severity, and product. For each dimension, a set of features is considered. For each bug report  $BR$  our feature extraction module processes various fields of  $BR$  and a bug database of reports created prior to the reporting of  $BR$ . It would then produce a vector of values for the features listed in Table III.

Each dimension/factor is characterized by a set of features. For the temporal factor, we propose several features that capture the number of bugs that are reported in the last  $x$  days with priority level  $y$ . We vary the values of  $x$  and  $y$  to get a number of features (TMP1-12). Intuitively, if there are many bugs reported in the last  $x$  days with a higher severity level than  $BR$ ,  $BR$  is likely not assigned a high priority level since there are many higher severity bug reports in the bug tracking system that need to be resolved too.

For the textual factor, we take the description of the input bug report  $BR$  and perform the text pre-processing steps mentioned in Section II. Each of the resultant word token corresponds to a feature. For each feature, we take the number of times it occurs in a description as its value. Collectively these features (TXT1-n) describe what the bug is all about and this determines how important it is for a particular bug to get fixed.

For the author factor, we capture the mean and median priority, and number of all bug reports that are made by the author of  $BR$  prior to the reporting of  $BR$  (AUT1-3). We extract author factor features based on the hypothesis that if an author always reports high priority bugs, he or she might continue reporting high priority bugs. Also, the more bugs an author reports, it is likely that the more reliable his/her severity estimation of the bug would be.

For the related-report factor, we capture the mean and median priority of the top- $k$  reports as measured using  $REP^-$ .  $REP^-$  is a bug report similarity measure adapted from the work by Sun et al. [22] – described in Section II. We vary the value  $k$  to create a number of features (REP1-10). Considering that similar bug reports might be assigned the same priority, we analyze the top- $k$  most similar reports to a bug report  $BR$  to help us decide the priority of  $BR$ . For the severity factor, we use the severity field of  $BR$  as a feature.

For the product factor, we capture features related to the product and component fields of  $BR$ . The product field specifies a part of the software system that is affected by the issue reported in  $BR$ . The component field specifies more specific sub-parts of the software system that are affected by the issue reported in  $BR$ . For each of the product and component fields, we extract 11 features that capture the value of the field (PRO1,PRO12), some statistics of bug reports made for

TABLE III  
DRONE FEATURES EXTRACTED FOR A BUG REPORT  $BR$

Temporal Factor	
TMP1	Number of bugs reported within 7 days before the reporting of $BR$
TMP2	Number of bugs reported with the same severity within 7 days before the reporting of $BR$
TMP3	Number of bugs reported with the same or higher severity within 7 days before the reporting of $BR$
TMP4-6	The same as TMP1-3 except the time duration is 30 days
TMP7-9	The same as TMP1-3 except the time duration is 1 day
TMP10-12	The same as TMP1-3 except the time duration is 3 days
Textual Factor	
TXT1-n	Stemmed words from the description field of $BR$ excluding stop words (Specifically, n=395,996 in our experiment).
Author Factor	
AUT1	Mean priority of all bug reports made by the author of $BR$ prior to the reporting of $BR$
AUT2	Median priority of all bug reports made by the author of $BR$ prior to the reporting of $BR$
AUT3	The number of bug reports made by the author of $BR$ prior to the reporting of $BR$
Related-Report Factor	
REP1	Mean priority of the top-20 most similar bug reports to $BR$ as measured using $REP^-$ prior to the reporting of $BR$
REP2	Median priority of the top-20 most similar bug reports to $BR$ as measured using $REP^-$ prior to the reporting of $BR$
REP3-4	The same as REP1-2 except only the top 10 bug reports are considered
REP5-6	The same as REP1-2 except only the top 5 bug reports are considered
REP7-8	The same as REP1-2 except only the top 3 bug reports are considered
REP9-10	The same as REP1-2 except only the top 1 bug report is considered
Severity Factor	
SEV	$BR$ 's severity field.
Product Factor	
PRO1	$BR$ 's product field. This categorical feature is translated into multiple binary features.
PRO2	Number of bug reports made for the same product as that of $BR$ prior to the reporting of $BR$
PRO3	Number of bug reports made for the same product of the same severity as that of $BR$ prior to the reporting of $BR$
PRO4	Number of bug reports made for the same product of the same or higher severity as those of $BR$ prior to the reporting of $BR$
PRO5	Proportion of bug reports made for the same product as that of $BR$ prior to the reporting of $BR$ that are assigned priority P1.
PRO6-9	The same as PRO5 except they are for priority P2-P5 respectively.
PRO10	Mean priority of bug reports made for the same product as that of $BR$ prior to the reporting of $BR$
PRO11	Median priority of bug reports made for the same product as that of $BR$ prior to the reporting of $BR$
PRO12-22	The same as PRO1-11 except they are for the component field of $BR$ .

that particular product/component prior to the reporting of  $BR$  (PRO2-9,PRO13-20), and the mean and median priority levels of bug reports made for that particular product/component prior to the reporting of  $BR$  (PRO10-11,PRO21-22). Some products or components might play a more major role in the software systems than other products or components – for these products a triager might assign higher priority levels. We extract these 22 `product` features to characterize  $BR$ 's product and component for a better prediction of its priority level.

### C. Classification Module

Feature vectors produced by the feature extraction module for the training and testing data would be fed to the classification module. The classification module has two parts corresponding to the training and prediction phases. In the training phase, the goal is to build a discriminative model that could predict the priority of a new bug report with unknown priority. This model would be used in the prediction phase to assign priority levels to bug reports.

In this work, we propose a classification engine named

GRAY (Thresholding and Linear Regression to Classify Imbalanced Data). We illustrate our classification engine in Figure 2. It has two main parts: linear regression and thresholding. Our approach utilizes linear regression to capture the relationship between the features and the priority levels. As our data is imbalanced (i.e., most of the bug reports are assigned priority level P3), we employ a *thresholding* approach to calibrate a set of thresholds to decide the class labels (i.e., priority levels).

We follow a regression approach rather than a standard classification approach for the following reason. The bug reports are of 5 priority levels (P1-P5). These priority levels are not categorical values rather they are ordinal values. This is so as there is a total ordering among these levels. Level P1 is higher than level P2, which is in turn higher than level P3, and so on. To capture this ordering among levels, we use regression rather than a standard classification approach. Standard classification approaches, e.g., standard support vector machine, naive bayes, logistic regression, etc., consider the class labels to be categorical. Also, many approaches and standard tools only support two class labels: +ve and -ve.

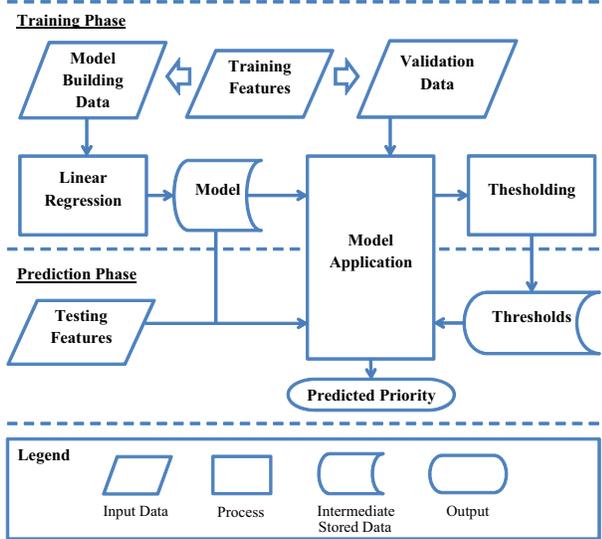


Fig. 2. GRAY Classification Engine

Given a training data, a linear regression approach would build a model capturing the relationship between a set of explanatory variables with a dependent variable. If the set of explanatory variables has more than one member, it is referred to as multiple regression, which is the case for our approach. In our problem setting, the features form the set of explanatory variables while the priority level is the dependent variable. A bug report in the prediction phase would be converted to a vector of features values, which is then treated as a set of explanatory variables. The model learned during linear regression could then be applied to output the value for the dependent variable which is a real number.

The next step is to convert the value of the dependent variable to one of the five priority levels. One possibility is to simply truncate the value of the dependent variable to the nearest integer and treat this as the priority level. However, this would not work well for our data as it is imbalanced with most bug reports having priority 3 – thus many of the values of the dependent variable are likely to be close to 3. To address this issue we employ a *thresholding* approach to pick four thresholds to be the boundaries of the five priority levels.

We split the training data into two (by default, a 50-50 split): model building and validation. The model building training data is used to train a regression model. The linear regression model is then applied on the validation data which generates a linear regression score for each report. The validation training data is used to infer the four thresholds using our thresholding approach. The pseudocode of this process which employs greedy hill climbing to tune the thresholds is shown in Algorithm 1. The resultant linear regression model and thresholds are then used to classify bug reports in the testing data whose priority level is to be predicted based on their feature vectors.

We first set the 4 thresholds based on the proportion of bug reports that are assigned as P1, P2, P3, P4, and P5 in

#### Algorithm 1 Tune Thresholds Using Greedy Hill Climbing

```

1: Input:
2:  $VData$ : Validation Data
3: Output:
4:  $T$ : The four thresholds:  $T_1, T_2, T_3$ , and  $T_4$ 
5: Method:
6: Initialize  $T$  based on the proportion of reports assigned as P1,
   P2, P3, P4, and P5 in  $VData$  (see text).
7: Let  $T_0$  = minimum regression score of reports in  $VData$ .
8: for all  $T_i \in \{T_1, T_2, T_3, T_4\}$  do
9:   Let  $D = T_i - T_{i-1}$ 
10:  repeat
11:    Try to increase  $T_i$  by  $1\% \times D$ , compute new F-measure on
      $VData$ 
12:    Try to decrease  $T_i$  by  $1\% \times D$ , compute new F-measure
     on  $VData$ 
13:    Update  $T_i$  if the increase or decrease improves F-measure
     and  $T_0 < T_1 < T_2 < T_3 < T_4$ 
14:  until  $T_i$  is not updated
15: end for
16: return Tuned thresholds  $T$ 

```

the validation data (Line 6). For example, if the proportion of bug reports belonging to P1 in the validation data is only 10%, then we sort the data points in the validation data based on their linear regression scores, and set the first threshold as the regression output of the data point at the 10th percentile. Next, we modify each thresholds one by one to achieve higher F-measure (Lines 8-15). For each threshold level, we try to increase it or decrease it by a small step, which is 1% of the distance between a threshold level to the previous threshold level (Lines 9, 11-12). At each step, after we change the threshold level, we evaluate if the resultant threshold levels could increase the average F-measure for the validation data points or not. If it is, we will keep the new threshold level otherwise we will discard the new threshold level (Line 13). We continue the process until we can no longer improve the average F-measure by moving a threshold level, with a constraint that a threshold cannot be moved beyond the next threshold level or under the previous threshold level, i.e., the second threshold cannot be set higher than the third threshold (Line 14).

#### IV. EXPERIMENTS & ANALYSIS

In this section, we first describe the datasets that we use to investigate the effectiveness of DRONE. Next, we present our experimental setting and evaluation measures. Finally, we present our research questions followed by our findings.

##### A. Dataset

We investigate the bug repository of Eclipse. Eclipse is an integrated development platform to support various aspects of software development. It is a large open source project that is supported by and used by many developers around the world. We consider the bug reports submitted from October 2001 to December 2007 and download them from Bugzilla<sup>1</sup>. We

<sup>1</sup><https://bugs.eclipse.org/bugs/>

TABLE IV  
DATASET DETAILS

Dataset	Period		REP <sup>-</sup> Training Reports		DRONE Training Reports	Testing Reports
	From	To	#Duplicate	#All	#All	#All
Eclipse	2001-10-10	2007-12-14	200	3,312	87,649	87,648

collect only defect reports and ignore those that correspond to feature requests.

We sort the bug reports in chronological order. We divide the dataset into three: REP<sup>-</sup> training data, DRONE training data, and the test data. The REP<sup>-</sup> training data is the first  $N$  reports containing 200 duplicate bug reports (c.f. [22]). This data is used to train the parameters of REP<sup>-</sup> such that it is better able to distinguish similar bug reports. We split the remaining data into DRONE training and testing data. We use the first half of the bug reports (sorted in chronological order) for training and keep the other half for testing. We separate training data and testing data based on chronological order to simulate the real setting where our approach would be used. This evaluation method is also used in many other research studies that also analyze bug reports [7], [17], [19]. We show the distribution of bug reports used for training and testing in Table IV.

### B. Experimental Setting

We compare our approach with an adapted version of Severis which was proposed by Menzies and Marcus [16]. Severis predicts the severity of bug reports. In the adapted Severis, we simply use it to predict the priority of bug reports. We use the same feature sets and the same classification algorithm described in the Menzies and Marcus’s paper. Following the experimental setting described in their paper, we use the top 100 word token features (in terms of their information gain) as it has been shown to perform best among the other options presented in their paper. We refer to the updated Severis as Severis<sup>Prio</sup>. We also add severity label as an additional feature to Severis<sup>Prio</sup> and refer to the resultant solution Severis<sup>Prio+</sup>. We compare Severis<sup>Prio</sup> and Severis<sup>Prio+</sup> to our proposed framework DRONE. All experiments are run on an Intel Xeon X5675 3.07GHz server, having 128.0GB RAM, and running Windows Server 2008 operating system.

### C. Evaluation Measures

Precision, recall, and F-measure, which are commonly used to measure the accuracy of classification algorithms, are used to evaluate the effectiveness of DRONE and our baseline approaches: Severis<sup>Prio</sup> and Severis<sup>Prio+</sup>. We evaluate the precision, recall, and F-measure for each of the priority levels. This follows the experimental setting of Menzies and Marcus to evaluate Severis [16]. The definitions of precision, recall, and F-measure for a priority level  $P$  are given below:

$$\begin{aligned} \text{prec}(P) &= \frac{\text{Number of priority } P \text{ reports correctly labeled}}{\text{Number of reports labeled as of priority level } P} \\ \text{recall}(P) &= \frac{\text{Number of priority } P \text{ reports correctly labeled}}{\text{Number of priority } P \text{ reports}} \\ \text{F-measure}(P) &= 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \end{aligned}$$

### D. Research Questions

- RQ1 How accurate is our proposed approach as compared with the baseline approaches namely *Severis<sup>Prio</sup>* and *Severis<sup>Prio+</sup>*?
- RQ2 How efficient is our proposed approach as compared with the baseline approaches namely *Severis<sup>Prio</sup>* and *Severis<sup>Prio+</sup>*?
- RQ3 Which of the features are the most effective in discriminating the priority levels?
- RQ4 What are the effectiveness of various classification algorithms in comparison with GRAY in predicting the priority levels of bug reports?

### E. Experimental Results

Here, we present the answers to the four research questions. The first two compare DRONE with Severis<sup>Prio</sup> and Severis<sup>Prio+</sup> on two dimensions: accuracy and efficiency. The best approach must be accurate and yet can complete training and prediction fast. Next, we zoom in to the various factors that influence the effectiveness of DRONE. In particular, we inspect the features that are most discriminative. We also replace the classification module of DRONE with several other classifiers and investigate their effects on the accuracy of the resultant approach.

#### RQ1: Accuracy of DRONE vs. Accuracy of Baselines

The result of DRONE is shown in Table V. We note that we can predict the P1, P2, P3, P4, and P5 priority levels by an F measure of 41.76%, 11.64%, 86.85%, 0.43%, and 8.01% respectively. The F-measures are better for P1, P2, and P3 priority levels but are worse for P4, and P5 priority levels. We believe in report prioritization high accuracy for high priority bugs is much more important than high accuracy for low priority bugs.

TABLE V  
PRECISION, RECALL, AND F-MEASURE FOR DRONE

Priority	Precision	Recall	F-Measure
P1	41.15%	42.39%	41.76%
P2	10.92%	12.46%	11.64%
P3	91.36%	82.77%	86.85%
P4	0.24%	1.77%	0.43%
P5	4.97%	20.72%	8.01%
Average	29.73%	32.02%	29.74%

The result for Severis<sup>Prio</sup> is shown in Table VI. We note that Severis<sup>Prio</sup> can predict the P1, P2, P3, P4, and P5 priority levels by an F-measure of 0.00%, 0.00%, 93.76%, 0.00%, and 0.00% respectively. The F-measures of Severis<sup>Prio</sup> are zeros for P1, P2, P4, and P5 as it does not assign any bug report correctly in any of these priority levels. Comparing these with the result of DRONE (in Table V), we note that we can improve the average of the F measures by a relative

TABLE VI  
PRECISION, RECALL, AND F MEASURE FOR SEVERIS<sup>Prio</sup>

Priority	Precision	Recall	F-Measure
P1	0.00%	0.00%	0.00%
P2	0.00%	0.00%	0.00%
P3	88.25%	100.00%	93.76%
P4	0.00%	0.00%	0.00%
P5	0.00%	0.00%	0.00%
Average	17.65%	20.00%	18.75%

TABLE VII  
PRECISION, RECALL, AND F MEASURE FOR SEVERIS<sup>Prio+</sup>

Priority	Precision	Recall	F-Measure
P1	0.00%	0.00%	0.00%
P2	0.00%	0.00%	0.00%
P3	88.25%	100.00%	93.76%
P4	0.00%	0.00%	0.00%
P5	0.00%	0.00%	0.00%
Average	17.65%	20.00%	18.75%

improvement of 58.61% (i.e.,  $(29.74 - 18.75)/18.75 \times 100\%$ ). Thus, clearly DRONE performs better than Severis<sup>Prio</sup>.

The result for Severis<sup>Prio+</sup> is shown in Table VII. We note that the result of Severis<sup>Prio+</sup> is the same as Severis<sup>Prio</sup>. Thus, our proposed approach DRONE also outperforms Severis<sup>Prio+</sup>.

### RQ2: Efficiency of DRONE vs. Efficiency of Baselines

We compare the runtime of DRONE with those of Severis<sup>Prio</sup> and Severis<sup>Prio+</sup>. The result is shown in Table VIII. The four columns refer to the average feature extraction time (for training data), the model building time, the average feature extraction time (for testing data), and the average model application time. We could note that the time for feature extraction is slower for DRONE than the two variants of Severis. This is the case as DRONE utilizes more features than the two variants of Severis. Severis<sup>Prio</sup> only utilizes the textual features of bug reports. Severis<sup>Prio+</sup> only utilizes the textual and severity features of bug reports. The time for model building however is faster for DRONE than the two variants of Severis. We compare the efficiency of the approaches since the required running time determines the usability of the system for triagers.

TABLE VIII  
EFFICIENCY OF SEVERIS<sup>Prio</sup>, SEVERIS<sup>Prio+</sup>, AND DRONE. FE = AVERAGE FEATURE EXTRACTION TIME. MB = MODEL BUILDING TIME. MA = AVERAGE MODEL APPLICATION TIME.

Approach	Time (in seconds)			
	FE (Train)	MB	FE (Test)	MA
Severis <sup>Prio</sup>	<0.01	812.18	<0.01	<0.01
Severis <sup>Prio+</sup>	<0.01	773.62	<0.01	<0.01
DRONE	0.01	69.25	0.02	<0.01

### RQ3: Most Discriminative Features

Next, we would like to find the most discriminative features among the 20,000+ features that we have (including the word tokens). Information gain [15] and Fisher score [5] are often used as discriminativeness measures. Since many of the features are non-binary features, we use Fisher score as it captures the differences in the distribution of the feature values

TABLE IX  
TOP-10 FEATURES IN TERMS OF FISHER SCORE

Rank	Feature Name	Fisher Score
1	PRO5	0.142
2	PRO16	0.132
3	REP1	0.109
4	REP3	0.101
5	PRO18	0.092
6	PRO10	0.091
7	PRO21	0.088
8	PRO7	0.088
9	REP5	0.087
10	"1663"	0.079

across the classes (i.e., the priority levels).

At times features that are only exhibited in a few data instances receive high Fisher score. This is true for the word tokens. However, these are not good features as they appear too sparsely in the data. Thus we focus on features that appear in at least 0.5% of the data. For these features, Table IX shows the top-10 features sorted according to their Fisher score (the higher the better). We notice that six of them are features related to `product` factor and three of them are features related to `related-report` factor. It suggests that the product a bug report is about and existing related reports influence the priority label assigned to the report.

We notice that the 10<sup>th</sup> most discriminative feature is a word token "1663". This token comes from a line in various stack traces included in many bug reports which is:  
`org.eclipse.ui.internal.Workbench.run(Workbench.java:1663)`

It is discriminative as it appears in 15% of the bug reports assigned priority level P5, while it only appears in 0.77%, 1.29%, 0.99%, and 0.00% of the bug reports assigned priority level P1, P2, P3, and P4 respectively. It seems the inclusion of stack traces that include the above line enables developers to identify P5 bugs better.

### RQ4: Effectiveness of Various Classification Algorithms

The classification engine of our DRONE framework could be replaced with other classification algorithms aside from GRAY. We experiment with several classification algorithms (SVM-MultiClass [4], RIPPER [3], and Naive Bayes Multinomial [15]) and compare their F-measures across the five priority levels with GRAY. We use the implementation of SVM-MultiClass available from [24]. We use the implementations of RIPPER and Naive Bayes Multinomial in WEKA [1]. We show the result in Table X. We notice that in terms of average F-Measures GRAY outperforms SVM-MultiClass by a relative improvement of 58.61%. Naive Bayes Multinomial is unable to complete due to an out-of-memory exception although we have allocated more than 9GB of RAM to the JVM in our server. RIPPER could not complete after running for more than 8 hours.

## V. DISCUSSION

In this section, we first present the threats to validity. Next, we explain the reasons that might cause the bad performance of the baseline approaches. Finally, we compare our model

TABLE X  
COMPARISONS OF AVERAGE F-MEASURES OF GRAY VERSUS OTHER CLASSIFIERS. CLASS. = CLASSIFIERS. SM = SVM-MULTICLASS. NBM = NAIVE BAYES MULTINOMIAL. OOM = OUT-OF-MEMORY (MORE THAN 9GB). CC = CANNOT COMPLETE IN TIME (MORE THAN 8 HOURS).

Class.	F-Measures					
	P1	P2	P3	P4	P5	Ave.
GRAY	41.76%	11.64%	86.85%	0.43%	8.01%	29.74%
SM	0%	0%	93.76%	0%	0%	18.75%
RIPPER	CC	CC	CC	CC	CC	CC
NBM	OOM	OOM	OOM	OOM	OOM	OOM

with a trivial approach that simply assigns a bug report to the most common priority level.

#### A. Threats to Validity

Threats to construct validity relates to the suitability of our evaluation measures. We use precision, recall, and F-measure which are standard metrics used for evaluating classification algorithms. Also, these measures are used by Menzies and Marcus to evaluate Severis [16]. Threats to internal validity relates to experimental errors. We have checked our implementation and results. Still, there could be some errors that we did not notice. Threats of external validity refers to the generalizability of our findings. We consider the repository of Eclipse containing more than a hundred thousand bugs which are reported in a period of more than 6 years. Still, we have only analyzed bug reports from one software system. We exclude some other Bugzilla datasets from two other software systems that we have as most of the reports there do not contain information on the priority field. In the future, we plan to extend our study by considering more programs and bug reports.

#### B. Dismal Performance of Baseline

Results of  $Severis^{Prio}$  and  $Severis^{Prio+}$  shown in Table VI and Table VII are poor. This might be due to a few factors:

- 1) **(Treating Ordinal Data as Categorical Data)** The RIPPER classification algorithm used by Severis considers the class labels as categorical data. RIPPER, as well as other standard classification algorithms (e.g., SVM, etc), does not consider how different a pair of class labels is as compared to other pairs of class labels. In our setting, RIPPER simply treats P1, P2, P3, P4, and P5 as different labels. P1 is as different to P2, as P1 to P5. We know that this is not the case. Bug reports labeled as P1 are likely to be more similar to those labeled as P2, than those labeled as P5.

Our approach that enhances linear regression treats class labels as ordinal data. Thus, in our setting P1 is closer to P2 than it is to P5.

- 2) **(Data Imbalance)** Data imbalance might also negatively affect the performance of the baseline approaches. There are much more bug reports assigned as P3 (74,210 out of 87,649) in the training data; this might make the classifier “thinks” that the best label to assign to *any* bug report is P3. We solve this problem by a *thresholding* approach that varies the ranges of regression output

values, corresponding to each class labels, based on a set of validation data points.

#### C. Comparison with a Trivial Approach

Besides comparing our approach and those adapted from a previous work, we also compare our approach with a trivial approach. The trivial approach simply labels every bug as P3 based on the fact that nearly 90% of the bug reports are assigned P3 priority level. The result of this trivial approach is the same as those of  $Severis^{Prio}$  and  $Severis^{Prio+}$  shown in Tables VI and VII respectively. This means that this trivial approach can predict the P1, P2, P3, P4 and P5 priority levels by an F-measure of 0.00%, 0.00%, 93.76%, 0.00% and 0.00% respectively, with an average F-measure of 18.75%. Thus our model can better predict the priority of bug reports. In particular, our approach performs much better in predicting high priority bug reports that are more important than lower priority ones.

## VI. RELATED WORK

Menzies and Marcus are the first to predict the severity of bug reports [16]. They analyze the severity labels of various bugs reported in NASA. They propose a technique that analyzes the textual contents of bug reports and outputs *fine-grained* severity levels – one of the 5 severity labels used in NASA. Their approach extracts word tokens from the description of the bug reports. These word tokens are then pre-processed by removing stop words and performing stemming. Important word tokens are then selected based on their information gain. Top-k tokens are then used as features to characterize each bug report. The set of feature vectors from the training data is then fed into a classification algorithm named RIPPER [3]. RIPPER would learn a set of rules which are then used to classify future bug reports with unknown severity labels.

Lamkanfi et al. extend the work by Menzies and Marcus to predict severity levels of reports in open source bug repositories [13]. Their technique predicts if a bug report is severe or not. Bugzilla has six severity labels including `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. They drop bug reports belonging to the category `normal`. The remaining five categories are grouped into two groups – severe and non-severe. Severe group includes `blocker`, `critical` and `major`. Non-severe group includes `minor` and `trivial`. Thus they focus on the prediction of *coarse-grained* severity labels.

Extending their prior work, Lamkanfi et al. also try out various classification algorithms and investigate their effectiveness in predicting the severity of bug reports [14]. They tried a number of classifiers including Naive Bayes, Naive Bayes Multinomial, 1-Nearest Neighbor, and SVM. They find that Naive Bayes Multinomial perform the best among the four algorithms on a dataset consisting of 29,204 bug reports.

Recently, Tian et al. also predict the severity of bug reports by utilizing a nearest neighbor approach to predict fine grained bug report labels [27]. Different from the work by Menzies and

Marcus which analyzes a collection of bug reports in NASA, Tian et al. apply the solution on a larger collection of bug reports consisting of more than 65,000 Bugzilla reports.

Our work is orthogonal to the above studies. Severity labels are reported by users, while priority levels are assigned by developers. Severity labels correspond to the impact of the bug on the software system as perceived by users while priority levels correspond to the importance “a developer places on fixing the bug” in the view of other bug reports that are received [20].

Khomh et al. automatically assign priorities to Firefox crash reports in Mozilla Socorro server based on the frequency and entropy of the crashes [11]. A crash report is automatically submitted to the Socorro server when Firefox fails and it contains a stack trace and information about the environment to help developers debug the crash. In our study, we investigate bug reports that are manually submitted by users. Different from a crash report, a bug report contains natural language descriptions of a bug and might not contain any stack trace or environment information. Thus, different from Khomh et al.’s approach, we employ a text mining based solution to assign priorities to bug reports.

There are other lines of work that also analyze bug reports; these include the series of work on duplicate bug report detection [19], [23], [22], [28], bug localization [31], bug categorization [6], [8], [26], bug fix time prediction [12], [30], and bug fixer recommendation [10], [25]. Our work is also orthogonal to these studies.

## VII. CONCLUSION AND FUTURE WORK

In this work, we propose a framework named DRONE (PreDICTing PRiority via Multi-Faceted FactOrs ANalysEs) to predict the priority levels of bug reports in Bugzilla. We consider multiple factors including: temporal, textual, author, related-report, severity and product. These features are then fed to a classification engine named GRAY (ThresholdinG and Linear Regression to CIAssify Imbalanced Data) built by combining linear regression with a thresholding approach to address the issue with imbalanced data and to assign priority labels to bug reports. We have compared our approach with several baselines based on the state-of-the-art study on bug severity prediction by Menzies and Marcus [16]. The result on a dataset consisting of more than 100,000 bug reports from Eclipse shows that our approach outperforms the baselines in terms of average F-measure by a relative improvement of 58.61%.

In the future, we plan to include more bug reports from more open source projects to experiment with. We also plan to further improve the accuracy of our approach. For instance, we can try to construct a linear regression model using only the most discriminative features and evaluate the resulting solution. We also plan to analyze the impact of inaccuracies in the thresholding process on the final result of DRONE.

## REFERENCES

[1] “<http://www.cs.waikato.ac.nz/ml/weka/>,” Weka 3: Data Mining Software.

- [2] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *ETX*, 2005.
- [3] W. W. Cohen, “Fast effective rule induction,” in *ICML*, 1995.
- [4] K. Crammer and Y. Singer, “On the algorithmic implementation of multiclass kernel-based vector machines,” *Journal of Machine Learning Research*, vol. 2, 2001.
- [5] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley Interscience, 2000.
- [6] M. Gegick, P. Rotella, and T. Xie, “Identifying security bug reports via text mining: An industrial case study,” in *MSR*, 2010.
- [7] L. Hiew, “Assisted detection of duplicate bug reports,” Ph.D. dissertation, The University Of British Columbia, 2006.
- [8] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, “AutoODC: Automated generation of orthogonal defect classifications,” in *ASE*, 2011.
- [9] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *DSN*, 2008.
- [10] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in *ESEC/SIGSOFT FSE*, 2009.
- [11] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, “An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox,” in *WCRE*, 2011.
- [12] S. Kim and E. J. W. Jr., “How long did it take to fix bugs?” in *MSR*, 2006.
- [13] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *MSR*, 2010.
- [14] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, “Comparing mining algorithms for predicting the severity of a reported bug,” in *CSMR*, 2011.
- [15] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, 2008.
- [16] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” in *ICSM*, 2008.
- [17] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, “Duplicate bug report detection with a combination of information retrieval and topic modeling,” in *ASE*, 2012.
- [18] S. Robertson, H. Zaragoza, and M. Taylor, “Simple BM25 Extension to Multiple Weighted Fields,” in *CIKM*, 2004.
- [19] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *ICSE*, 2007.
- [20] [http://wiki.eclipse.org/Bug\\_Reporting\\_FAQ#What\\_is\\_the\\_difference\\_between\\_Severity\\_and\\_Priority.3F](http://wiki.eclipse.org/Bug_Reporting_FAQ#What_is_the_difference_between_Severity_and_Priority.3F).
- [21] [www.ils.unc.edu/~keyeg/java/porter/PorterStemmer.java](http://www.ils.unc.edu/~keyeg/java/porter/PorterStemmer.java).
- [22] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, “Towards more accurate retrieval of duplicate bug reports,” in *ASE*, 2011.
- [23] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” in *ICSE*, 2010.
- [24] [http://svmlight.joachims.org/svm\\_multiclass.html](http://svmlight.joachims.org/svm_multiclass.html).
- [25] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, “Fuzzy set-based automatic bug triaging,” in *ICSE*, 2011.
- [26] F. Thung, D. Lo, and L. Jiang, “Automatic defect categorization,” in *WCRE*, 2012.
- [27] Y. Tian, D. Lo, and C. Sun, “Information retrieval based nearest neighbor classification for fine-grained bug severity prediction,” in *WCRE*, 2012.
- [28] Y. Tian, C. Sun, and D. Lo, “Improved duplicate bug report identification,” in *CSMR*, 2012.
- [29] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *ICSE*, 2008.
- [30] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” in *MSR*, 2007.
- [31] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports,” in *ICSE*, 2012.

# An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks

Gabriele Bavota<sup>1</sup>, Gerardo Canfora<sup>1</sup>, Massimiliano Di Penta<sup>1</sup>, Rocco Oliveto<sup>2</sup>, Sebastiano Panichella<sup>1</sup>

<sup>1</sup>University of Sannio, Benevento, Italy

<sup>2</sup>University of Molise, Pesche (IS), Italy

{gbavota, canfora, dipenta}@unisannio.it, rocco.oliveto@unimol.it, spanichella@unisannio.it

**Abstract**—When developers perform a software maintenance task, they need to identify artifacts—e.g., classes or more specifically methods—that need to be modified. To this aim, they can browse various kind of artifacts, for example use case descriptions, UML diagrams, or source code. This paper reports the results of a study—conducted with 33 participants—aimed at investigating (i) to what extent developers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts. Results indicate that, although participants spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, participants—especially more experienced ones—follow an “integrated” approach by using different kinds of artifacts.

## I. INTRODUCTION

Maintenance tasks are generally facilitated when software documentation (e.g., the requirements specification, design document, test report, and user manual) is available [2], [9], [28]. Indeed, having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20% [28]. In addition, besides time reduction, documentation allows developers to find better and more accurate technical solutions to a given maintenance task [28].

Although several studies have shown the usefulness of documentation during maintenance tasks (see e.g., [2], [6], [9], [17], [26], [28]), it is still unclear how such documentation is browsed by developers to understand how the system should be modified to implement a specific change. At one extreme, one can argue for using all the available documentation, as each artifact is equally useful, since it provides a description of the system with different levels of details. Also, the documentation could be browsed starting from high-level artifacts (e.g., use cases) to low level artifacts (e.g., dynamic models). Even if there is an anecdotal evidence that such an approach could work, without a proper empirical investigation it remains only a conjecture. Also, different developers—with different skills and experience—might follow different paths. Thus, on one hand, guessing a priori navigational paths is quite challenging. On the other hand, understanding such paths is relevant not only to highlight the importance of high-level documentation, but also to help tool developers enhancing modelers and Integrated Development Environments (IDEs) to better support program comprehension activities by facilitating effective and efficient artifact navigation and browsing.

All these considerations motivate our work. We conducted a study, involving 33 participants—among undergraduate and

graduate students from different universities—aimed at analyzing to what extent developers use different kinds of documentation when identifying pieces of code (e.g., methods) to be changed and whether they follow specific navigation paths among different kinds of artifacts. In the context of our study, we asked participants to perform 8 different maintenance tasks on a Java software system. Besides source code, participants had available use case descriptions, sequence diagrams, class diagrams, and Javadoc. We used an Eclipse plugin to capture how much time was spent by participants on different artifacts, and how they navigated from an artifact to another.

The obtained results indicated that—even if a substantial proportion of time (about 80% on average) is spent on source code, participants also browsed back and forth between source code and either static (class) or dynamic (sequence) diagrams, the latter being more used than the former. Less frequently, participants—and in particular those with a higher degree of experience, i.e., graduate students—follow an “integrated” approach, in which different kinds of artifacts were used, for example starting the task from use cases, then browsing sequence and/or class diagrams before accessing the source code. Such results could be used to enhance IDEs with a recommendation system able to suggest a particular navigation path aiming at facilitating the browsing of the available documentation. Such a recommender might be particularly useful in large systems where the browsing of myriad software artifacts could represent an obstacle instead of a facilitation when performing the maintenance task [7], [14].

**Paper organization.** Section II presents the definition and planning of our study, while Section III discusses the results achieved. Section IV presents the threats that could affect the validity of our study. Finally, after a discussion of the related literature (Section V), Section VI concludes the paper outlining direction for future work.

## II. STUDY DEFINITION AND PLANNING

This section describes the design and planning of our empirical study. The *goal* of the study is to observe how developers browse different kinds of software artifacts, with the *purpose* of understanding how they build knowledge needed to deal with a maintenance task and, specifically, to identify classes and class elements (methods and attributes) that need to be changed when performing a maintenance task. The *perspective* is of researchers interested to identify relevant navigation paths across artifacts that result helpful during a software evolution task. This result can be used, for example, to build smart recommenders that guide developers by suggesting navigations across artifacts or to better organize and index the documentation available for a software project.

### A. Context Selection

The study involved 33 participants, selected entirely on a voluntary basis—i.e., using a convenience sampling—mainly among undergraduate students of the Computer Science Degree at the University of Molise, and among master students, PhD students (including visiting students) of the Computer Science Engineering Degree of the University of Sannio. Overall, 11 Bachelor students, 18 Master students, and 4 PhD students participated to the study. Master and PhD students had already experience on some industrial or research projects, as well as on the development and maintenance of complex software systems.

The objects which the tasks were performed on are use case descriptions, design level sequence and class diagrams, Javadoc, and Java source code files of a school automation system, named SMOS, developed by graduate students at the University of Salerno (Italy). SMOS offers a set of features aimed at simplifying the communication between the school and the student's parents. The system is composed of 121 classes with their respective Javadoc for a total size of 23 KLOC. The documentation is represented by 67 use cases, 72 design level sequence diagrams, and 6 design level class diagrams. Each class diagram represents the relationships between all the classes involved in a specific subsystem, e.g., teaching management. On average, each use case describes 4 interactions between the actor and the system, each sequence diagram reports 10 interactions between the actor and the system's code components, and each class diagram depicts 15 classes and their dependencies.

In the context of our study, we asked participants to perform 8 different maintenance tasks on SMOS, of which 3 were bug-fixing tasks, 3 related to add a new feature, and 2 related to improve existing features, i.e., performing a perfective maintenance task. On average, each maintenance task impacted 5 code components (with a minimum of 1 and a maximum of 15).

### B. Research Questions

The study aims at investigating the following research questions:

- **RQ<sub>1</sub>**: *How much time did participants spend on different kinds of artifacts?* This research question aims at analyzing the time spent by participants on the different kinds of available artifacts. On the one hand, artifacts used for less time can be thought of being less useful. On the other hand, some artifacts intrinsically require more time to be read (e.g., source code) while for others (e.g., use cases, sequence diagrams) a quick look may just suffice to provide a useful piece of information.
- **RQ<sub>2</sub>**: *How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?* This research question is the core of our study aimed at analyzing the sequences of interactions made with different artifacts. In particular, we will investigate (i) how do participants start the task, (ii) what kinds of artifacts do they browse before getting to the source code, and (iii) whether there are frequent browsing patterns, e.g., repeated navigation back and forth between source code and class diagrams.

TASK DESCRIPTION
In SMOS a registered user can have six different roles: Admin, Teacher, Student, Parent, Janitors, and Director. Suppose that we want to remove the "Director" role, which changes do you need to made on source code? Specify for each involved class/method the changes you would apply.
QUESTIONS
Write the list of methods modified to perform this task specifying how you modified these methods. For example: "application.userManagement.UpdateUser.doGet". I added the line of code "x=3;" after the line of code "y++;".
Write the list of attributes modified to perform this task. For example: "bean.User.UID".

Fig. 1. Example of task description and related questions.

For each research question, we also analyzed the impact of participants' experience on the use and the navigation of the software documentation.

### C. Study Procedure and Material

Before the study, we explained to participants what we expected them to do during their tasks. Specifically, we asked them to identify methods and attributes to be changed when performing each change task. We provided an overview of what kinds of artifacts they have available, briefly summarizing the purpose of each of them.

After illustrating the study, we gave participants up to 3 hours of time to perform the task. Note that it was not our intention to measure the task efficiency, hence we were not strict with the time. We only made sure participants properly performed the task, without collaborating.

We provided each participants with a customized Eclipse installation containing:

- The Java Development Environment (JDT) with the SMOS software system already imported together with its documentation, i.e., sequence diagrams, class diagrams, use cases, and Javadoc.
- FLUORITE<sup>1</sup> (Full of Low-level User Operations Recorded In The Editor), an Eclipse plug-in able to capture all of the low-level events when using the Eclipse editors. FLUORITE keeps track of all of the events that occur in the Eclipse editors also storing timestamps for each event. All data is saved in an XML log file.
- The Pdf4Eclipse<sup>2</sup> plug-in (used to visualize use cases, sequence diagrams, and class diagrams).
- An Eclipse HTML Editor<sup>3</sup> plug-in, used to visualize the Javadoc files.

Also, we provided participants with an URL of a page on ESurveysPro<sup>4</sup>, a online survey tool we used to collect participants' answers.

<sup>1</sup><http://www.cs.cmu.edu/fluorite/>

<sup>2</sup><http://borisvl.github.io/Pdf4Eclipse/>

<sup>3</sup>[http://amateras.sourceforge.jp/cgi-bin/fswiki\\_en/wiki.cgi?page=EclipseHTMLEditor](http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=EclipseHTMLEditor)

<sup>4</sup><http://www.esurveyspro.com/>

During the study, we instructed participants to access the ESurveysPro page and, for each of the eight tasks to be performed, to work following this procedure:

- 1) Access the page describing the task, and read the task description.
- 2) Then, use Eclipse to find a solution for the task (without however applying the change).
- 3) Answer the questions in the opened ESurveysPro page. For each task, participants had to provide, using two different form fields, the list of methods and instance variables (attributes) that need to be modified. Fig. 1 shows an example of task description and questions being asked for the task. For each question examples of answers are provided. We made clear to participants that example answers are not related to the task, thus they are not valid answers.

After having completed the 8 tasks, participants had to fill a post-study questionnaire. The post-study questionnaire asked participants an opinion about the usefulness of the various kinds of artifacts, using a Likert scale [16] ranging between 1 (totally useless) and 5 (very useful). We also asked participants to provide a comment for the rank assigned to each kind of artifact.

#### D. Data Collection

After tasks were completed, we collected from each participant (i) the XML logs generated by FLUORITE; and (ii) the answers provided on ESurveysPro. Concerning FLUORITE logs, they have been parsed through a Java tool developed on purpose. The tool extracts, for each task performed by each participant, the ranked list of documents explored during such a task together with the time spent on each document. An example of generated list is:

*UseCase(27) → SequenceDiagram(48) → Code(82)*

indicating that the participant started by reading an use case description for 27 seconds, moving then to a sequence diagram for 48 seconds, and finally access the source code for 82 seconds.

We pruned out from such logs browsing activities shorter than 5 seconds. Such a threshold was set by observing how subjects navigated source code during the tasks. Although this would remove some potentially useful information, we assume that such short activities are mainly due to the need for scrolling across various windows in the IDE.

#### E. Analysis Method

To answer RQ<sub>1</sub>, we measure (in seconds) the time spent by participants on each of the artifact types considered in our study (i.e., the four different documentations plus source code). We also analyze the scores provided by the participants in the post-survey questionnaire to indicate their perceived usefulness of the exploited artifacts. Results are reported in terms of descriptive statistics and boxplots.

Besides analyzing the whole dataset collected during our study, we investigate whether participants with different levels of experience (graduate vs. undergraduate students) use artifacts differently. Due to the limited number of PhD students,

and also for the sake of simplicity, we just distinguish between undergraduate (i.e., bachelor) and graduate (i.e., Master or PhD) students. The main reason why we analyzed results of graduate and undergraduate students separately is because the former had (i) some real working experience, and (ii) in all cases, some experience in the development and maintenance of complex projects, which would often favor the need for using high-level documentation when performing a comprehension task.

In addition to descriptive statistics and boxplots, we use Mann-Whitney test [5] to compare the proportion of time spent on each kind of diagram by participants having different levels of experience. We use a non-parametric test because the Shapiro-Wilk normality test indicated that data—related to all kinds of artifacts for both undergraduate and graduate—deviate from a normal distribution (p-value < 0.001 in all cases). We also evaluate the magnitude of the observed differences using the Cliff’s Delta (or  $d$ ), a non-parametric effect size measure [11] for ordinal data. We followed the guidelines in [11] to interpret the effect size values: small for  $d < 0.33$  (positive as well as negative values), medium for  $0.33 \leq d < 0.474$  and large for  $d \geq 0.474$ .

Still in the context of RQ<sub>1</sub>, we verify if there is a correlation between the kind of artifacts exploited by participants and the correctness of the performed tasks. Note that our study does not aim at investigating whether the usage of different artifacts influences the task correctness. This cannot be done, because it would have required a specific controlled experiment with participants receiving different treatments, e.g., using some diagrams only, or “forced” to follow specific navigational paths only. Instead, this analysis should be considered as a form of sanity-check, to determine whether participants performed tasks seriously and whether participants using more specific kinds of artifacts could have suffered particular problems.

To measure the completeness and correctness of the tasks performed by each participant (i.e., her ability in correctly individuating the code components impacted by a maintenance activity), we used a combination of two well-known Information Retrieval metrics, recall and precision [3]. Recall measures the percentage of code components actually impacted by a maintenance activity correctly identified by a participant, while precision measures the percentage of identified components that are actually impacted. Since recall and precision measure two different (but related) concepts, we use their harmonic mean (i.e., F-measure [3]) to obtain a balance between them when measuring task correctness.

The correlation between the type of artifacts exploited by participants and the correctness and completeness of the performed tasks is computed through (i) the Spearman correlation, performed between the time spent by participants in each task on each type of artifact and the correctness achieved in the task, and (ii) by building a logistic regression model for correctness based on the use (or not) of different kinds of artifacts.

Concerning RQ<sub>2</sub> we extracted, using the data derived by the FLUORITE plugin, information concerning how participants navigate different artifacts, and specifically:

- What artifacts did participants looked first, i.e., where the comprehension task started. Usually, one assumes this

TABLE I. RECALL, PRECISION, AND F-MEASURE ACHIEVED BY PARTICIPANTS WHEN PERFORMING THE TASKS.

Dataset		Recall	Precision	F-measure
Undergraduates	Mean	0.65	0.79	0.71
	Median	0.81	1.00	0.82
	St. Dev.	0.40	0.38	0.37
Graduates	Mean	0.67	0.88	0.76
	Median	0.88	1.00	0.93
	St. Dev.	0.37	0.31	0.35
All	Mean	0.67	0.85	0.75
	Median	0.88	1.00	0.86
	St. Dev.	0.38	0.34	0.36

starts from requirements/use cases, although there are developers that start from source code directly.

- What artifacts did participants browse before getting to source code. This could potentially indicate the pattern followed to locate the source code element to be changed.
- What is the likelihood of making a transition from one kind of artifact to the other. This can likely indicate how the information gained by browsing a certain kind of artifact raises the need for accessing another kind of artifact, e.g., browsing source code after accessing sequence or class diagrams, or else looking at static models after dynamic models.
- What are the most frequently followed patterns. This was done by matching regular expressions of length varying from two to four onto the mined logs, and determining for each pattern whether it was iterated, e.g., participants could go back and forth between source code and class or sequence diagrams repeatedly.

Finally, we investigated whether participants with different levels of experience followed different patterns and whether following certain patterns can influence the task correctness.

All statistical analyses of this paper have been performed using the *R* environment [18]. For all statistical procedures, we assumed a significance level of 95%.

#### F. Replication package

To facilitate the replication of this study, a complete replication package is available<sup>5</sup>. It includes (i) an Eclipse installation bundle, with all the exploited plug-ins installed and the object system SMOS (source code and other artifacts) already imported, (ii) the task description for all 8 tasks, (iii) the post-study questionnaire, and (iv) the FLUORITE logs for the 33 participants. Also, the package includes the working data set with our study results.

### III. ANALYSIS OF THE RESULTS

Before answering the research questions formulated in Section II-B, it is important to verify whether participants seriously performed the assigned tasks. To this aim, Table I reports the average values for recall, precision, and F-measure achieved by undergraduate and graduate students, as well as when considering the entire dataset. Results show that participants were able to achieve quite good performances, with an average F-measure of 0.75. This sanity check makes us confident that participants seriously performed the assigned tasks. Also, as expected, graduate students achieved, on average, better

<sup>5</sup><http://distat.unimol.it/reports/icsm-docs/>

TABLE II. USE (PERCENTAGE OF TASKS AND TIME SPENT) OF DIFFERENT KINDS OF ARTIFACTS: DESCRIPTIVE STATISTICS.

Artifacts	Tasks (%) (All)	Tasks (%) Undergrad.	Tasks (%) Graduate	Time spent (all data, %)			
				mean	IQ	median	3Q
Use case	33	28	36	3	0	0	2
Sequence Diagram	72	68	74	10	0	7	16
Class Diagram	60	49	66	13	0	4	15
Javadoc	15	21	11	2	0	0	0
Source Code	100	100	100	72	66	79	89

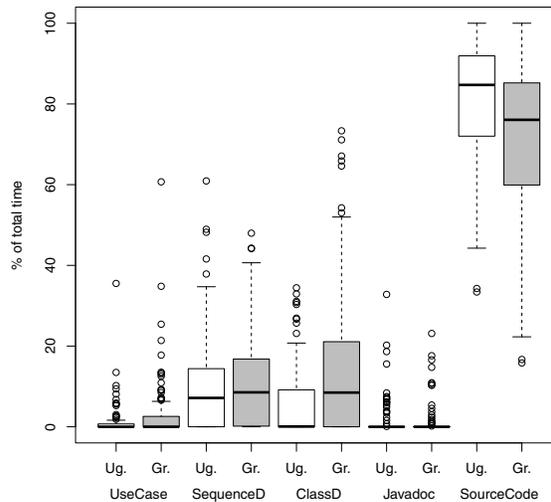


Fig. 2. Usage (in percentage) of different kinds of artifacts. Ug = undergraduate students, Gr = graduate students.

performances than undergraduate students (+5% in terms of F-measure).

#### A. RQ1: How much time did participants spend on different kinds of artifacts?

Table II reports the percentage of tasks in which each kind of artifact has been used (for the entire dataset as well as by separately considering participants with different levels of experience), and descriptive statistics about the percentage of time spent on various kinds of artifacts (by considering the entire dataset). Fig. 2 shows boxplots of such percentage for different levels of experience.

If considering the whole dataset, and analyze the time spent on artifacts (right-side of Table II), results indicate that participants spent most of their time (72% on average) on source code. Our conjecture—partially supported by what we observed during the tasks and by talking with participants—is that this might be due to two reasons. First, even when participants were able to identify the impacted components by analyzing documentation artifacts we observed that they checked-back in the source code that the identified methods/attributes were actually there and really impacted by the maintenance activity to perform. This suggests a kind of distrust with respect to documentation artifacts, as also confirmed by the fact that source code has been used in 100% of the tasks. Second, source code clearly requires more time to be read and understood as compared to the artifacts present in the documentation. In particular, participants spent, on average, 154 seconds on each source code file, compared to the 70 spent on a class diagram, 49 on a Javadoc file, 35 on a sequence diagram, and 34 on a use case.

TABLE III. PERCENTAGE OF TIME SPENT ON ARTIFACTS BY PARTICIPANTS WITH DIFFERENT EXPERIENCE: MANN-WHITNEY TEST AND CLIFF'S  $d$  EFFECT SIZE (POSITIVE VALUES INDICATE DIFFERENCES IN FAVOR OF GRADUATE STUDENTS, NEGATIVE OF UNDERGRADUATES).

Artifact	p-value	Cliff's $d$
Use Case	0.1020	0.1030
Sequence Diagram	0.3102	0.0749
Class Diagram	<b>0.0001</b>	0.2757
Javadoc	0.0268	-0.1040
Source Code	< <b>0.0001</b>	-0.2939

If we look at the percentage of tasks in which each kind of artifact was used at least once (left-side of Table II), we notice that—besides source code, obviously used in 100% of the tasks—the most commonly used documentation artifacts are class and sequence diagrams. The latter were used in 72% of the task. On such diagrams, participants spent on average 10% of their time (median=7%). Only one of the 33 participants did not exploit at all sequence diagrams during the tasks and justified such a choice in the post-study questionnaire: “*sequence diagrams would be useful only if class diagrams were not present*”. However, as we will see shortly, this is an isolate point-of-view.

As for class diagrams, they were used in 60% of tasks and participants spent, on average, 15% of their time on them (median=4%). This strong misalignment between the mean and the median values for class diagrams highlights that, while generally they are used for a lower proportion of time as compared with sequence diagrams, some participants spent a very high proportion of their time on class diagrams, as also shown by the outliers reported in Fig. 2. Two participants did not use at all class diagrams in the tasks.

Turning to use cases, they were used in 33% of tasks by participants, which focused on them just the 3% of their time, on average. As said before, participants spent just 34 seconds, on average, on each consulted use case against, for instance, the 154 spent on each source code file. Among the 33 participants, three of them did not access at all use cases.

Finally, Javadoc documentation was not used a lot by participants of our study. They accessed Javadoc in just 15% of the tasks. Also, 11 participants out of 33 never open Javadoc files during the tasks.

Concerning the time spent by participants with different experience levels on different artifacts, Fig. 2 and the results of the Mann-Whitney test reported in Table III indicate that: (i) there is no significant difference in accessing use cases and sequence diagrams; (ii) graduate students use class diagrams significantly more than undergraduates, with a medium effect size; (ii) undergraduates students used source code and Javadoc significantly more than graduate students, with a small and medium effect size respectively. Such results partially contradict those of other studies [19], which indicated that junior developers tend to benefit of models than senior developers, that tend to directly focus onto source code.

To better understand the results of the quantitative analysis, we analyzed the feedbacks provided us by means of the post-study questionnaire. Fig. 3 shows boxplots—for different levels of experience—of the ratings provided by participants to the usefulness of the different kinds of artifacts. As explained in Section II-C, one corresponds to classify a kind of artifact

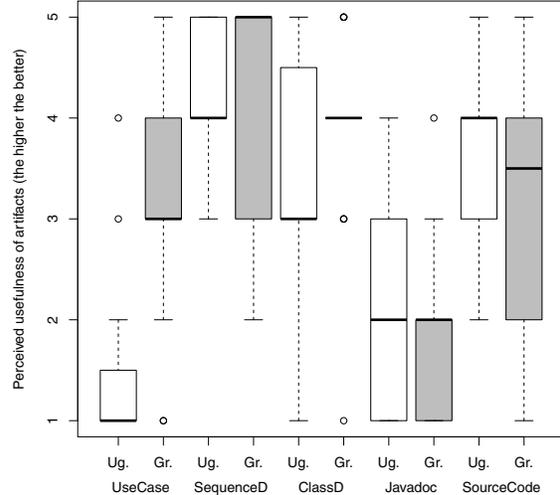


Fig. 3. Perceived usefulness of the different kinds of artifacts as indicated by participants. Ug = undergraduate students, Gr = graduate students.

(documentation as well as source code) as “totally useless”, while five indicates a “very useful” kind of artifact.

As we can notice, sequence diagrams are considered to be the most useful kinds of artifact, with a mean score of 4.3 for both undergraduates and graduates (median 4 for undergraduates and 5 for graduates). Some of the comments left by participants in the post-study questionnaire explain the reasons behind this evaluation. Several of them explained how “*once found the sequence diagram(s) describing the feature(s) involved in a change request, it was easy to identify the candidate impacted components. This strongly speeds up the tasks.*” Others explained as sequence diagrams “*represent a fair compromise between use cases (too abstract) and class diagrams (providing useless details about an entire subsystem)*”.

Class diagrams and source code were generally ranked as equally useful. However, while undergraduates found source code slightly more useful (mean 3.6, median 4) than graduates (mean 3.2, median 3.5), the opposite happens for class diagrams, that were found more useful by graduate students (mean 3.9, median 4) than by undergraduates (mean 3.4, median 3). Among the 8 participants that considered class diagrams very useful, five of them explained as “*it is easy to map class diagrams on source code, and thus to fast check the candidate impacted components identified from the diagram.*” Five of the 33 participants declared the source code as the most useful artifact. The perceived reason is that: “*while the provided high-level documentation is useful to speed-up the task, consulting source code is mandatory to perform some of them, like the required bug-fixes.*”

As for use cases, it is interesting to note that the usefulness assessment provided by graduates (mean 3.2, median 3) is higher than for bachelor (mean 1.5, median 1). This is the only case for which the Mann-Whitney test reveals a statistically significant difference (p-value=0.002, Cliff's  $d$  0.68 – high), while for all other artifacts the differences between the two levels of experience are not significant. This suggests how more experienced participants are able to start the task from

TABLE IV. WHAT PARTICIPANTS LOOKED FIRST.

Artifact	All data		Undergrad.		Graduates	
	# of Tasks	Perc (%)	# of Tasks	(%)	# of Tasks	(%)
Use Case	31	11.92	3	3.16	28	16.97
Sequence Diagram	66	25.38	23	24.21	43	26.06
Class Diagram	45	17.31	9	9.47	36	21.81
Javadoc	9	3.46	4	4.21	5	3.03
Source Code	109	41.92	56	58.95	53	32.12

requirements/use cases before accessing models and source code. Undergraduates failed to explain use cases, as they tried to identify object names within them “*in use cases it was not possible to find information about components of the system impacted by a change*”, rather than relying on use cases to identify the piece of functionality to be changed before accessing sequence/class diagrams.

Finally, the provided feedbacks confirmed that Javadocs were perceived as the least useful artifacts. For Javadoc, the mean and median score was 2 (“useless”) for both undergraduate and graduate students. Participants declared that “*with the other sources of documentation available Javadoc became useless to identify impacted components.*” This is to say, our study does not show that Javadoc is useless: it is likely to be very useful during development activities, e.g., when using a new API. Instead, it provides a limited (or no) support when analyzing the impact of a change.

As explained in Section II-E, we also analyzed the presence of possible correlations between the time spent by participants on the different kinds of artifacts and the correctness of the performed tasks in terms of recall, precision, and F-measure. By applying the Spearman correlation test no interesting correlations were found for undergraduate and graduate students, as well as when considering all participants as a single dataset. Also, a logistic regression model for correctness based on the use (or not) of different kinds of artifacts did not lead to any significant result, i.e., none of the artifacts resulted significant in the model.

#### Summary for RQ<sub>1</sub>.

- 1) Participants spent more time to analyze low-level artifacts as compared to high-level artifacts.
- 2) Participants consider sequence diagrams as the most useful source of documentation when performing the required tasks, followed by class diagrams and source code.
- 3) Undergraduate students spent a significantly higher proportion of time on source code than graduate students who, instead, spent more time on class diagrams.

*B. RQ2: How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?*

Table IV reports, for each kind of artifact used in our study, the number and percentage of tasks participants started from such artifact. The most frequent starting point is by far source code (42% of the tasks), followed by sequence diagrams (25%), class diagrams (17%), use cases (12%), and Javadoc (3%). Note that this result is quite surprising since one could expect that developers start their analysis from high-level artifacts going down to the code. Instead, in our study 84%

TABLE V. PATTERNS FOLLOWED BEFORE REACHING SOURCE CODE.

Pattern	All data		Undergrad.		Graduates	
	# of Tasks	(%)	# of Tasks	(%)	# of Tasks	(%)
S	36	13.85	17	17.89	19	11.51
D	35	13.46	8	8.42	27	16.36
(SD)+	22	8.46	2	2.10	20	12.12
(US)+	18	6.92	2	2.10	16	9.70
U(SD)+	7	3.46	1	1.05	6	3.64
(DS)+	7	2.69	1	1.05	6	3.64
J	4	1.54	3	3.16	1	0.60
U	4	1.54	0	0.00	4	2.42
S(US)+	3	1.15	1	1.05	2	1.21
SU(SD)+	2	0.77	0	0.00	2	1.21
Other	13	5.00	4	4.21	9	5.45

S = Sequence Diagram, D = Class Diagram  
U = Use Case, J = Javadoc

of the tasks started from source code and design models, i.e., class or sequence diagrams.

When observing data for different levels of experience (right-side of the table), what we notice is pretty consistent with findings of RQ<sub>1</sub> concerning the proportion of usage for different kinds of diagrams. Basically, undergraduates tend to start tasks mainly using source code (58%), while this percentage is only 32% for graduates. The percentage of participants starting with sequence diagrams is similar (24% for undergraduates, 26% for graduates), while graduates tend to start with class diagrams more than undergraduates (22% vs 9%). Finally, there is a non-negligible proportion of graduates that starts from use-cases (17%, vs. 3% of undergraduates). This is likely due to the fact that graduate students have a better training on software engineering principles and on how using models and high-level artifacts during maintenance tasks, and also because they have more experience in evolving existing systems.

Since we found that in 58% of cases source code does not represent the entry point, we analyzed what are, in these cases, the pattern followed by participants before reaching source code. Table V reports all possible patterns followed by participants (using through regular expressions). In a similar proportion of tasks, participants access sequence or class diagrams before going to source code. This happens in 71 tasks, 36 for sequence (14%) and 35 for class (13%) diagrams.

Another frequently followed path consists of one or more switches between sequence and class diagrams. This path is more frequent starting from the sequence (22 tasks)—row (SD)+ in Table V—than from class diagrams (7 tasks)—row (DS)+ in Table V. In both cases, participants tried to gain source code knowledge from its most direct model representations (i.e., class and sequence diagrams) before going through it. Also, for 18 tasks, participants switch one or more times between use case (used as starting point) and sequence diagrams—row (US)+ in Table V. Overall, it is interesting to note that sequence diagrams are accessed in four out of the five most frequent path followed before reaching source code. Other paths reported in Table V are quite uncommon, e.g., opening a use case (row U) or a Javadoc file (row J).

When looking at results by different levels of experience (right-side of Table V), it can be noticed that, besides what it is known already from previous analyses, graduate students use much more navigation patterns across different kinds of diagrams. As the table shows, undergraduates just looked at

TABLE VI. AVERAGE TRANSITION FREQUENCIES BETWEEN THE KINDS OF ARTIFACTS.

From/To	KINDS OF ARTIFACTS.				
	U	S	D	J	C
U		56%	8%	0%	36%
S	5%		17%	1%	77%
D	2%	18%		2%	78%
J	0%	6%	16%		78%
C	7%	49%	37%	7%	

S = Sequence Diagram, D = Class Diagram  
U = Use Case, J = Javadoc, C = Source Code

sequence or class diagrams before diving into source code. Instead, graduate students also followed more complex navigation patterns, e.g., sequence+class (with some iterations), use case+sequence (with some iteration), or even use cases followed by iterations on sequence and class diagrams. Once again, this indicated that people with more experience are more prone to follow an “integrated” approach when performing a comprehension task. Then, we analyzed the transition frequencies between the different kinds of artifacts used in our study. Table VI reports the results considering the entire dataset. As it can be noticed, the most frequent transitions are toward the source code (column C), 77% of which are from a sequence diagram, and 78% from class diagrams and Javadoc files. The take-away of these results is that, after have gathered information from one of those kinds of artifacts, developers try to map them into source code elements. Note that this is true also when separately analyzing participants having different experience levels with small changes in the transition frequencies.

The behavior of participants when reading use cases is, instead, pretty different from the one observed above. They shift toward source code in just 36% of times, privileging the reading of a design diagram (64% of the cases, 56% for sequence and 8% for class diagrams) before reaching source code. However, when analyzing the data for participants having different experience, some differences came out. In particular, graduate students tend to consult a low-level diagrams after accessing an use case (72%, 64% for sequence and 8% for class diagrams), against the 43% of undergraduates (35% for sequence and 8% for class diagrams). After reading an use case, undergraduates go to source code in 56% of cases, against the 27% of graduates. This further confirms that more experienced developers are more prone to use different sources of documentation when performing a comprehension task.

Other common transitions between different kinds of artifacts occur (i) when reading a sequence diagram toward a class diagram (17%) and (ii) when reading a class diagram toward a sequence diagram (18%). In this case, no interesting difference has been observed between participants having different experience.

It is also interesting to analyze what other artifacts participants access immediately after browsing source code. Table VII indicates that participants go back from source code to documentation just to access design diagram, i.e., sequence (49%) and class (37%) diagrams. Again, no important differences were found between participants having different experience.

Finally, we analyzed the most frequent navigational patterns followed by participants during the tasks. Table VII and Fig. 4 report information about the seven most frequent patterns we found. In particular, Table VII reports the number

TABLE VII. MOST FREQUENT NAVIGATIONAL PATTERNS.

Pattern	All data		Undergrad.		Graduates	
	Occ.	(%)	Occ.	(%)	Occ.	(%)
USDC	12	4.62	0	0.00	12	7.27
USD	21	8.08	2	2.11	19	11.52
USC	35	13.46	10	10.53	25	15.15
UDC	13	5.00	3	3.16	10	6.06
SDC	55	21.15	15	15.79	40	24.24
SC	153	58.85	58	61.05	95	57.58
DC	128	49.23	39	41.05	89	53.94

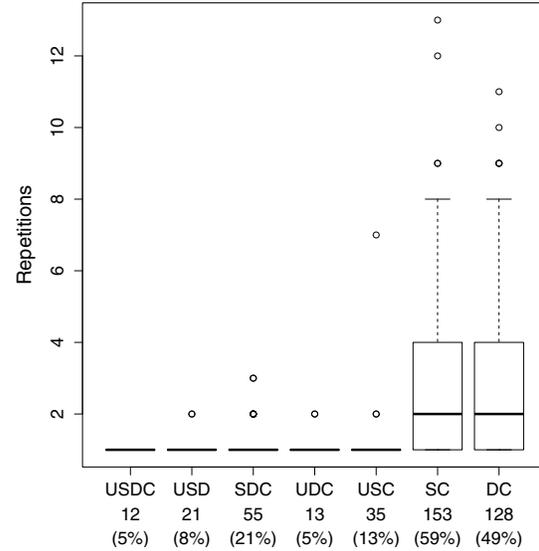


Fig. 4. Most frequent navigational patterns and distribution of their repetitions. S = Sequence Diagram, D = Class Diagram, U = Use Case, C = Source Code.

and percentage of occurrences on the whole dataset and for participants with a different degree of experience, whereas Fig. 4 shows the boxplots for the distribution of its repetitions (i.e., the number of times a pattern appears in a single task). As it can be expected according to what observed so far, the most frequent pattern consists of going back and forth from sequence diagram to source code: this occurred in 153 tasks (59%). The median of its repetitions is two, but we also found cases where this pattern has been repeated more than 10 times in a single task. Another very frequent pattern is that going back and forth from class diagrams to source code, present in 128 tasks (49%) with also a median repetition of two. Among the longer patterns (i.e., those having a length > 2), the most frequent is that going from sequence to class diagram and then to source code (SDC in Fig. 4). This pattern has been followed by participants in 21% of the performed tasks, generally with a single repetition. Also, in 13% of tasks, participants went from use cases toward sequence diagrams, and finally to the code. In addition, from the analysis of Fig. 4 we can conclude that (i) Javadoc is not present in any of the most common patterns; and (ii) all common patterns end (as expected) with a source code artifact.

When looking at the occurrences of patterns among participants with a different level of experience (right-side of Table VII), we can notice that (i) the SC pattern (sequence+code) is consistently followed by about 60% of both undergraduates and graduates; (ii) patterns involving use cases (USDC, USD,

USC, and UDC) are much more frequent for graduate than for undergraduates; and (iii) for what concerns longer patterns followed by undergraduates, the SDC pattern was followed in 16% of the cases, and USC in 10% of the cases. In summary, we can notice a higher proportion of patterns reflecting a more “integrated” approach for graduates. Also, graduate students followed patterns involving class diagrams and code (DC) more (54%) than undergraduates (41%). We did not notice any significant difference in the number of iterations for all the above mentioned patterns, except for the SC pattern, that received a median of 3 iterations for undergraduates, that used it and of 2 iterations by graduates that used it. The difference is statistically significant (p-value =00017) and the Cliff’s  $d$  effect size medium ( $d = 0.293$ ). In other words, less-experienced participants had to go back and forth between sequence diagrams and source code more than experienced ones to locate the methods to be changed.

As done for RQ<sub>1</sub>, we also statistically verified the relationship between the patterns followed by participants and the correctness of their tasks. In particular, we built a logistic regression model for correctness with respect to the use (or not) of the different patterns. Also in this case, we did not find any statistically significant result.

**Summary for RQ<sub>2</sub>.**

- 1) Participants tend to start the assigned task from source code or from design documents, i.e., class and sequence diagrams.
- 2) More experienced participants tend to follow a more integrated approach than less experienced ones, traversing different kinds of diagrams, e.g., starting from use cases, and then browsing design documents, until reaching source code.
- 3) During their task, participants tend to go back and forth repeatedly between source code and to design diagrams (sequence and class diagrams).

**IV. THREATS TO VALIDITY**

Threats to *construct validity* concern the relation between the theory and the observation. In our study, this threat can mainly be due to errors in the collected measurements. For what concerns capturing participant’s browsing activities, we relied on an existing tool (FLUORITE), making sure each participant had correctly installed it, and carefully instructed them how to browse artifacts in Eclipse while using the tool. When collecting results, we discarded cases of short access to artifacts (less than five seconds) that are unlikely to be an indication of reading the document, but rather of scrolling different documents. Clearly, this might have meant losing some quick, but valid, accesses. Another threat concerns the way the correctness of the task is evaluated, i.e., by means of precision, recall, and F-measure computed over the list of elements to be modified as identified by participants. On the one hand this allows a subjective evaluation and allows to perform a comprehension task without requiring the execution of source code. On the other hand, this can provide a coarse-grained and partial evaluation of how the comprehension task was performed.

Threats to *internal validity* concern any confounding factor that could influence our results. For example, such a threat may be due to the fact that some participants might have decided not to browse diagrams because they were unreadable or the tool was not usable. To mitigate such a threat, we avoided to use any specific UML modeler (we used PDF documents instead), and we produced diagrams large enough to be easily readable.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. As explained in Section II, this is more an observational study rather than a controlled experiment, as all participants received the same treatment. Wherever possible, however, we used appropriate statistical procedures and effect size measure to support our claims.

Threats to *external validity* concern the generalization of our findings. This study has been conducted with students, and for this reason the obtained results may not generalize to professionals, which might be used to perform comprehension task using high-level artifacts in a different way, or in some cases not using them at all. To some extent, our participants can be considered as representative of junior developers, joining a project as newcomers to perform a maintenance task. Another threat to external validity is related to the use of source code and documentation related to a single project. We do not know whether the comprehension of other kinds of projects would benefit of navigation patterns different than those discovered in this paper.

**V. RELATED WORK**

Several studies have been performed to analyze the benefits of UML documentation during software development and evolution [4]. In the next section we focus the attention on studies analyzing the effect of documentation on maintenance/comprehension tasks. In addition, we also discuss studies carried out to analyze the behavior—from different perspectives—of developers performing maintenance tasks.

**A. Impact of UML documentation on Maintenance Tasks**

Experiments aimed at studying the impact of UML documentation in software maintenance [2] indicated that such a documentation improves the functional correctness of changes and the quality of the design. While simple class diagrams, with or without stereotypes, help low ability or low experience participants, a complete, thorough UML documentation requires a certain learning curve to become useful [2]. In fact, in some cases the previous experience of participants influences the understandability of UML diagrams. Torchiano [27] showed that object diagrams have a significant impact on comprehension tasks, when compared with UML documentation consisting of class diagrams only. Dzidek *et al.* [9] performed a controlled experiment aimed at investigating the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of software systems. In the context of the experiment, participants (represented by professional developers) performed evolutionary tasks with and without UML documentation. Their results indicated that participants using UML documentation were able to statistically increase the correctness of changes.

UML limitations in aiding program understanding are highlighted in experiments performed by Tilley and Huang [26]. They highlighted that UML does not provide a sufficient support to represent domain knowledge.

Scanniello *et al.* [21] analyzed if source code comprehension increases when exploiting UML class and sequence diagrams. An experiment conducted with 16 Master's students show how participants benefited from the use of the UML diagrams during comprehension activities.

The role of dynamic UML diagrams in software comprehension was investigated by Otero and Dolado [17]. The comprehension level and the time required to perform the comprehension task resulted different for different diagrams and system complexities. Abrahão *et al.* [1] also analyzed the support given by sequence diagrams during the comprehension of functional requirements. The results showed that sequence diagrams improve the comprehension of the functional requirements in the case of high ability and more experienced participants.

We share with the aforementioned studies the need to analyze the support given by software documentation during software evolution. However, we did not focus on a specific kind of documentation. Instead, we provided to participants several documentation artifacts aim at studying which are the most used artifacts and how developers use such artifacts.

Tryggeseth [28] conducted a study, for some aspects similar to our, to analyze the impact of the availability of up-to-date documentation on maintenance tasks. In the context of the experiment participants were asked to perform maintenance tasks with and without software documentation (requirements specification, design document, test report, and user manual). Their results indicated that participants using the available documentation spent less time to understand how to implement a change request. Besides reducing the time, the documentation also allowed participants to better understand the system and provided more detailed solution on how to incorporate the changes. While we share with Tryggeseth the need for analyzing the impact of several software documentation artifacts on maintenance tasks, our study presents two main differences: (i) we analyzed how developers use documentation during software evolution aimed at identifying particular navigation paths; and (ii) we also investigated the effect of experience on how participants follow different usage paths.

### B. Developers' Behavior during Maintenance Tasks

von Mayrhauser and Vans [29] observed how professional developers work when performing maintenance tasks, finding that programmers use a multi-level approach during source code understanding, switching between different programs as well as between different sources of documentation.

Robillard *et al.* [20] performed an exploratory study to analyze the factors that contribute to effective program investigation behavior, while Sillito *et al.* [23] performed two qualitative studies aimed at understanding what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools help in that process.

Singer *et al.* [24] studied the daily activities of developers. Such a study provides some guidelines for tool designers that represent an alternative to the traditional paths taken in human-computer interaction, namely those issuing from the study of the users' cognitive processes and mental models, and the emphasis on usability. Also, DeLine *et al.* [8] identified several usability issues of conventional development environments when a developer has to update a software system, including maintaining the number and layout of open text documents and relying heavily on textual search for navigation.

de Alwis and Murphy [7] analyzed how programmers experience disorientation when using Eclipse, identifying three factors that may lead to disorientation: the absence of connecting navigation context during program exploration, thrashing between displays to view necessary pieces of code, and the pursuit of sometimes unrelated subtasks.

Storey *et al.* [25] performed a study aimed at analyzing whether program understanding tools enhance or change the way that programmers understand programs. Based on the results achieved the authors suggested that tools should support multiple strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration.

The behavior of software developers has also been analyzed aimed at identifying approaches able to reduce the information overload (e.g., number of artifacts to be analyzed) of developers by filtering and ranking the information presented by the development environment [10], [14], [15]. The findings of our paper can complement such models. The usage patterns identified in our study can be used to complement such approaches providing a more effective support during program comprehension.

Recently, eye tracking systems have been used to investigate the comprehension of UML diagrams [12], [30], the effect of the layout on the comprehensibility of software documentation artifacts [22], and the effect of design patterns on comprehension [13]. The use of eye tracking systems is particularly useful to investigate on the way developers look at the documentation aimed at deriving guidelines for facilitating the comprehension of software documentation.

We share with all these studies the need to empirically analyze the behavior of developers during software development and maintenance. However, we analyzed the behavior from a different perspective. Specifically, our analysis aimed at analyzing how developers use software documentation in order to identify recurring usage paths. Such paths could be used to enhance contemporary IDEs and provide more effective strategies for browsing documentation artifacts.

## VI. CONCLUSION AND FUTURE WORK

This paper reported a study aiming at investigating how developers navigate and browse documentation artifacts during maintenance tasks. We asked 33 participants to perform 8 different maintenance tasks on a Java software system providing them, besides the source code, use case descriptions, sequence diagrams, class diagrams, and Javadoes. Through an Eclipse plugin, we recorded how much time participants spent on different artifacts, and how they navigated from an artifact to another.

Results of our study indicated that participants spent most of their time on source code when identifying code components impacted by a maintenance activity, while preferring sequence diagrams among the available sources of documentation, followed by class diagrams. Also, they generally started their tasks from source code, or from design documents (84% of cases), then browsing back and forth between source code and either class or sequence diagrams. Less frequently, participants—especially more experienced ones (i.e., graduate students)—followed an “integrated” approach, by using different kinds of artifacts, namely starting from use cases, then accessing design documents (class and/or sequence diagrams), and finally accessing source code.

As a first direction for future work, we plan to corroborate our results by replicating our study with different participants and systems. Moreover, we plan to conduct other controlled experiments to explicitly investigating possible relationships existing between the way developers use the available documentation and the correctness of the tasks they perform.

## VII. ACKNOWLEDGEMENT

We would like to thank all the students that participated in our study.

## REFERENCES

- [1] S. Abrahão, C. Gravino, E. Insfrán, G. Scanniello, and G. Tortora. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transaction on Software Engineering*, 39(3):327–342, 2013.
- [2] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.
- [3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
- [4] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Software: Practise and Experience*, 41(4):363–392, 2011.
- [5] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.
- [6] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini. Evaluating the effect of composite states on the understandability of UML statechart diagrams. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*.
- [7] B. de Alwis and G. C. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 51–54, Brighton, UK, 2006. IEEE Computer Society.
- [8] R. DeLine, A. Khella, M. Czerwinski, and G. G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the ACM 2005 Symposium on Software Visualization*, pages 183–192, St. Louis, Missouri, USA, 2005. ACM.
- [9] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Transaction on Software Engineering*, 34(3):407–432, 2008.
- [10] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer’s activity indicate knowledge of code? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–350, Dubrovnik, Croatia, 2007. ACM.
- [11] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
- [12] Y.-G. Guéhéneuc. TAUPE: towards understanding program comprehension. In *Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research, October 16-19, 2006, Toronto, Ontario, Canada*, pages 1–13. IBM, 2006.
- [13] S. Jeanmart, Y.-G. Guéhéneuc, H. A. Sahraoui, and N. Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78, Lake Buena Vista, Florida, USA, 2009.
- [14] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, Oregon, USA, 2006. ACM.
- [15] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.
- [16] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.
- [17] M. C. Otero and J. J. Dolado. An initial experimental assessment of the dynamic modelling in UML. *Empirical Software Engineering*, 7(1):27–47, 2002.
- [18] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. ISBN 3-900051-07-0.
- [19] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. How developers’ experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. *IEEE Transactions on Software Engineering*, 36(1):96–118, 2010.
- [20] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.
- [21] G. Scanniello, C. Gravino, and G. Tortora. Does the combined use of class and sequence diagrams improve the source code comprehension?: results from a controlled experiment. In *Proceedings of the 2nd International Workshop on Experiences and Empirical Studies in Software Modelling*, pages 4:1–4:6, 2012.
- [22] B. Sharif and J. I. Maletic. An eye tracking study on the effects of layout in understanding the role of design patterns. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–10, Timisoara, Romania, 2010. IEEE Computer Society.
- [23] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.
- [24] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research*, page 21, Toronto, Ontario, Canada, 1997. IBM.
- [25] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, 2000.
- [26] S. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *SIGDOC ’03: Proceedings of the 21st annual international conference on Documentation*, pages 184–191, New York, NY, USA, 2003. ACM Press.
- [27] M. Torchiano. Empirical assessment of UML static object diagrams. In *Proceedings of the International Workshop on Program Comprehension*, pages 226–229. IEEE Computer Society, 2004.
- [28] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering*, 2(2):201–207, 1997.
- [29] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the 16th international conference on Software engineering, ICSE ’94*, pages 39–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [30] S. Yusuf, H. H. Kagdi, and J. I. Maletic. Assessing the comprehension of UML class diagrams via eye tracking. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 113–122, Banff, Alberta, Canada, 2007. IEEE Computer Society.

# Architecture Compliance Checking of Semantically Rich Modular Architectures

## A Comparative Study of Tool Support

Leo Pruijt, Christian Köppe

Information Systems Architecture Research Group  
 HU University of Applied Sciences  
 Utrecht, The Netherlands  
 {leo.pruijt, christian.koppe}@hu.nl

Sjaak Brinkkemper

Department of Information and Computing Sciences  
 University Utrecht  
 Utrecht, The Netherlands  
 s.brinkkemper@uu.nl

**Abstract**—Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. ACC is used to prevent architectural erosion during the development and evolution of a software system. Static ACC, based on static software analysis techniques, focuses on the modular architecture and especially on rules constraining the modular elements. A semantically rich modular architecture (SRMA) is expressive and may contain modules with different semantics, like layers and subsystems, constrained by rules of different types. To check the conformance to an SRMA, ACC-tools should support the module and rule types used by the architect. This paper presents requirements regarding SRMA support and an inventory of common module and rule types, on which basis eight commercial and non-commercial tools were tested. The test results show large differences between the tools, but all could improve their support of SRMA, what might contribute to the adoption of ACC in practice.

**Keywords**—Software Architecture; Modular Architecture; Architecture Compliance; Architecture Conformance; Architectural Erosion; Static Analysis

### I. INTRODUCTION

Software architecture is of major importance to achieve the business goals, functional requirements and quality requirements of a system. However, architectural models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system [1]. Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent decreased maintainability, caused by architectural erosion. *Architectural erosion* is “the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture” [2]. Opposing terms are architecture compliance and its synonym architecture conformance. Knodel and Popescu defined *architecture compliance* as “a measure to which degree the implemented architecture in the source code conforms to the planned software architecture” [3].

Many tools and techniques are available to analyze a software system, and to reconstruct, visualize, check, or restructure its architecture [4]. In our study we focus on tools

supporting static ACC, which analyze the software without executing the code. These tools, which we label as *static ACC-tools*, focus on the modular structure in the source code and identify structural elements such as packages and classes. In addition, they analyze use-relations between these elements, such as an invocation of a method or access of an attribute. Furthermore, these tools support the definition of rules on the structural elements in the code, or on logical modular elements that are mapped to the code. Finally, ACC-tools check the compliance and report violations to the rules. For example, if a method call from class A to class B in the code corresponds with a not-allowed dependency from a lower layer to a higher layer in the intended architecture, then the tool should report a violation.

Although Shaw and Clements included ACC in 2006 in their list of promising areas [5], the adoption of ACC-tools is still limited [2], [6], and research is necessary to advance and improve current methods and tools [7]. A few studies have compared ACC-tools and techniques, and these studies revealed large differences in terminology and approach. A high level overview of techniques and tools is included in a survey on architectural erosion [2] and in a survey on software architecture reconstruction [4]. Two other studies [3], [8] identified and compared five static ACC techniques at a more detailed level. One of these studies [8] also explored the effectiveness and usability of three tools, each representing one technique, by executing tests on the basis of a small system.

Our research builds on these previous studies, but we focus on ACC-tool support of *semantically rich modular architectures* (SRMAs). We use this term for expressive modular architectures, composed of different types of modules, which are constrained by different types of rules; explicitly defined rules, but also rules inherent to the module types. Kazman, Bass, and Klein have stated the principle that elements in a software architecture should be coarse enough for human intellectual control, but also specific enough for meaningful reasoning [9]. Modules with specific semantics, like subsystems, layers, components or facades, enhance the expressiveness of a modular architecture and support architecture reasoning. Adersberger and Philippsen consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering [10]. Furthermore, they make clear that support of semantically rich constructs reduces the number of rules that need to be

defined, compared to semantically poorer boxes and lines models.

We started our study with the following research question: *Do static ACC-tools provide functional support for semantically rich modular architectures?* To answer this question, we identified requirements, developed test-ware based on the requirements, and we tested eight ACC-tools. We restricted our study to the functional support of SRMAs by ACC tools, and consequently we do not focus on other aspects, like usability, scalability or accuracy (in another study, we investigated the accuracy of dependency analysis and violation reporting [11]). Other approaches than ACC that may be supported by the same tools, like architecture reasoning and re-engineering, are outside the scope of this paper as well.

The next section of this paper identifies the information available in semantically rich modular architectures, presents requirements and a classification of common module and rule types. Section 3 describes the test method and introduces the tools, while Section 4 holds the test results. Section 5 discusses the test outcome and compares it to related work, while Section 6 concludes this paper with recommendations, and addresses some issues that require further research.

## II. MODULAR ARCHITECTURES

### A. Focus of Static ACC

Software architecture compliance checking covers a large field, since software architecture is a broad term. According to Perry and Wolf, software architecture “provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system” [12]. Static ACC does not cover the full width of software architecture, but only the static structure of the software: the modular architecture. According to the Views and Beyond approach [13], [14], module styles focus on the structure of the units of implementation and not on runtime behavior or the allocation to non-software resources. Different module styles are defined such as the decomposition style, uses style, generalization style, and layer style.

A modular architecture should describe the modular elements, their form (properties and relationships) and rationale [12]. Modular elements, properties and relationships, are in ACC’s center of attention, and should be included in a complete compliance check. A *modular element*, or module, is an implementation unit of software with a coherent set of responsibilities [14]. Properties and relationships express architectural rules. *Properties* are used to define constraints on the modular element and its content. *Relationships* are used to constrain how the different elements may interact or otherwise may be related [12].

### B. Requirements Regarding SRMA Support

A semantically rich modular architecture may contain a lot of information about the modules and the rules constraining these modules. Modules may be of types with different semantics, while different types of rules may be used to constrain the modules. A rich set of module types

provides a language to express characteristics of the modules in an architectural model, as well as default constraints associated to the type of module. A rich set of rule types provides a language to express constraints on the modules in an architectural model. Provision of a rich rule set allows architects to define logical rules in a comparable way as expressed in regular language, without the need to translate a logical rule to one or more rules at tool level.

Consequently, to support compliance checks of SRMA’s, ACC-tools should preferably be able to: a) register common information in SMRAs (modular elements, properties and relationships of different types); b) prevent inconsistencies in the definition of the architectural model; and c) check the rules included in the architectural model and report violations. Inconsistencies in the model, like modules not properly mapped to code, will hamper the accuracy of the actual rule check. Consequently, inconsistencies should be recognized and reported.

In line with these requirements, we focused our research on the following questions. Do ACC-tools provide support for: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture?

To determine the module types, rules types and inconsistency checks relevant to our research, we studied academic and professional literature, as well as software architecture documents from professional practice and ACC-tool documentation. The following subsections describe the outcome of our study.

### C. Common Module Types.

SMRAs may contain modules of different types. We identified six common types of modules relevant for static ACC:

1) *Physical clusters* are the type of modules that represent a wide variety of software structures or units in the code, like classes, Java packages, or C# namespaces [14]. This type of module does not represent a unit in the design, but in the code.

2) *Logical clusters* represent units in the system design with clearly assigned responsibilities, but with no additional semantics. Comparable terms are subsystems, or packages.

3) *Layers* represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. The concept of layering can be traced back to the works by Dijkstra [15] and Parnas [16]. Although the layered style is not supported by UML [5], it is one of the most common styles used in software architecture [14], [17]. We cite Larman [18], who summarizes the essence of a layered design as “the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers.”

4) *Components* within a software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a

modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas [16]). Consequently, a component differs from a logical cluster in the fact that it has a Façade sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior, and a module in a module view may turn into many runtime components within the “component and connector view” [14].

5) *Façades* are related to a component and act as an interface as described under components. We use the term façade, referring to the façade pattern [19], to differentiate with the Java interface, which has not exactly the same meaning as a design-level interface. A façade may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.

6) *External systems* represent platform and infrastructural libraries or components used by the target system. Useful ACC support includes the identification of external system usage and checks on constraints regarding their usage [20].

#### D. Example of an SRMA

An example of an SRMA, with modular elements of different types, is shown in Fig. 1. The model shows a part of a modular architecture of one of the systems at an airport,

where it was subject of an ACC. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, et cetera.

Various notations for modular architecture diagrams are used in practice [14]. The example in Fig. 1 shows UML icons, but also an identification of the layers, not included in UML. The model combines three modular styles, namely the decomposition style, uses style, and layered style. Examples of modules of different types are visible in Fig.1. such as “Interaction layer”, logical cluster “HiWeb”, component “HiManager”, façade “HimInterface”, and external system “Hibernate”. The modules are easily identifiable, but the rules are not. In this case, the basic principle is, “no module is allowed to use another module”, except when a dependency relation indicates “is allowed to use”. Furthermore, the rules related to the layered style are not visible, but the default rules apply: Interaction Layer is not allowed to use Technology Layer (skip call ban); Technology Layer is not allowed to use Service layer or Interaction Layer (back call ban).

#### E. Common Rule Types.

SMRAs may contain rules of different types, where each rule type characterizes the constraint. Constraints in a software architecture are categorized in literature [12], [14] as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static ACC, resulted in two categories related to properties and relationships: Property rule types; and Relation rule types.

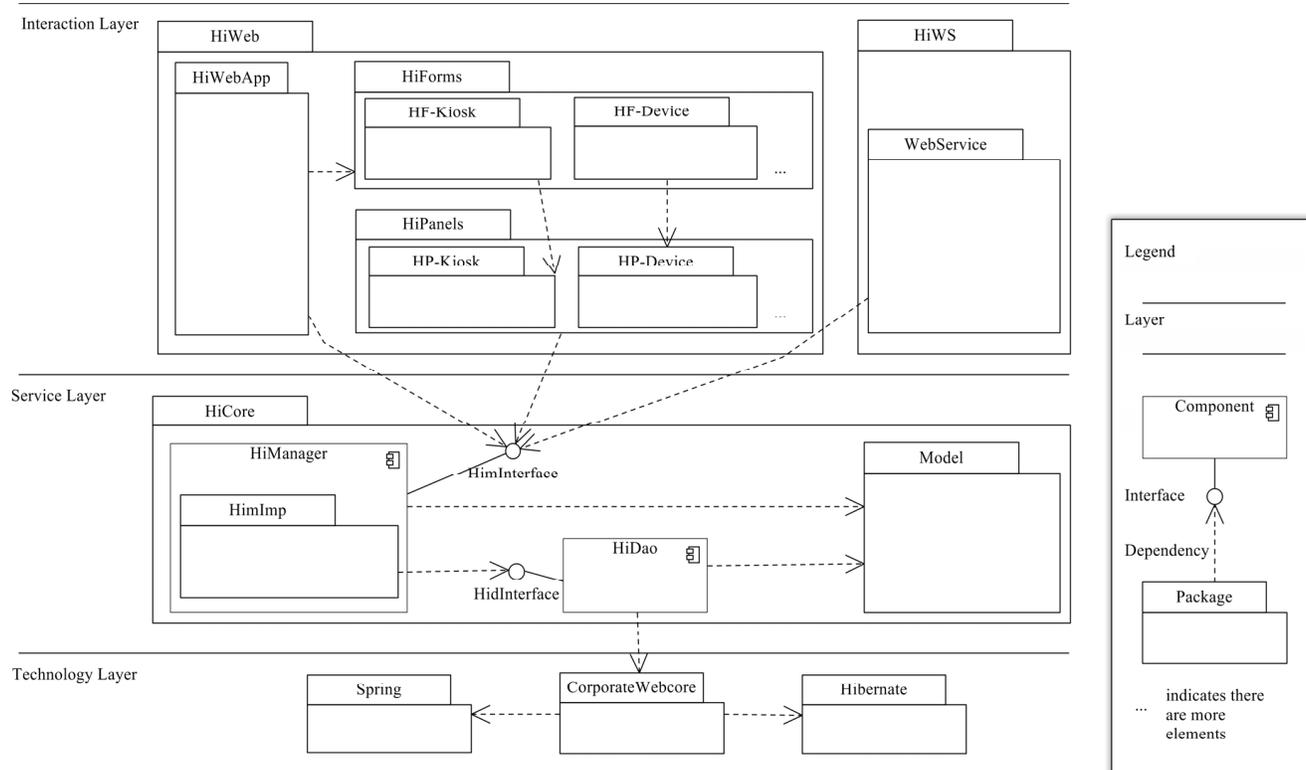


Figure 1. Example of a semantically rich modular architecture model

*Property rule types* constrain the elements included in the module; their sub modules, et cetera. Clements et al. [14] distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Façade convention, Inheritance convention), which represent the property Implementation information. The identified rule types are shown in Table I. The table contains per rule type: a description, an example, and an exemplary reference to literature covering the topic. The example rules constrain the modules of the modular architecture shown in Fig.1.

Naming conventions may be useful, since names are used by practitioners to unify software architecture and its implementation [21]. Responsibility conventions are useful to preserve the designed distribution of responsibilities over modules. Visibility conventions and Façade conventions can be used to enforce implementation hiding. Inheritance conventions may be used to enforce a selected generalization style. Finally, exceptions to property rules may be useful too. For instance, an exception to the Visibility convention example in Table I is, “HiManager classes have package visibility or lower, except for façade HimInterface.”

*Relation rule types* specify whether a module A is allowed to use a module B. The basic types of rules are “is allowed to use” and “is not allowed to use”. However, we encountered useful specializations of both basic types, which we included in the classification shown in Table I. When several rules of the same type are defined on the same from-module, then they should be interpreted as complementary rules; even if the word “only” is part of the name of the rule type.

Some rule types are complex, because they include dependency checks on other modules than only the from-module and to-module. Exceptions to all relation rules are complex, as well as the two following types: “Is only allowed to use”, and “Is the only module allowed to use”. Complex rule types are very useful in practice, for the following reasons:

- Complex rule types allow architects to define rules in a comparable way as expressed in regular language. Complex rules of type “Is only allowed to use” may constitute a significant part of the total rule set [22].
- Complex rule types help to transform rules in a UML-like diagram to rules in most ACC-tools. For instance, the dependency relationship from module HF-Kiosk to module HP-Kiosk in Fig.1 expresses the rule “HF-Kiosk is only allowed to use HP-Kiosk.” Transformation is often necessary. The basic principle underlying UML-like diagrams is restricting (no other than the defined dependencies are allowed), while in most tools, the basic principle is non-restricting (all dependencies are allowed, unless there is a not-allowed-to-use rule).
- Complex rule types may diminish the number of rules, since one complex rule often replaces many “is not allowed to use” rules. For instance, when the “is only allowed to use” rule type is not supported by a tool, then the dependency relationship from module HF-Kiosk to module HP-Kiosk in Fig.1 may have to be translated to many “not allowed to use” rules from HF-Kiosk to all the other modules, except to HP-Kiosk.

TABLE I. COMMON RULE TYPES (REF= PRIMARY LITERATURE REFERENCE)

Category\Type of Rule	Description (D), Example (E)	Ref
<b>Property rule types</b>		
Naming convention	D: The names of the elements of the module must adhere to the specified standard. E: HiDao elements must have suffix DAO in their name.	[8]
Responsibility convention	D: All elements of the module must adhere to the specified responsibility. E: HiForms is responsible for presentation logic only.	[18]
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.	[14]
Facade convention	D: No incoming usages of the module are allowed, except via the façade. E: HiManager may be accessed only via HimInterface.	[19]
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.	[8]
<b>Relation rule types</b>		
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HF-Kiosk is not allowed to use HP-Device.	[3]
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.	[23]
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a lower layer that is more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.	[23]
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms (including its sub modules).	[14]
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-module(s). E: HF-Kiosk is only allowed to use HP-Kiosk.	[8]
Is the only module allowed to use	D: No elements, outside the selected module(s) are allowed to use the specified to-module. E: HiDao is the only module allowed to use CorporateWebcore.	[8]
Must use	D: At least one elements of the module must use the specified to-module. E: HiDao must use CorporateWebcore.	[3]

### F. Associations between Module and Rule Types

Optimal support of SRMAs includes the automatic provision of rule types inherent to the type of module. For instance, layers are inherently associated to a “Back call ban” rule and a “Skip call ban” rule. Furthermore, components are inherently associated to a “Façade convention” rule (and possibly a “Visibility convention” rule, if supported by the implementation language). Options to disable an inherent rule, for instance in case of a relaxed layered model, or to define an exception, will enhance the usability.

## III. TEST METHOD AND TESTED TOOLS

### A. Test Method

Based on the requirements and classification of module types and rule types described in Section 2, a test was designed to assess the ACC-tools on their SRMA support. For each rule type, at least two test cases were included: one without, and one with violations to the rule. A special test software system was developed in Java. This system included the various module types and separate packages for each rule type, which contained classes with injected violations to a rule and classes without. In addition, a test script was prepared to instruct the tester and to document the test results. The test script and test system are available on request.

After the test preparation, the eight ACC-tools were

tested. During the first step of the test of a tool, the intended architecture was entered. Thereafter, the modules were mapped to source code units and the rules were entered into the tool. If a tool did not support a rule type explicitly, then we looked for a workaround; such as a combination of separate rules. The first step was concluded by test actions aimed at the tool’s ability to prevent inconsistencies in the architecture definition. During the second step, the outputs of the tool’s dependency analysis and conformance check were studied and compared with the expected result. During the third step, reports were prepared, after which the tools could be compared on their SRMA support.

Two iterations of testing and reporting were conducted. The first iteration was performed with 25 bachelor students in the course of a third year specialization semester “Advanced Software Engineering”, where each team studied and tested a tool. In a second iteration, the authors studied the tools and verified and refined the results of the students, by using the tools and repeating the tests. ConQAT was added afterwards to our tool set and was tested only by the authors.

### B. ACC-Tools Included in the Test

Many tools are available with some facilities to support ACC. Our research focused on tools with explicit support of ACC. We selected eight publicly available tools, which were mentioned in academic work (e.g., [4] [8] [10]), were able to

TABLE II. CHARACTERISTICS OF THE TOOLS IN THE TEST

Tools <sup>1</sup> → Characteristics ↓	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
<b>General functionalities</b>								
Dependency browsing		√	√				√	√
Dependency visualization			√		√		√	√
Architecture compliance checking	√	√	√	√	√	√	√	√
Architecture refactoring/simulation			√				√	√
Team support							√	√
<b>Code variants</b>								
Java	√	√	√	√	√	√	√	√
Other languages	√		√		√			√
Source file analysis					√	√	√	
Compiled file analysis	√	√	√	√		√	√	√
<b>Licensing</b>								
Free: commercial and non-commercial use	√	√		√		√		
Paid: commercial use			√		√		√	√

<sup>1</sup> ConQat AA– version 2011.9 – [www.conqat.org](http://www.conqat.org);

dTangler - GUI version 2.0 - [web.sysart.fi/dtangler](http://web.sysart.fi/dtangler);

Lattix LDM - version 7.2 - [lattix.com](http://lattix.com);

Macker - version 0.4.2 - [sourceforge.net/projects/macker](http://sourceforge.net/projects/macker);

SAVE - version 1.7 - [iese.fraunhofer.de](http://iese.fraunhofer.de);

Sonar ARE - version 3.2 - [docs.codehaus.org/display/SONAR/Architecture+Rule+Engine](http://docs.codehaus.org/display/SONAR/Architecture+Rule+Engine);

Sonargraph Architect (fusion of Sotograph and SonarJ) - version 7.0 - [hello2morrow.com](http://hello2morrow.com);

Structure101 - version 3.5 - [structure101.com](http://structure101.com).

analyze Java, and provided evaluation or research licenses (two vendors rejected and one did not respond). We excluded tools that focus mainly on architecture visualization, metrics and/or architecture refactoring. The eight tools included in our study are shown in Table II, which also gives an overview of functionalities, code variants and licensing.

The tools provide their support of ACC in various ways. The eight tools can be subdivided in four categories of tools. 1) Macker and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools, which support relation conformance rules. These tools provide HTML-based violation reports. 2) dTangler and Lattix are based on the Dependency Structure Matrix (DSM) technique, complemented with text-based editors to define rules. The DSM is used to select modules and to show dependencies and violations. Lattix is also able to visualize architectures graphically, and provides extensive reporting facilities.

3) ConQAT Architecture Analysis (ConQAT AA) and SAVE are strictly based on the Reflexion Model (RM) technique [1], and both tools provide a graphical editor to define the intended architecture and to show violations after the evaluation. Textual reports are generated at request. 4) Sonargraph Architect and Structure101 are diagram-based too, but these tools are not based on the RM-technique. To define modules and rules, these tools provide diagrams in which the horizontal and vertical position of a module implies rules. Violations are shown in these diagrams, but textual reports are provided in addition.

## IV. TEST RESULTS

### A. Support of Common Module Types

In Section 2 we identified six common types of modules, relevant for static ACC. The results of our tests concerning the support of these module types are shown in Table III, and the most interesting findings are described below.

*Clusters* are supported by all tools. Five of the eight tools support *physical clusters*. The advantages to use them are that they allow fast, ad hoc rule checking; for instance, when there is no formal modular architecture. The disadvantage is the diminished or lost traceability to the formal modular architecture, if there is one. Sonar ARE is the only tool that supports only this type of modules. *Logical clusters* are supported by seven tools. Although in very different ways, these tools provide support to register logical clusters and to map the logical clusters to code units. Furthermore, support is provided to define rules constraining logical clusters and to check these rules at code level.

*Layers* are supported by only one tool, Structure101, on all indicators: modules can be marked as layers; back call and skip call rules are reported; and layers are visualized. Two other tools support the definition and visualization of layers, but do not provide inherent support of the related rules.

*Components* and *Facades* are supported by SAVE and Sonargraph Architect, on the following indicators: modules can be marked as component; facades can be defined. SAVE visualizes components and facades, but does not actively

TABLE III. TOOL SUPPORT OF COMMON MODULE TYPES (+ = EXPLICIT SUPPORT; ± = PARTIAL SUPPORT; - = NO SUPPORT)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
<b>Clusters</b>								
Physical cluster	-	+	+	+	-	+	-	+
Logical cluster	+	+	+	+	+	-	+	+
A module can be marked as layer	-	-	-	-	+	-	+	+
Back call violations are reported	-	-	-	-	-	-	-	+
Skip call violations are reported	-	-	-	-	-	-	-	+
<b>Components and Facades</b>								
A module can be marked as component	-	-	-	-	+	-	+	-
Facade can be defined	-	-	-	-	+	-	+	-
Facade-skip violations are reported	-	-	-	-	±	-	+	-
<b>External systems</b>								
A module can be marked as external system	-	-	-	-	-	-	+	-
A module can be mapped to an external system	+	+	+	+	+	+	+	+
Rules constraining their use are checked	+	+	+	+	+	+	+	+
<b>Visualization</b>								
Clusters are visualized	+	-	+	-	+	-	+	+
Layers are recognizable visualized	-	-	-	-	+	-	+	+
Components are recognizable visualized	-	-	-	-	+	-	-	-
Facades are recognizable visualized	-	-	-	-	+	-	+	-
External systems are recognizable visualized	-	-	-	-	+	-	+	-

support any of their semantics. Sonargraph Architect visualizes facades and supports their semantics; it reports facade-skip violations automatically when a facade is associated to a module. ConQAT AA seems to support components at first glance, since it depicts all modules as UML components. However, it does not provide any other icons and does not support the semantics of a component; reason why we classified ConQAT’s components as logical clusters.

*External systems* are not designated as a special module type by all tools, except Sonargraph Architect, but all enable conformance checks on modules mapped to external libraries.

Five tools support visualization of modular architectures. However, only two tools offer three or more different icons. A notable observation is that the tools that support semantically rich modules all have their own terminology, icons, rules and ways to visualize the architecture. SAVE provides an UML-like notation, while Sonargraph Architect and Structure101 position the modules horizontally and vertically. SAVE discerns five module types, while Sonargraph Architect discerns six types (which are only partly overlapping with those of SAVE), whereas Structure101 does not show the logical meaning of a module, but uses an icon to show the type of the related physical item.

### B. Support of Common Rule Types

In section 2 we identified twelve common types of rules, relevant for static ACC. The results of our tests concerning

the support of these rule types are shown in Table IV. Explicit support of a rule type is depicted by a “+”, meaning that one logical rule can be registered as one rule in the tool. Partial support, depicted by “±”, means that it is possible to register a rule of this type, but only via a workaround; often a combination of several rules. The most interesting findings from the test are described below.

#### 1) Property rule types

Property rule types are poorly supported. No tool provides facilities to specify and check conventions regarding naming, responsibility, or inheritance. Although names are used, in combination with regular expressions, to map modules to the code, no facilities are provided to check all the packages and/or classes contained by a module on conformance to a naming convention.

Only rule types to enforce implementation hiding are supported by some tools. Visibility convention rules are partly supported by Sonargraph Architect and Structure101. These tools provide a property to restrict the accessibility of a module, but do not check at code level on accessibility settings; reason why they did not score a “+”. However, when a module is marked as hidden or private, violation messages are reported, when dependencies to the module are detected from outside.

Facade convention rules are supported explicitly only by Sonargraph Architect. Four other tools enable the definition of this type of rules by default means, resulting in a combination of separate rules, so their support is scored with “±”.

#### 2) Relation rule types

Relation rule types are supported by all the tools, but no

TABLE IV. TOOL-SUPPORT OF COMMON RULE TYPES (+ = EXPLICIT SUPPORT; ± = PARTIAL SUPPORT; - = VERY WEAK OR NO SUPPORT)

Support is provided for	ConQAT AA	dTangler	Latix	Maaker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
<b>Property rule types</b>								
Naming convention	-	-	-	-	-	-	-	-
Responsibility convention	-	-	-	-	-	-	-	-
Visibility convention	-	-	-	-	-	-	±	±
Facade convention	-	±	±	±	±	-	+	-
Superclass inheritance convention	-	-	-	-	-	-	-	-
<b>Relation rule types</b>								
Is not allowed to use	+	+	+	+	+	+	+	±
Back call ban (inherent to layer)	-	-	-	-	-	-	-	+
Skip call ban (inherent to layer)	-	-	-	-	-	-	-	+
Is allowed to use	+	+	+	+	±	-	+	+
Is only allowed to use	±	±	±	±	±	-	±	±
Is the only module allowed to use	±	±	±	±	±	-	±	±
Must use	-	-	-	-	+	-	-	-
Exception (to relation rules)	±	±	±	±	-	-	±	±
<b>Visualization of rules and violations</b>								
Rules are visualized	+	-	+	-	+	-	+	+
Violations are visualized	+	+	+	-	+	-	+	+

more than three rule types are explicitly supported per tool.

Complex rule types (Is only allowed to use, Is the only module allowed to use, Exceptions to a relation rule) are not explicitly supported, or not at all. Without explicit support, workarounds are needed, for instance for the rule “HF-Kiosk is only allowed to use HP-Kiosk”. In Lattix, dTangler and Macker, two combined rules are needed such as: “HF-Kiosk Cannot-Use \$Root” + “HF-Kiosk Can-Use HP-Kiosk”. Since these rules are not related to each other, they form a threat to the maintainability and traceability of the set of rules. Sonargraph Architect and Structure101 may require the specification of more than two rules or property settings for complex rules, and sometimes many rules are needed, depending on the number and position of other modules. Sonar ARE provides no support at all to check complex rules. ConQAT AA and SAVE work quite differently from the other tools, since no transformation is required of rules in UML-like diagrams to rules in the tool. SAVE supports only the “Must use” rule type explicitly, while ConQat AA supports “Is allowed to use” and “Is not allowed to use” rule types. Complex rules can be checked, but this requires interpretation of the architecture model and the conformance check output.

### 3) Visualization

Six tools are able to visualize rules and violations. Lattix and dTangler show colors in a DSM. ConQAT AA, SAVE, Sonargraph Architect, and Structure101 use lines in diagrams to define and show rules, and to show violations. However, not all rules are visible in these diagrams.

### C. Support of Inconsistency Prevention

In section 2 we defined the requirement, “ACC-tools should prevent inconsistent definitions of modules and rules.” The results of our tests concerning this requirement are shown in Table V. Most tools allow, without a warning, incomplete or contradictory definitions of modules and/or rules. ConQAT AA scored best and prevented six out of six types of inconsistency included in our test. Lattix prevented five out of six types, while the other tools prevented or warned for upmost three types. Six of the tools start the compliance check without a warning when the defined modules and rules model is inconsistent. In such a case, the tool does not check all the rules as intended by the user, and

consequently the outcome of the check may be unreliable.

## V. DISCUSSION

To our opinion, all tested tools are providing useful functionality to support ACC or ad hoc rule checking. Apart from our laboratory experiments described in the paper, we used all seven tools to analyze an open source system. Furthermore, we performed ACCs on professional software systems with use of Lattix, Sonargraph Architect, and Structure101. Based on these experiences we can conclude that these tools are of great help for architecture reconstruction and ACC. However, our tests show that all seven tools could improve their support regarding SRMAs, though in varying degrees. Not one of the tested tools is able to support all the module types and rule types included in our classification. However, we encountered interesting examples of partial support. SAVE supports the graphical definition of modules of nearly all the types in our classification; only physical clusters are missing. However, SAVE’s rule language is very limited, and the semantics of the modules are not supported. ConQAT provides the same types of diagrams, but complements the rule setting capabilities considerably. Furthermore, ConQAT checks the consistency of the defined architectural model accurately. However, ConQAT provides one type of module only, and does not support any semantics. Sonargraph Architect and Structure101 are the only tools that actively support the semantics of two module types in our classification. Sonargraph Architect supports the definition of Facades and relates the “Façade convention” rule to a defined façade. Structure101 supports the definition of Layers and relates a layer to the “Back call ban” and the “Skip call ban” rules. Combination of these examples of partial support builds an image of the provision of full functional SRMA support.

Another observation during our study is that the combination of visualization, rule definition, and rule checking appears to be challenging. Lattix, dTangler, and Macker provide no support to define the architecture via a graphical editor, but enable the definition and checking of quite a diversity of rules, including complex rules. ConQAT, SAVE, Sonargraph Architect and Structure101 provide graphical support to define and check the architecture, but lack the freedom of rule definition, as provided by the

TABLE V. PREVENTION OF INCONSISTENCIES (+ = SUPPORTED; - = NOT SUPPORTED; N/A = NOT APPLICABLE)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Modules must have (unique ) name or ID	+	+	+	+	-	n/a	+	-
A module may have only one parent.	+	-	+	-	+	n/a	-	-
Modules must be mapped to code file(s)	+	-	-	-	-	n/a	-	+
Mapped code files must exist	+	-	+	-	-	n/a	-	-
Rules must be completely specified	+	-	+	+	+	-	+	+
Rules cannot be contradictory	+	-	+	-	+	-	+	+
Tool checks model prior to conf. check	+	-	+	-	-	-	-	-

textual-rule based tools. Furthermore, sometimes we experienced serious problems related to the graphical models. Defining sub-subsystems, exceptions, and other complex rules in the graphical models, is hard in some tools, and impossible in others. Furthermore, it may result in many lines, which makes the diagrams unreadable. Structure101 and Sonargraph Architect have introduced additional rule-setting techniques to reduce the number of required rule-lines. In these tools, the module type, the horizontal and vertical position, and the value of a visibility property per module may imply dependency rules. On top of that, Sonargraph Architect provides a “transversal access” variable per module as well. To our opinion, the combination of all the rule-setting techniques increases the complexity considerably, and it reduces the transparency of the set of defined rules.

#### A. Limitations

Our study can be characterized as a quasi-experiment, according to Wohlin et al. [24], since we did not work with a randomized selection of tools. Consequently, our findings may not be generalized to other tools, even though we tested eight tools in a small market.

Furthermore, we do not claim that our classification of common module types and common rule types is complete, since *common* is not a qualified term. We aimed to cover the most used types of modules and rules, reasoning from the functional point of view; the architect’s view, not the tool builder’s view. Creating the classification proved a valuable step in our study. The classification was used as a basis for our tests and will be used as starting point for our future work.

#### B. Related work.

Requirements regarding the functional support of ACC can be derived from quite a number of sources, like general literature on software architecture and design, and studies on ACC. In Section 2 we described the most relevant sources used for our requirements and classification. Several studies on ACC propose the inclusion of support for some specific module and/or rule types, for instance [10] [20] [3] [8]. However, to the best of our knowledge, none of these studies or other studies on ACC have provided and substantiated a broad inventory and classification of module and rule types. We intentionally did not include very specific or detailed module or rule types, but kept the set of requirements broad and not too specific. However, some interesting studies elaborate on particular types. For instance, Adersberger and Philippsen [10] describe the constraints and checks regarding components in detail. Furthermore, Terra and Valente [22] identified different types of dependencies (accessing methods and fields, declaring variables, creating objects, extending classes, implementing interfaces, throwing exceptions, and using annotations), and based fine grained rule types on these dependency types. Lattix, SAVE and Structure101 provide support to define or configure rules at this level of detail.

Not much comparative research on ACC-tools has been performed, as described in the Introduction section. Only

Passos et al. [8] presented similar work. They evaluated three tools, including Lattix and SAVE, on the basis of a very small system. During our study no findings have arisen that contradict their tool evaluations. Our study adds a substantiated set of requirements focused on SRMA support, as well as test results of eight tools.

## VI. CONCLUSION

Architecture compliance checking (ACC) relies on the support of tools to define modules and rules, to analyze the code, to check the compliance, and to report violations to the rules. In this study, we have investigated the support of semantically rich modular architectures (SRMAs) provided by static ACC-tools. We identified requirements to the support of SRMAs and classified module types and rule types relevant for static ACC. Furthermore, we prepared a test, and we tested eight tools on their support of SRMAs.

We started our study with the following research question: Do static ACC-tools provide functional support for semantically rich modular architectures? We focused our test on the support of: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture.

Our tests regarding the *support of common module types* show that five tools support non-semantic clusters only. The three other tools distinguish also one or more semantically rich module types from our classification. SAVE supports the graphical definition of five types of modules, but does not support their semantics. Sonargraph Architect supports the semantics of a Façade actively, while Structure101 supports the semantics of Layers actively. However, no tool provides the combined support of layers, components, and facades.

Our tests regarding the *support of common rule types* show that per tool only a few rule types are explicitly supported. Complex relation rules are by no tool explicitly supported. Consequently, complex relation rules at logical level require workarounds at tool-level, which often result in two or more unrelated rules; a threat to the maintainability and traceability of the set of rules. Furthermore, only two of the five property types are supported, and only partially, not explicitly.

Our tests regarding the *support of inconsistency prevention* show that only two tools, ConQAT and Lattix, score high on the prevention of inconsistencies in the module and rule model, while inconsistent models may result in an unreliable outcome of the compliance check.

Based on our study and experiments, we present the following recommendations to ACC-tool developers: 1) *Widen the scope of the tools from dependency checking to software architecture compliance checking, including SRMAs.* Provide explicit support for semantically rich module types with their related rule types. The requirements and the classification of common module and rule types, presented in this paper, may be used as a starting point. 2) *Minimize the difference between logical rules, as perceived by the architect, and the technical implementation in the tool.* Offer rule types that match with logical rule

types, including exceptions, and support each type explicitly.

3) *Provide one method to define and edit rules.* Do not mix several rule setting mechanisms. Keep it simple to the user.

4) *Provide several, best adaptable, views* on the modular structures, the rules, and the violations against the rules: reports, browsers, and diagrams. Do not mix too much types of information into one view.

5) *Check on inconsistencies in the architecture definition,* and inform the user when it is incorrect or incomplete.

Not all issues identified in this study can be solved easily. The provision of SRMA support calls for further research. Techniques need to be identified, and support needs to be designed and tested on effectiveness by means of prototypes and case studies. Specific topics deserve attention too. For instance, visualization, rule definition and rule checking appeared to be a challenging combination. Furthermore, automatic recognition of responsibility at code level, needed to check against the defined responsibility of a module, is an unresolved issue, though responsibility is an important property of a module at design level.

In conclusion, the eight tested tools provide useful support for ACC, but all could improve their support of SRMAs. Solutions need to be found to reduce the gap between documented modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side. More-complete functional support of SRMAs might contribute to the adoption of ACC and ACC-tools, and consequently could improve the effectiveness of software architecture in the practice and education of software engineering.

#### ACKNOWLEDGMENT

The authors would like to thank the students of the specialization “Advanced Software Engineering” at the HU University of Applied Sciences, but also colleagues and reviewers for their contributions to this study.

#### REFERENCES

- [1] G. C. Murphy, D. Notkin, and K. Sullivan, “Software reflexion models,” *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, Oct. 1995.
- [2] L. de Silva and D. Balasubramaniam, “Controlling software architecture erosion: A survey,” *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012.
- [3] J. Knodel and D. Popescu, “A Comparison of Static Architecture Compliance Checking Approaches,” in *Working IEEE/IFIP Conference on Software Architecture*, 2007, pp. 12–21.
- [4] S. Ducasse and D. Pollet, “Software Architecture Reconstruction: A Process-Oriented Taxonomy,” *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, Jul. 2009.
- [5] M. Shaw and P. Clements, “The golden age of software architecture,” *IEEE Software*, vol. 23, no. 2, pp. 31–39, 2006.
- [6] M. Gleirscher and D. Golubitskiy, “On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises,” *Software Quality. Process Automation In Software Development*, 2012.
- [7] G. Canfora, M. Di Penta, and L. Cerulo, “Achievements and challenges in software reverse engineering,” *Communications of the ACM*, vol. 54, no. 4, p. 142, Apr. 2011.
- [8] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Das Chagas Mendonca, “Static Architecture-Conformance Checking: An Illustrative Overview,” *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.
- [9] R. Kazman, L. Bass, and M. Klein, “The essential components of software architecture design and analysis,” *Journal of Systems and Software*, vol. 79, no. 8, pp. 1207–1216, 2006.
- [10] J. Adersberger and M. Philippsen, “ReflexML: UML-based architecture-to-code traceability and consistency checking,” in *Proceedings of the 5th European conference on Software architecture*, 2011, pp. 344–359.
- [11] L. Puijnt, C. Köppe, and S. Brinkkemper, “On the Accuracy of Architecture Compliance Checking Support: Accuracy of Dependency Analysis and Violation Reporting,” in *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 172–181.
- [12] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40 – 52, 1992.
- [13] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Third Edit. Addison-Wesley, 2012.
- [14] P. Clements, F. Bachmann, L. Bass, D. Garlan, P. Merson, J. Ivers, R. Little, and R. Nord, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2010.
- [15] E. W. Dijkstra, “The structure of the ‘THE’-multiprogramming system,” *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.
- [16] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [17] N. B. Harrison and P. Avgeriou, “Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation,” in *Seventh Working IEEE/IFIP Conference on Software Architecture*, 2008, pp. 147–156.
- [18] C. Larman, *Applying UML And Patterns*. Prentice Hall PTR, 2005.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vissedes, *Design Patterns: Elements of Reusable Object-Oriented Software (Google eBook)*. Pearson Education, 1995.
- [20] N. Ali, J. Rosik, and J. Buckley, “Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study,” *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, pp. 23–32, 2012.
- [21] E. Woods and N. Rozanski, “Unifying software architecture with its implementation,” in *Proceedings of the Fourth European Conference on Software Architecture*, 2010, pp. 55–58.
- [22] R. Terra and M. Valente, “A dependency constraint language to manage object oriented software architectures,” *Software: Practice and Experience*, no. June, pp. 1073–1094, 2009.
- [23] S. Sarkar, G. Rama, and R. Shubha, “A method for detecting and measuring architectural layering violations in source code,” in *APSEC*, 2006.
- [24] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

# LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider Massimiliano Di Penta<sup>†</sup>  
 Department of Computer Science, University of Saskatchewan, Canada  
<sup>†</sup>Department of Engineering, University of Sannio, Italy  
 {md.asad, chanchal.roy, kevin.schneider}@usask.ca, dipenta@unisannio.it

**Abstract**—Tracking source code lines between two different versions of a file is a fundamental step for solving a number of important problems in software maintenance such as locating bug introducing changes, tracking code fragments or defects across versions, merging file versions, and software evolution analysis. Although a number of such approaches are available in the literature, their performance is sensitive to the kind and degree of source code changes. There is also a marked lack of study on the effect of change types on source location tracking techniques. In this paper, we propose a language-independent technique, *LHDiff*, for tracking source code lines across versions that leverages *simhash* technique together with heuristics to improve accuracy. We evaluate our approach against state-of-the-art techniques using benchmarks containing different degrees of changes where files are selected from real world applications. We further evaluate *LHDiff* with other techniques using a mutation based analysis to understand how different types of changes affect their performance. The results reveal that our technique is more effective than language-independent approaches and no worse than some language-dependent techniques. In our study *LHDiff* even shows better performance than a state-of-the-art language-dependent approach. In addition, we also discuss limitations of different line tracking techniques including ours and propose future research directions.

**Keywords**-diff; line tracking; lightweight; levenshtein;

## I. INTRODUCTION

Tracking source code across multiple versions of a program is an essential step for solving a number of problems related with multi-version program analysis. For example, consider the problem of locating bug introducing changes. Existing techniques solve this problem by finding the lines that are affected through bug fixes and then trace back those lines to determine their origin. If a bug has been identified in a software system, tracking lines containing the bug in the subsequent versions can help us determine whether the same problem persists in the next versions and if yes, allows developers to fix the problem at ease. Results obtained from a line tracking technique can be aggregated for fine-grained evolutionary analysis. For example, clone evolution analysis requires tracking clone fragments across multiple versions of a software system. Tracking lines of a clone fragment can help us understand how that fragment evolves over time. Such analysis can also guide us in understanding the nature, effects, and reasons for cloning. To support collaboration in software development, annotation or tagging has been used in source code that can facilitate both navigation and coordination [28],

and source location tracking techniques can help manage tags across versions. Software development involves more than just the creation of source code. There are also different kinds of software artifacts that can benefit from mapping lines across versions. Possible applications are, but not limited to, studying when and how requirements are changed or ensuring whether intended changes have been applied to all configuration files.

A number of line tracking techniques can be found in the literature. Reiss [20] listed a number of approaches for tracking lines across versions and found that language-independent approaches often provide good results. Canfora et al. [4] proposed another language-independent line matching technique, called *ldiff*, that uses a combination of information retrieval techniques and Levenshtein distance for mapping lines. Techniques that take into account the syntactic structure of source files can provide change information at a more fine-grain level [9]. An example of a line tracking technique that falls in this group is *SDiff* [27] that can determine changes at method and identifier levels. If fine-grain source code change information is not required, language-independent text-based approaches are suitable for tracking source code lines and can be applied to different kinds of documents besides source code (e.g., test cases or use cases). They also have the potential to be integrated with existing version control systems with little or no modification. However, the performance of these techniques vary depending on the degree of changes applied to the source code. In a recent comparative study of source location tracking techniques, William and Spacco[27] found that techniques that performed well in the Reiss study did not perform well against their benchmark in which files had a higher degree of changes. This motivates us to investigate the reasons behind this discrepancy and to devise a robust language-independent solution.

This paper introduces *LHDiff*, a language-independent technique to track the evolution of source code lines across versions of a software system. The technique uses *Unix diff* between two different versions of a file to determine the set of unchanged lines. To track the remaining lines, it uses a combination of context and content similarity. However, to speed up the mapping process, it first leverages *simhash* technique to determine a list of mapping candidates for each deleted line in the old file. Next, the technique computes similarity scores again, but this time on the source code lines instead of the *simhash* values, to select one line from each set of mapping

candidates. To validate the effectiveness of our language-independent technique, we compare it against state-of-the-art techniques using different benchmarks where the files are collected from real world applications. We further evaluate our technique with other approaches using different types of changes in a mutation based analysis. The experimental results in both cases suggest that the technique is superior to other language-independent approaches, and even often gives better result than the language-dependent technique *SDiff*.

The remainder of the paper is organized as follows. Section II covers previous work related with our study. Section III describes our hybrid line mapping technique. Section IV presents a quantitative evaluation of line tracking techniques using three different benchmarks. In Section V, we describe results of mutation based analysis. Section VI explains some threats to our study and finally, Section VII concludes the paper with future research directions.

## II. RELATED WORK

Tracking source code lines across program versions is crucial to support various maintenance activities and several approaches exist that consider line content, context, abstract syntax tree, edit distance or a combination of these techniques to solve the problem. In general, existing techniques can be divided into two categories: (1) text-based and (2) syntax tree-based. The first group of techniques is purely textual in nature and does not require parsing source files. As a language-independent technique the *Unix diff* algorithm has been widely used in many studies, not only to track lines but also for program differencing. *Diff* is based on solving the problem of longest common subsequence and it reports the minimum number of line changes that can convert one file to another. However, it has its limitations. For example, it cannot detect reordered lines. The addition of comments to a line can cause *diff* to report deletion of the old line and addition of a new line. However, such cosmetic changes are irrelevant to a programmer and should be ignored [13].

*Diff* reports regions of file lines that differ between a pair of files where each region is called a hunk. Zimmermann et al. [32] addressed this modification changes using an annotation graph where large modifications are considered as combined addition and deletion of lines, otherwise all lines between hunk pairs are connected with each other in a modification. The technique detects origins conservatively, does not consider the issue of reordered lines and is susceptible to errors.

Canfora et al. [5], [4] developed a line differencing technique, called *ldiff*, to track line locations independent of languages. Their technique uses the *Unix diff* algorithm to determine the unchanged lines. After that, set-based and sequence-based metrics are used to complete the mapping of remaining lines. However, they only compared the technique with *Unix diff* algorithm. To the best of our knowledge, Reiss was the first to conduct a study to evaluate the performance of several techniques for tracking source locations [20]. Interestingly, the result reveals that simple techniques like the

one mentioned above that do not consider program structure perform better than those that consider syntactic structure of source files (like abstract syntax tree-based techniques). Reiss also recommended the *W\_BESTI\_LINE* technique to track line locations, which uses a combination of context and content similarity to find the evolution of lines independent of languages. While *LHDiff* is also language-independent, our technique differs from the above approaches in that it uses the *simhash* technique to speed up the mapping process and a set of heuristics to improve the effectiveness of tracking source locations.

Spacco and Williams [27] extended the idea of tracking lines for tracking program statements across multiple revisions of Java code. They developed *SDiff*, an abstract syntax tree based technique that leverages tokenization, *Unix diff* and *Kunhn-Munkres* algorithm [17] to complete tracking line locations. *SDiff* is a great algorithm to determine differences at the line level. However, the technique cannot be applied to arbitrary source code and cannot handle comments. While comments are not executed, their importance cannot be ignored. Tags [28] are usually associated with comments to support asynchronous collaboration and they need to be tracked across versions. Moreover, *SDiff* requires that the source code to be parsed without any error. However, in reality this can not be guaranteed. For example, the source code may be written using an old grammar of a language or developers may issue file differencing commands in the middle of an edit operation. *LHDiff* on the contrary is a language-independent technique, can be applied to arbitrary source code, and can track the evolution of comments/tags across versions.

Line location information can be obtained through techniques that determine fine-grain differences between two versions of a file. However, these techniques require knowledge about language constructs. Among these approaches, most notable is the ChangeDistiller [9], which considers the abstract syntax tree of a Java source file and leverages a tree differencing algorithm to determine fine-grain changes in the source code. The algorithm is not immune to cosmetic changes (changes that do not affect the behaviour of a program like addition of a comment), cannot work on arbitrary text files and is limited to Java files only. Xing and Stroulia [31] presented an algorithm, known as UMLdiff to determine structural changes in object-oriented software. It uses a combination of name similarity and structure similarity measures for recognizing conceptually the same entities. Apiwattanapong [2] presented an algorithm to determine changes between two Java programs. The technique considers program structure and semantics of programming language constructs to determine changes that are difficult to detect with a pure textual differencing technique. While these approaches are language specific and focus on fine-grain change details, *LHDiff* focuses on tracking lines independent of source code languages.

Techniques for tracking program elements across versions are also related with our study. Matching higher level language constructs prior to mapping lines improves mapping quality. A comprehensive survey of various techniques can be found in

the work of Kim et al. [16]. Godfrey and Zou [11] developed a semi-automatic technique to detect merging and splitting of source code entities during the evolution of a software system. Kim et al. [15] used a combination of textual similarity and a location overlapping score to track clone fragments across versions. Duala-Ekoko and Robillard developed a technique to track the evolution of clones [8]. The technique is also available as an Eclipse-plugin, called Clone Tracker. Here, clones are identified using an abstract clone region descriptor (CRD) that is independent of source code line locations. While the above techniques focus on tracking code fragments, we focus on tracking individual lines.

### III. LHDIFF: A LANGUAGE-INDEPENDENT HYBRID LINE TRACKING TECHNIQUE

This section introduces *LHDiff*, our language-independent hybrid line tracking technique. Figure 1 summarizes the entire mapping process.

#### A. Preprocess input files

The algorithm starts with reading lines from two different versions of a file. A large number of changes in source code are only cosmetic in nature and do not change the behaviour of a program. Examples include changes to whitespace/newline characters. They are inserted to change the indentation of a program to improve readability. To ignore such changes each line of the source file is normalized so that multiple spaces are replaced by only one. We also remove all parentheses and punctuation symbols from the text except curly braces because we obtained best results when considering them as part of the line context.

#### B. Detect unchanged lines

After preprocessing we apply *Unix diff*, which uses the longest common subsequence algorithm to determine the set of unchanged lines. We use *diff* because previous studies report that it can detect the set of unchanged lines with great accuracy [20]. *Diff* reports the sequences of lines that have been deleted or added between the files. We store the list of deleted and added lines into two different lists. From now on, we refer them as the left and right lists correspondingly.

#### C. Generate Candidate List

Now, our goal is to determine the mapping of a line from the left list to that of the right list. Similar to *W\_BESTI\_LINE* (recall that *W\_BESTI\_LINE* is another language-independent source location tracking technique), we use both context and content of a line to determine the correct mapping. The line itself represents the content and the context is created by concatenating four lines before and after the target line. While the content similarity between a pair of lines is calculated using normalized Levenshtein edit distance which considers the order of characters in them, the content similarity is calculated applying cosine similarity that does not consider the order of tokens/characters. While building context we ignore blank lines, but keep the curly braces. Such changes allow

us to gather sufficient contextual information for a line. We then determine a combined similarity score by considering both content and context similarity. We need to calculate such scores between all possible pairs of deleted and added lines. After that we map only those lines that provide the highest similarity scores and also exceed a predefined threshold value. However, the complete operation would take a long time to complete because of the complexity associated with computing similarity scores, particularly the normalized Levenshtein edit distance. To improve the running time of the algorithm we follow a different strategy. We first apply a form of locality sensitive hashing and calculate the combined similarity score of the hash values instead of the original lines to determine a small set of possible mapping candidates for each line of the left list. Then we use the original line content and combined similarity score to select a line from each set of mapping candidates. Since the hashing technique reduces large data into a much shorter sequence of bits, the overall mapping time is reduced significantly.

While a cryptographic hash function tries to avoid generating the same key to ignore collisions, in this form of hashing files containing similar content are mapped to identical or very similar binary hash keys. The technique is known as *simhash* [7] and it has been found that the technique is practically useful to determine near-duplicate pages in a large collection of documents [18]. The core of the algorithm uses a hash function to generate *simhash* values. Among various non-cryptographic hash functions we use *Jenkin* hash function since it shows better similarity preserving behaviour compared to other functions and also found effective in detecting near-miss code fragments in other studies [1], [29], [30]. We generate a 64 bit *simhash* value for both context and content using the *simhash* algorithm [25]. Instead of working on the original lines, we are now working on the *simhash* values. We now calculate the context and content similarity for each pair of added and deleted lines by calculating the Hamming distance between their corresponding *simhash* values. While the Hamming distance between two strings is the number of substitutions required to convert one string into another, for binary strings ( $a$  and  $b$ ) Hamming distance is calculated by counting the number of 1 bits in  $a \oplus b$  (bitwise exclusive OR operation between  $a$  and  $b$ ). The smaller the Hamming distance is, the closer the two strings are (see Figure 2). The context and content similarity values are normalized between zero and one. A combined similarity score is calculated for each pair of lines using 0.6 times the content similarity and 0.4 times the context similarity (this is determined after experimenting with different combinations of values). For each line on the left list, we then determine  $k$ -neighbours from the right list that are the most probable mapping candidates of that line based on the combined score. We refer this set as the matching candidates list. Since we are not comparing raw source code lines for selecting the mapping candidates, this saves significant time. During our study with different values of  $k$  we found that  $k = 15$  is a good choice to work with.

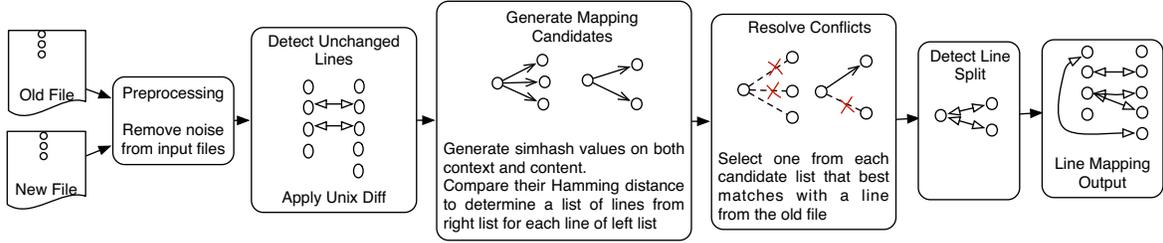


Fig. 1: Summarizing the line mapping process in LHDiff

#### D. Resolve conflicts

We now have a candidate list for each line of the left list (except those that are detected as deleted/unchanged by the algorithm), but we do not know the exact mapping yet. The objective of this step is to select one line from each candidate list to resolve the conflicts. As an example, consider that we are looking for the mapping of line 20, and from the previous step we determine that the candidate list consists of three lines (37, 46, 51). We now use the original lines to generate both context and content, and the algorithm determines the combined similarity score between each possible mapping pairs ( $\{20, 37\}$ ,  $\{20, 46\}$ , and  $\{20, 51\}$ ). It selects the one that gives the highest similarity score and also exceeds a predefined threshold value. We now use normalized Levenshtein distance to measure context similarity and cosine similarity to measure content similarity. Both the values are normalized and the combined similarity score is determined using 0.6 times the content similarity and 0.4 times the context similarity. These were the same values used by Reiss. We set the threshold value to 0.45 after experimenting with various other values because at this setting *LHDiff* provides best result.

#### E. Detect line splits

The last part of our technique deals with detecting line splitting. We use the term line splitting instead of statement splitting since *LHDiff* is not aware of the boundary of a statement, and only works at the line-level. An example of line splitting is shown in the Figure 3 where a single line breaks into multiple small lines. The basic *LHDiff* algorithm tries to map each line from the old file to a line in the new file, but fails to map some lines where the textual similarity differs a lot. To track lines affected by the line splitting we use the following approach. For each unmapped line of the new file, we repeatedly concatenate the line to the successive lines, one at a time, and determine normalized Levenshtein distance (LD) with another unmapped line in the old file until the similarity value starts to decrease. Then, if the textual similarity between the concatenated lines and the left side line crosses a predefined threshold value we map them. As an example, we concatenate line 20 with 21 and determine the normalized Levenshtein distance between  $R_{20+21}$  and  $L_{40}$ . The similarity value is greater than the similarity between  $R_{20}$  and  $L_{40}$ . Thus we concatenate the next line and determine the similarity again between  $R_{20+21+22}$  and  $L_{40}$ . Since the similarity is again greater than the previous step we continue the concatenate operation until the similarity decreases. This happens as soon as we add line 24 ( $LD(R_{20+21+22+23+24}, L_{40}) < LD(R_{20+21+22+23}, L_{40})$ ). This indicates that line 24 cannot be a part of the split lines. Since  $LD(R_{20+21+22+23}, L_{40})$  exceeds the threshold value and there are four lines in the concatenated part, *LHDiff* maps all these four lines to line 40 on the left side. To avoid false mapping we set the threshold value to a high value, 0.85, and we also limit the concatenation to a maximum of 8 lines. During our manual analysis we did not find any example where a line splits more than that. This heuristic approach can only compensate line splitting when such an operation does not change the contents of the line or changes only little. It cannot detect other complex line splitting operations, such as those described in Section V-A.

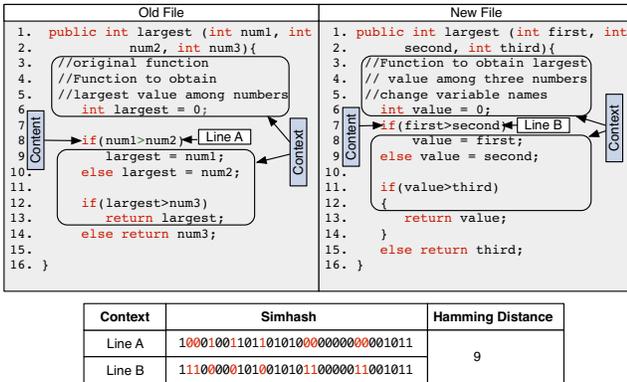


Fig. 2: An example of calculating Hamming distance

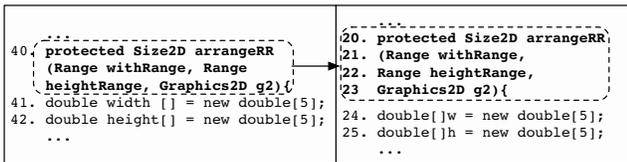


Fig. 3: An example of a line splitting

#### F. Evaluation

We evaluate the effectiveness of *LHDiff* using two different methods. First, we consider three different benchmarks, each of them containing line mapping information of versions of files where the files are collected from real world applications. Second, we also compare *LHDiff* with other state-of-the-art techniques using a mutation based analysis where we consider

different types of changes. We describe details about these two evaluation methods in the following two sections.

#### IV. EVALUATION USING BENCHMARKS

This section describes the benchmarks we used to evaluate source location tracking techniques including results of our evaluation.

##### A. Experiment details

We used two different benchmarks available from previous studies to measure the effectiveness of source location tracking techniques. The first one was developed by Reiss, which contains location information of 53 lines of a file (`ClideDatabaseManager.java`) in 25 revisions and amounts to 1325 test cases [20]. We also evaluated our technique with another benchmark developed by Reiss that contains locations of 14 lines of the `JiveRuntime.java` file in 27 different revisions. However, we did not report the result in this paper because the changes are simple and there is no significant differences among source location tracking techniques for those changes. In both cases, lines were selected by looking at every tenth line of the source file discarding those containing blank lines. Additional problematic or interesting changes (such as name and comment changes) were also included. Williams and Spacco developed another benchmark (known as Eclipse benchmark) containing the change information of 232 lines [27]. We manually analyzed all changes. By doing this we not only validated changes but also identified interesting change patterns during the evolution of these lines. It should be noted that during our manual investigation we found a few incorrect mappings in the Eclipse benchmark. We used the benchmark in our evaluation after correcting those mappings. That is why readers will notice a slight difference in our results for Eclipse benchmark than what was reported in the original paper [27].

In addition, we also developed a third benchmark using source code from the NetBeans project [19]. NetBeans is an integrated development environment for developing applications using different languages. We randomly selected a number of lines (including those we found challenging) and then determined the new locations of those lines in another version. Since the decision is subjective in nature, to avoid bias the first two authors of this paper determined the correct mapping of these lines separately. Cases where there was a disagreement between the two authors, we removed that line from our study.

TABLE I: Three forms of incorrect mappings

Label	Meaning
Change	the algorithm finds a mapping of a line but the mapping is not correct
Spurious	the algorithm detects mapping of a line but the line is deleted
Eliminate	the algorithm detects deletion of a line but the line exists in the new file

Although our technique is language-independent, we considered techniques in both categories for comparison. We instructed *ldiff* to ignore all whitespaces and also to ignore changes whose lines are all blank. For all other settings of *ldiff* we used default values except we changed the number of iterations to four different values. When the value is set to 0, *ldiff* considers only line similarity. For all other positive values it considers hunk similarity. In the case of *SDiff*, we evaluated all different nine configurations used in the original experiment. For details of these configurations we refer interested readers to their original paper [27]. Among various techniques evaluated by Reiss we report the result of *W\_BESTI\_LINE*, with the same similarity thresholds and context length suggested by Reiss, because Reiss recommended this technique for tracking lines. We also include results of *diff*, configured to ignore spacing differences and cases of letters during the mapping process.

To calculate a score for each method, we followed the basic scoring mechanism used in previous studies where the score was calculated by determining the number of correct mappings. To document line mapping information, we used the following approach. If a line in the old file is deleted, the new location of that line is encoded with  $-1$ , otherwise each line of an old file is associated with a non-negative integer representing the new position of that line. Incorrect classifications can result from three different types of errors and we use different labels to signify them as shown in Table I. All experiments were conducted on a computer running with Ubuntu Linux that has a 2.93 GHz Intel Core i7 processor and 8 GB of memory.

##### B. Results

Table II shows the results of our evaluation. For each benchmark and line tracking technique we not only provide the percentage of correct mappings but also show results for each incorrect mapping type (these are: change, spurious and eliminate; see Table I for their definitions). From the table we can see that *LHDiff* performs better than both *SDiff* and *ldiff*. The default settings of *ldiff* uses cosine similarity for mapping hunks and Levenshtein distance for mapping lines. We considered all possible combinations of metrics and tokenizers. While for the Reiss benchmark *ldiff* correctly maps around 96.5% of lines, the performance drops significantly for the Eclipse benchmark. This is similar to the result provided by Spacco and Williams. On the other hand, although *W\_BESTI\_LINE* performs similar to *LHDiff* for the Reiss benchmark, accuracy drops to around 53% for the Eclipse benchmark. The latter contains a large number of changes as the files are heavily edited in it. The performance of *W\_BESTI\_LINE* and *ldiff* varies depending on the number of changes occurred to the files. However, both *SDiff* and *LHDiff* are stable in nature. Although *SDiff* shows around 87% accuracy for the Reiss benchmark, according to the authors accuracy can be even up to 96% if we ignore curly braces and non executable statements. For the Eclipse benchmark, *SDiff* produces around 74% accurate result. However, *SDiff*

TABLE II: Running location tracking techniques against Reiss, Eclipse, and NetBeans benchmarks

Type	Method	Reiss Benchmark					Eclipse Benchmark					NetBeans Benchmark				
		Correct %	Incorrect Mapping Categories			Time [s]	Correct %	Incorrect Mapping Categories			Time [s]	Correct %	Incorrect Mapping Categories			Time [s]
			Change %	Spurious %	Eliminate %			Change %	Spurious %	Eliminate %			Change %	Spurious %	Eliminate %	
Language independent Techniques (Text-based approach)	W_BESTI_LINE	96.7	0	29.6	70.5	4.8	52.6	28.2	56.4	15.5	0.7	61.9	41.9	45.2	12.9	2.9
	LHDiff	<b>97.0</b>	<b>42.5</b>	<b>0</b>	<b>57.5</b>	<b>4</b>	<b>82.8</b>	<b>22.5</b>	<b>20</b>	<b>57.5</b>	<b>3.3</b>	<b>85.5</b>	<b>18.6</b>	<b>13.6</b>	<b>67.8</b>	<b>6.8</b>
	diff	92.5	0	0	100	0.2	41.0	0.7	0	99.3	0.1	48.4	1.4	0	98.6	0.2
	ldiff [-i 0]	96.4	0	66.7	33.3	25.7	62.5	9.2	1.2	89.7	58.2	74.0	16.0	7.6	76.4	6.1
	ldiff [-i 1]	96.4	0	25	75	73.9	59.1	9.5	1.1	89.5	209.2	76.4	14.6	5.2	80.2	103.5
	ldiff [-i 3]	96.2	0	29.4	70.6	118.6	72.0	20	4.7	75.4	419.7	80.6	24.1	8.9	67.1	171.5
	ldiff [-i 5]	96.2	0	29.4	70.6	160.7	72.4	20.3	4.7	75	472.4	81.3	25	9.2	65.8	218.2
	Git	92.5	0	0	100	0.8	45.3	0.8	0	99.2	0.3	53.7	1.6	0.5	97.9	0.2
Language Dependent Technique (Requires access to abstract syntax tree)	SDiff	86.2	0	86.3	13.7	1.9	70.7	11.8	80.9	7.4	1.7	71.8	7.0	86.1	7.0	4.2
	SDiff-probable	87.0	3.0	82.4	14.6	1.7	69.9	11.4	75.7	12.9	1.5	73.5	9.3	82.4	8.3	4.3
	SDiff-possible-probable	87.0	3.0	82.4	14.6	1.7	70.7	11.8	75	13.2	1.6	75.4	12	77	11	4.3
	SDiff-min	85.6	0	82.5	17.5	2.6	74.1	13.3	76.7	10	2	72.5	7.1	85.7	7.1	6.8
	SDiff-min-probable	86.4	2.9	78.6	18.5	2.6	73.3	12.9	74.2	12.9	2.1	74.2	10.5	81.0	8.6	6.8
	SDiff-min-possible-probable	86.4	2.9	78.6	18.5	2.6	74.1	13.3	73.3	13.3	2.1	75.2	12.9	74.3	12.9	6.8
	SDiff-token	85.6	0	82.5	17.4	1.0	73.3	11.3	79.0	9.7	1.0	71.0	6.8	87.3	5.9	2.3
	SDiff-token-probable	85.6	2.7	79.8	17.5	1.0	73.3	11.3	75.8	12.9	1.0	73.7	10.3	82.2	7.5	2.3
SDiff-token-possible-probable	86.4	2.9	78.7	18.5	1.0	74.1	11.7	75	13.3	1.0	74.5	11.5	76.9	11.5	2.3	

takes into account the structure of source files, requires the files to be syntactically valid, and is only available for Java. *LHDiff*, on the other hand, is purely textual in nature and can be applied to any files (be it a source file or a plain text file). Without considering structural information, *LHDiff* provides around 97% of correct mappings for the Reiss benchmark and for the Eclipse one the result is around 83%. Among the incorrect mappings we found for the Eclipse benchmark, the highest amounts (more than 57%) were due to elimination.

For the NetBeans benchmark (see Table II), *LHDiff* shows better result than the other techniques. While *LHDiff* detects around 86% correct mapping, *SDiff* detects only around 75%. Although *ldiff* detects more correct mappings (81%) than *SDiff*, it requires more computation time.

### C. Discussion

In this section we first explain the reasons behind the poor results of *W\_BESTI\_LINE* and also *ldiff* on the Eclipse benchmark that originally motivated us to develop another language-independent source location tracking technique. We then compare *LHDiff* with the content tracking technique of *Git* and present another study result where we tried to improve the accuracy of *LHDiff* using tokenization.

1) *Why technique recommended by Reiss or ldiff did not perform well with Eclipse benchmark?:* The algorithm (*W\_BESTI\_LINE*) recommended by Reiss works fine as long as the context and content of a line do not change significantly. The technique fails when both of them go through a large amount of changes. Another possible threat to this approach is the addition of blank lines or lines containing stop list or punctuation symbols only. If a developer inserts blank lines before and/or after a line, the line becomes isolated and the context differs remarkably. If the line also changes

significantly, the algorithm fails to map the old line to the new one. Reiss did not consider this issue. However, we can eliminate this problem by ignoring blank lines which can help locate proper contextual information. Another downside of this technique is that the cost of running the technique is high. For a line in the old version, the technique compares the line with every other lines in the new version and selects the one that provides the best matching. If the objective is to determine new locations of a few lines of the old file, then the technique may be adequate. However, situations where we need to map every line of an old file to the new file, then the approach may be expensive particularly when the size of the file is large. That is why we use the *simhash* technique in *LHDiff* to faster the mapping process. *W\_BESTI\_LINE* tries to map a subset of lines of an old file to the new file whose mappings are requested by a user. While this approach lessens the running time of the algorithm, it drops the accuracy of the technique, particularly where files have gone through a large number of changes. A line ( $l_i^{old} \in f_{old}$ ) cannot be aggressively mapped to another line ( $l_j^{new} \in f_{new}$ ) without considering the similarity of  $l_j^{new}$  to all other lines of the old file. Their might be another line ( $l_k^{old} \in f_{old}$ ) to which the newly mapped line ( $l_j^{new} \in f_{new}$ ) best matches. For these reasons *W\_BESTI\_LINE* performed poorly for the Eclipse benchmark.

The running time of *ldiff* improves because of applying *Unix diff* at the beginning to determine unchanged lines. A threat to the technique comes from the fact that before mapping lines *ldiff* tries to map a hunk from the old file to another hunk of the new file. If the hunk similarity exceeds a threshold value then a line mapping process begins. However, there is a possibility that hunk similarity does not exceed the threshold value because of their size difference and only a few

TABLE III: Results of LHDiff before and after applying tokenization

Benchmark	LHDiff	Correct%	Change%	Spurious%	Eliminate%
Reiss	Non-tokenize	96.98	42.50	0	57.50
	Tokenize	93.66	0	48.81	51.19
Eclipse	Non-tokenize	82.76	22.50	20	57.50
	Tokenize	82.76	47.50	32.50	20
NetBeans	Non-tokenize	85.50	18.64	13.56	67.80
	Tokenize	82.56	32.39	9.86	57.75

lines are common between the hunks. In such a situation *ldiff* reports them as deleted and added lines which contributed to its poor performance for the Eclipse benchmark.

2) *How is LHDiff comparable to the content tracking technique of Git?:* Some version control systems (like CVS or SVN) change code authorship of a line even if the line goes through formatting changes or moves to a different location. In both cases, the line is reported as a new one. Git overcomes such a limitation by tracking the content of a file across versions. Git *blame -C -M* command can track the movement of unchanged lines. Here, *-C* finds code copies and *-M* finds code movement. However, if the lines move even with slight changes, Git assigns code authorship to the new author and marks them as newly created, although ideally these lines come from different locations of the previous version. To compare the content tracking technique of Git with *LHDiff*, we first commit each pair (the original one and its new version) of file versions of a benchmark in a local repository and then apply the *git blame -M -n* to determine the origin of line locations of the new file. Here, *-n* tells Git to show the line numbers in the original commit which is by default turned off. The results are summarized in Table II. In general, the mapping accuracy is similar to diff, except it detects more correct mappings for some cases where lines are moved within files (such as encapsulation, function split and reordering categories).

Besides *blame*, Git also offers *pickaxe* to dig deeper into the history. While it can find the commits that change a block of code in a file, it cannot find the list of commits that contain the block of code. In general, Git focuses more on tracking code blocks instead of individual lines and stays out from the advanced diff or line tracking technique, possibly because of eliminating the overhead of running such an algorithm. Thus, the line level content tracking technique of Git is not as powerful as *ldiff* [3] or *LHDiff*.

3) *Can tokenization improve the performance of LHDiff?:* Source code can go through various cosmetic changes that can affect performance of text-based source location tracking techniques. For example, the order in which parameters appear in the method definition can be changed. In object-oriented programming languages, developers often use *this* keyword to access object fields or methods which does not change the behaviour of the program. The access modifier of a class

Fig. 4: Example of tokenization

can change from private to public to make a class visible to all other classes. Handling these changes requires accessing individual source code elements and in this regard tokenization can assist us.

Tokenization is the process of converting a sequence of characters of a source file into a set of tokens, where each token is a string of characters representing a category of symbols. Tokenization does not require parsing source files and can be implemented using regular expressions. We anticipate that instead of working on raw source files, working on the tokens may help us to ignore the effect of source code changes and thus, improve the accuracy of the algorithm. To verify this, we first tokenize source files of our benchmarks according to the lexical rules of the Java programming language. During this process we keep track of the line locations of each token so that we can construct source files with tokens later. Next, we apply a set of transformation rules to normalize token sequences (an example is shown in Figure 4). This step is necessary to eliminate differences between source code versions. We then reconstruct source files using transformed token sequences and use *LHDiff* to track source code lines across file versions.

Working on the tokens does not improve the performance of the *LHDiff* algorithm. While for the Eclipse benchmark tokenization does not change mapping quality, performance drops by 3% for the NetBeans benchmark. When we examined those lines that were not correctly mapped by *LHDiff*, we found that working on the tokens eliminates differences between file versions. While for some cases it helps to correctly map lines that were earlier detected as deleted (the percentage of incorrect mapping for *eliminate* category drops from 67.8% to 57.75% for the NetBeans benchmark) but for some cases it leads to false mapping too (the percentage of spurious mapping increases from 18.64% to 32.39%). We observe similar picture for other benchmarks also. Though tokenization does not require parsing of source code, it requires knowledge of the tokens of programming languages. Since our objective is to develop a language-independent solution, *LHDiff* does not include tokenization as a part of the detection process. Moreover, our study results reveal that even tokenization can lead to poor result.

Although the above results show the performance of *LHDiff* over other methods, it does not explain how source code changes affect line tracking techniques. To find this out, we use a mutation based analysis that considers the editing taxonomy of source code changes as describe in Section V-A.

## V. EVALUATION USING MUTATION BASED ANALYSIS

While previous studies evaluate source location tracking techniques against manually verified line mapping data, there

TABLE IV: Editing taxonomy of source code changes

No	Name	Example		Description
		Old File	New File	
1	Line Splitting/Merging	<code>else if (list.get(i)%3 == 0)</code>	<code>else if (list.get(i)%3 == 0)</code>	An <i>else if</i> statement splits into two lines
		<code>if (list.get(i)%3 == 0) sum += list.get(i);</code>	<code>if (list.get(i)%3 == 0){ System.out.println("i = " + i); sum += list.get(i) }</code>	A statement in one line splits into multiple lines where each contains one statement and some lines are added in between them.
		<code>list.add(Integer.parseInt(line));</code>	<code>number = Integer.parseInt(line); System.out.println(number); list.add(number);</code>	Another example of line splitting of the above kind.
2	Function Splitting/Merging	<code>public int sum(File file){ //Read numbers from the file //Store them in a list ... //Now process the list //Calculate sum of the numbers ... return sum; }</code>	<code>public ArrayList readFile(File file){ //Read numbers from the file //Store them in a list ... return numberList } public int sum(File file){ ArrayList list = readFile(file) //Now process the list //Calculate sum of the numbers ... return sum; }</code>	A developer creates a function <i>readFile</i> with some lines from <i>sum</i> and replace those lines in <i>sum</i> by adding a call to that function in the new version of <i>sum</i> or vice versa
3	Wrapping/Unwrapping	<code>while((line = br.readLine()) != null)</code>	<code>try { while((line = br.readLine()) != null) } catch (IOException e){ e.printStackTrace() }</code>	A line in the old file moves to a try-catch block in the new version of the file or vice versa
4	Change in Data Structure	<code>ArrayList list = new ArrayList();</code>	<code>LinkedList list = new LinkedList();</code>	<i>ArrayList</i> data structure is replaced by <i>LinkedList</i> or vice versa
5	Renaming	<code>sum = sum + list.get(i); return sum;</code>	<code>total = total + list.get(i); return total;</code>	Variable <i>sum</i> is renamed to <i>total</i> or vice versa
6	Code Reordering	<code>public ArrayList readFile (File file) { ... } public int sum (File file) { ... }</code>	<code>public int sum (File file) { ... } public ArrayList readFile (File file) { ... }</code>	Functions <i>sum</i> and <i>readFile</i> are reordered or vice versa

is not much discussion about the types of changes appearing in them. This opens the question of how stable/vulnerable the techniques are to different change groups and makes it difficult to compare them in an objective fashion. Instead of a random selection of lines from source code and tracking their positions in subsequent versions, we use a taxonomy of source code changes to synthesize new lines and evaluate the techniques objectively. Towards this goal, in this section we first present an editing taxonomy that captures typical actions performed by developers during source code evolution and then use a mutation based analysis to evaluate line tracking techniques based on the taxonomy following Roy and Cordy's [22], [21] mutation analysis framework for evaluating software clone detectors [23].

#### A. Editing taxonomy of source code changes

We developed an editing taxonomy by studying a large body of published work [10], [11], [16], [14], clone taxonomy [21], our previous study on code clones [24], and through manual investigation of the previous benchmarks [27], [20]. We further refined our taxonomy by analyzing changes of two open source

Java systems (NetBeans [19] and iText [12]). We choose Java because of our familiarity with this programming language which helped us to determine correct mapping of lines with less confusion. Presenting the frequency of the changes in software systems is out of the scope of this paper.

Table IV shows our editing taxonomy of source code changes. For each change type we also provide an example that shows the change from the old version of a file to the new version of that file. The edit operations describe in the taxonomy are not mutually exclusive. Instead, they can be applied together to create more complex changes. To save space, simple edits (addition, deletion or modification of lines) are not discussed although they are the building blocks of all kinds of edits.

The first group of change in our taxonomy is line splitting/merging. Use of code formatters can trigger line splitting to improve readability of a source code. Table IV shows three different kinds of line splitting examples. We do not show any example of line merging since it is the opposite action of line splitting. Function merging/splitting is the second change type in our taxonomy. While Merging is done for service

TABLE V: Results of mutation based evaluation

Method	Change in Data Structure	Wrapping	Line Split	Function Split	Renaming	Reordering
W_BESTI_LINE	18.8	86.7	4.3	93.4	17.9	82.7
<b>LHDiff</b>	<b>56.3</b>	<b>90</b>	<b>68.1</b>	<b>93.4</b>	<b>35</b>	<b>83.8</b>
Unix diff	6.3	80	2.1	55.7	0	13.5
Git	6.3	80	2.1	85.3	0	66.2
ldiff [-i 0]	12.3	80	40.1	55.7	42.5	13.5
ldiff [-i 1]	6.3	90	44.7	86.9	35	52.7
ldiff [-i 3]	6.3	90	40.7	95.1	35	78.4
ldiff [-i 5]	6.3	90	40.7	95.1	35	82.4
SDiff	81.3	80	51.1	10	10	82.4
SDiff-probable	93.8	80	51.1	10	10	82.4
SDiff-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-min	87.5	80	51.1	15	15	82.4
SDiff-min-probable	93.8	80	51.1	15	15	82.4
SDiff-min-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-token	87.5	80	51.1	15	15	82.4
SDiff-token-probable	93.8	80	51.1	15	15	82.4
SDiff-token-possible-probable	93.8	80	51.1	15	15	82.4

consolidation or code clone elimination, it can be split also. The term Wrapping refers to a change where an old line is moved inside a block of code and we define the opposite action as unwrapping. Wrapping makes a line difficult to trace even when we try with contextual information since the context may differ and can be worse if the line itself changes significantly. In some cases, one form of wrapping can be changed into another. For example, the line attached with an *if* statement can be moved to the *else* part. We found scenarios where data structures used in previous versions are replaced with other kinds. For example, a HashTable can be replaced by a Map or an ArrayList implementation can be replaced by a LinkedList (see Table IV(4)). Changes of this kind may or may not preserve the textual similarity of lines and pose a threat to line mapping techniques. Changes in identifier names are common in source code evolution, particularly where copy-paste programming is involved. Location tracking techniques in general perform well when identifiers use descriptive names. Other language constructs such as function, method or class name can be changed. The last group of change type in our taxonomy is code reordering where functions or blocks of code can be reordered.

### B. Experiment Details

In the generation phase, we mutate source files. First, we select a source file in version  $v_i$ . We then make another copy of that source file. The mutation is carried out either by injecting or changing existing code fragments in the copied file in such a way that considers several possible change scenarios listed in our taxonomy of source code changes. To avoid bias in comparison, we did not consider mapping of comments or curly braces in our analysis since *SDiff* ignores mapping them.

### C. Results

The results of our mutation based analysis is summarized in Table V. In most of the change groups, *LHDiff* either

outperforms other techniques or performs very close to the best technique with an exception in the data structure change category. In that case, accuracy drops to 56.3% whereas *SDiff* correctly detects around 93% mappings. When we investigate the reasons, it reveals that both context and content changes significantly which leads to such poor result.

*W\_BESTI\_LINE* suffers from the problem of detecting boundary of language constructs (such as a statement) and thus is immune to line splitting. *SDiff* operates at the statement level and can detect line splitting, but the accuracy is variable. For example, cases where lines are added in between of the split lines (see Table IV(1), second example), *SDiff* cannot determine correct mappings. In our mutation based analysis *LHDiff* performs not too bad in detecting line splitting. Manual investigation reveals that even in the case of line splitting, they do share some degree of similarity with the original line.

While *W\_BESTI\_LINE* and *ldiff* work at line level, they can track movement of lines because of function splitting/merging. *SDiff* is affected by function or method splitting and cannot detect a block of lines that is moved to other functions. Thus, *SDiff* reports them as deleted. For instance, *SDiff* cannot track lines that are moved from function *sum* to *readFile* (see Table IV(2)) and report them as deleted. The reason is that *SDiff* first tries to map functions and if it finds a mapping, it tries to map the lines. For this example, *SDiff* finds a mapping between *sum<sub>old</sub>* and *sum<sub>new</sub>* and then maps their lines without considering the fact that lines of the *sum* function can be moved to other functions also.

Renaming of variables, functions or classes also affect line tracking techniques and *ldiff*, *LHDiff* or *W\_BESTI\_LINE* may or may not map the line containing function declaration depending on the amount of changes occurred in the file. However, *SDiff* uses an origin analysis technique [14] to map functions across versions and thus can determine renaming of functions. In case of reordering, both *LHDiff* and *W\_BESTI\_LINE* show good performance. As expected *Unix diff* cannot detect any reordering since reordering of lines causes *Unix diff* to report them as addition and deletion of lines. While *SDiff* shows 82.4% accuracy in detecting changes via reordering, *Unix diff* become the last.

## VI. THREATS TO VALIDITY

There are a number of threats to the validity of this study. In this section we discuss them in brief.

First, the mapping of a line is subjective in nature and we cannot guarantee that there is no incorrect mapping in our benchmarks. However, we tried to minimize the number of false mapping as much as possible. The first two authors of this paper manually investigated all mappings including those found in the Reiss and Eclipse benchmarks. Cases where it was difficult to correctly map a line or there was a disagreement, the mapping was removed to avoid ambiguity. Second, considering the small sample size of the benchmarks one can argue that the sample may not represent the population and thus the performance observed in our study does not reflect the original scenario. However, finding changes of lines across versions

and determining their correctness is a time consuming task. We carefully validated the correctness of existing benchmarks and developed a new benchmark containing different types of changes. We also used mutation based analysis to evaluate our technique with other approaches. Third, lines can be changed in various ways. In this paper we describe various kinds of changes that can affect the evolution of a line. Although we cannot guarantee that benchmarks contain all change types, we tried to minimize the effect of change types on their evaluation through collecting line change data at random. Finally, we penalize line tracking techniques with the same weight for detecting different false mapping types (spurious, change and eliminate). While one can argue about weighting false mapping types, we want to highlight to the fact that this is the same scoring scheme used in previous studies [20] and we followed the same strategy to make the result comparable with others.

## VII. CONCLUSION

This paper proposes a novel, language-independent line tracking approach called *LHDiff*. Our experiment shows that lines can effectively be tracked across versions with *LHDiff*. We not only evaluate *LHDiff* with benchmarks created from real world applications but also use a mutation based analysis to evaluate it with other line tracking techniques against different types of source code changes. The results reveal that *LHDiff* is more effective than any other language-independent line tracking technique. We also compared *LHDiff* with *SDiff* which is a state-of-the-art language-dependent technique and found that in most cases *LHDiff* provides better result than *SDiff*. The mutation based analysis also enables us to explain the strength and weaknesses of line tracking techniques and can help a user to decide when to use which technique. Although *LHDiff* incorporates some features from *W\_BESTI\_LINE*, another simple line tracking technique developed by Reiss, but the potential of that technique is not fully explored in the early work, possibly because that work focused on comparing a large number of techniques and the benchmark used in that experiment contains small changes. *LHDiff* is reasonably fast (Comparable in speed to *SDiff* also), requires a small amount of memory, can easily be incorporated into source code control systems, and can be used with arbitrary text files. For future study, we plan to identify the degree of structural knowledge required to reduce incorrect mappings. The current implementation of *LHDiff* only uses information available within files and we would like to explore whether information stored within source code control systems can assist us mapping lines. While this work focuses on tracking lines, visualizing this information poses another challenge that we also want to address. The code of *LHDiff*, data files used in this experiment, and complete evaluation results can be found online [26].

**Acknowledgements:** We would like to thank Steven P. Reiss for giving us an implementation of *W\_BESTI\_LINE* and data files used in his experiment. Jaime Spacco provided useful comments on *SDiff* and supported our study by providing both code and data files.

## REFERENCES

- [1] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting Clones across Microsoft .NET Programming Languages", in Proc. WCRE, pp. 405-414, 2012.
- [2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs", *Automated Software Engg.*, vol. 14, no. 1, pp. 336, 2007.
- [3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git", in Proc. MSR, pp. 1-10, 2009.
- [4] G. Canfora, C. Luigi, and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach", in *IEEE Softw.*, pp. 50-57, 2009.
- [5] G. Canfora, L. Cerulo, M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories", in Proc. MSR, pp.14, 2007.
- [6] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool", in Proc. ICSE, pp. 595-598, 2009.
- [7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in Proc. STOC, pp. 380-388, 2002.
- [8] E. Duala-Ekoko, M. P. Robillard, "Tracking Code Clones in Evolving Software", in Proc. ICSE, pp. 158-167, 2007
- [9] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction", in *IEEE Trans. Softw. Eng.*, pp. 725-743, 2007.
- [10] B. Fluri and H. Gall, "Classifying Change Types for Qualifying Change Couplings", in Proc. ICPC, pp. 35-45, 2007.
- [11] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", *IEEE Transactions on Software Engineering*, v.31 n.2, pp.166-181, 2005.
- [12] "The iText", <http://www.sourceforge.net/projects/itext>
- [13] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories", in Proc. ICSE, pp. 351-360, 2011.
- [14] S. Kim , K. Pan , E. J. Jr. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships", in Proc. WCRE, pp. 143-152, 2005.
- [15] M. Kim, V. Sazawal, D. Notkin, G. Murphy, "An empirical study of code clone genealogies", in Proc. ESEC/FSE, pp. 187-196, 2005.
- [16] M. Kim and D. Notkin, "Program element matching for multi-version program analyses", in Proc. MSR, pp.58-64, 2006.
- [17] H. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quaterly*, pp. 2:83-97, 1955.
- [18] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting Near Duplicates for Web Crawling", in Proc. WWW, pp. 141-150, 2007.
- [19] "The NetBeans", <http://www.netbeans.org>
- [20] S. P. Reiss, "Tracking source locations", in Proc. ICSE, pp. 11-20, 2008.
- [21] C. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools", in Proc. ICSTW, pp. 157-166, 2009.
- [22] C. K. Roy and J. R. Cordy, "Towards a Mutation-Based Automatic Framework for Evaluating Code Clone Detection Tools", in Proc. C3S2E, pp. 137-140, 2008.
- [23] C. K. Roy, "Detection and Analysis of Near-Miss Software Clones", in Proc. ICSM, pp. 447-450, 2009.
- [24] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study", in Proc. SCAM, pp. 87-96, 2010.
- [25] "The Simhash Algorithm", <http://d3s.mff.cuni.cz/~holub/sw/shash/>
- [26] "Source code and data", <http://asaduzzamanparvez.wordpress.com/Research>
- [27] J. Spacco and C. Williams, "Lightweight Techniques for Tracking Unique Program Statements", in Proc. SCAM, pp. 99-108, 2009.
- [28] M. Storey, L. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development", in Proc. CSCW, pp. 195-198, 2009.
- [29] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems", in Proc. WCRE, pp. 13-22, 2011.
- [30] M. S. Uddin, C. K. Roy, and K. A. Schneider, "SimCad : An Extensible and Faster Clone Detection Tool for Large Scale Software Systems", in Proc. ICPC, pp. 236-238, 2013.
- [31] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing", in Proc. ASE, pp. 54-65, 2005.
- [32] T. Zimmerman, S. Kim, A. Zeller, and E. J. Jr. Whitehead, "Mining Version Archives for co-changed lines", in Proc. MSR, pp. 72-75, 2006.

# Mining Software Profile across Multiple Repositories for Hierarchical Categorization

Tao Wang\*, Huaimin Wang\*, Gang Yin\*, Charles X. Ling<sup>†</sup>, Xiang Li\* and Peng Zou<sup>‡</sup>

\*National Laboratory for Parallel and Distributed Processing,

College of Computer, National University of Defense Technology, Changsha, 410073, China

taowang.2005@gmail.com, whm\_w@163.com, {jack.nudt,shockleylee}@gmail.com

<sup>†</sup>Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7

cling@csd.uwo.ca

<sup>‡</sup>Academy of Equipment, Beijing, 100000, China

zpeng@nudt.edu.cn

**Abstract**—The large amounts of software repositories over the Internet are fundamentally changing the traditional paradigms of software maintenance. Efficient categorization of the massive projects for retrieving the relevant software in these repositories is of vital importance for Internet-based maintenance tasks such as solution searching, best practices learning and so on. Many previous works have been conducted on software categorization by mining source code or byte code, which are only verified on relatively small collections of projects with coarse-grained categories or clusters. However, Internet-based software maintenance requires finer-grained, more scalable and language-independent categorization approaches.

In this paper, we propose a novel approach to hierarchically categorize software projects based on their online profiles across multiple repositories. We design a SVM-based categorization framework to classify the massive number of software hierarchically. To improve the categorization performance, we aggregate different types of profile attributes from multiple repositories and design a weighted combination strategy which assigns greater weights to more important attributes. Extensive experiments are carried out on more than 18,000 projects across three repositories. The results show that our approach achieves significant improvements by using weighted combination, and the overall precision, recall and F-Measure can reach 71.41%, 65.60% and 68.38% in appropriate settings. Compared to the previous work, our approach presents competitive results with 123 finer-grained and multi-layered categories. In contrast to those using source code or byte code, our approach is more effective for large-scale and language-independent software categorization.

**Keywords**—Software Repository; Software Profile; Hierarchical Categorization

## I. INTRODUCTION

The Internet-based software online repositories such as SourceForge, Ohloh and RubyForge<sup>1</sup> hold large amounts of software projects, which create considerable opportunities for software engineering and are fundamentally changing the traditional paradigms of software maintenance. Traditional software maintenance tasks such as correcting discovered faults, adapting the system to changes in the environment, and improving the systems reliability and performance [1] are often confined to closed development teams or organizations and carried out in-house by using their internal repositories. With

the rapid growth of the freely-accessible repositories over the Internet, software maintenance is becoming a kind of global collaborative process, and the software developers increasingly resort to these repositories for solutions or best practices. For example, they often search the repositories to find reusable components [2], [3], discover new technical trends in the domain [4], [5], learn solutions from related systems [6], [7], predict and correct faults by analysing similar projects [8], or discover counterparts for help [9]. Efficient retrieval of the desired ones among the massive projects is of vital importance for facilitating the utilization of them.

Hierarchical categorization is considered to be an efficient way for retrieving useful information from large scale data repositories [10]. Firstly, it provides a uniform hierarchical categorization for organizing the huge amounts of software over the Internet. It clusters these projects according to their topics and is quite useful for browsing and retrieving resources with similar functions. Secondly, by combining keyword-based search with hierarchical categorization users can locate desired resources more efficiently and accurately. By refining the query results with specific category, it can filter out those irrelevant projects and return the most relevant ones. For example, when querying with “configuration” in SourceForge, more than 7,000 projects are returned. After refining the results with category “Database Engines/Servers”, the results will reduce to about 25. This remarkably reduce the time for user to find the desired resources. Such a mechanism has been adopted in the popular and successful software repositories like Sourceforge and RubyForge to retrieve the large amounts of resources. However, the software projects in these repositories are categorized manually by software managers, and a large proportion of them are not categorized. In addition, a lot of repositories like Ohloh and Freecode are not categorized at all.

Many researches have been conducted on automatic software categorization, most of which rely on analyzing software programs. In these works, [11], [12], [13], [14] focus on analyzing identifiers and comment terms in the source code. They parse the source code to get corresponding attributes and then apply textual classification approaches to do categorization. Differently, McMillan et.al [15], [16] propose a brand-new approach which leverages the third-party API calls in the program as semantic anchors to categorize software. As API calls can be extracted from both source code and byte code,

<sup>1</sup><http://sourceforge.net>, <http://www.ohloh.net>, <http://rubyforge.org/>

their works solve the problem of categorizing software whose source code are not available.

In these works, most of them only experiment on relatively small collections of projects with flat and coarse-grained categories like “Internet” and “Games/Entertainment”. Such coarse-grained categories are not sufficient enough for retrieving the most related ones among the huge amounts of projects in the repositories. For example, in SourceForge there are more than 35,350 projects under the category “Internet”, which are too many for user to give a quick and feasible choice. Thus, finer-grained categorization is urgently needed. However, considering the large amounts and the complexity of the projects in these repositories (e.g. there are more than 400,000 projects in Ohloh which are programmed in more than 100 different languages), efficient and automatic categorization of these large-scale repositories is quite a challenge problem.

In this paper we propose a hierarchical categorization approach which leverages the software online profiles as source information to do categorization. Online software repositories often curate a profile for each project, which summarizes the resource. A profile mainly consists of several types of attributes, i.e., a piece of description, a set of tags and so on. The profiles cover the functional or technical aspects of the software, which make them effective and alternative source information for categorization. Taking the popular database management system *MySQL* as an example, it is described and categorized/tagged as shown in Table I and Table II in different repositories.

TABLE I. DESCRIPTIONS OF MYSQL IN DIFFERENT REPOSITORIES

Repository	Software Description
Sourceforge	<i>MySQL is a well-known relational database manager used in a wide variety of systems, including . . . MySQL is a good choice for any situation requiring a database.</i>
Ohloh	<i>MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by Oracle Corporation.</i>
Freecode	<i>MySQL is a widely used and fast SQL database server. It is a client/server implementation that consists of a server daemon (mysqld) and many different client programs/libraries.</i>

TABLE II. CATEGORIES/TAGS OF MYSQL IN DIFFERENT REPOSITORIES

Repository	Software Categories/Tags
SourceForge	Database Engines/Servers
Ohloh	db, acid, database_server, rdms, database, dbi, jdbc, transactions, sql, software_development, mysql, dbms, . . .
Freecode	Database, Database Engines/Servers

The two types of profile attributes cover the important features of *MySQL* as shown in the Table I and II. Different from the identifiers or APIs in source code which usually reflect the detailed and implementation-level features of classes or packages, software profiles emphasize on the high-level functional or technical features of the whole software. These online profiles are given in natural language and widely used as normal software documents in repositories, they are more attainable and can be used for categorization regardless of the software programming languages.

Our approach first constructs a category hierarchy which contains more than 120 categories organized in four levels based on the predefined categories in SourceForge. Then we

design an SVM-based (Support Vector Machine) categorization approach, which classifies software hierarchically based on the software online profiles. We present the preliminary results and further analysis on how to improve the categorization performance by profile aggregation. Specifically, we design a weighted combination strategy to assign greater weights to important profile attributes and significantly improve the categorization performance. The main contributions of this paper include:

- We propose a hierarchical software categorization framework. It classifies software into multi-grained categories hierarchically. To the best of our knowledge, this is the first work on automatic software categorization hierarchically and the number of categories is significantly enlarged compared to previous works.
- We explore the multiple types of attributes in software online profiles for categorization and design an efficient combination strategy to aggregate them from multiple repositories. Such web-based software data is less studied for categorization in the previous works.
- We conduct extensive experiments on more than 18,000 software. The experiments show promising results for hierarchical categorization and prove the effectiveness and scalability of our method.

The reminder of this paper is organized as follows. Section II discusses the related works on software categorization and profile mining. Section III describes the hierarchical categorization approach in detail. Section IV presents the experiment questions and settings and Section V evaluates our approach. We discuss the validity of our work in Section VI, summarize the paper and discuss the future work in Section VII.

## II. REVIEWS OF PREVIOUS WORKS

In this section, we review the previous works on software categorizations and software profiles mining.

### A. Software Categorization

Many works have been conducted on software categorization. According to the source information used for analysis, these works can be mainly classified into two groups.

The first is about categorization based on source code identifiers and comments. [13], [11], [12], [17] are the typical works that leverage source code information to do categorization. In most of these works, software projects are viewed as documents consisting of source code identifiers and comments. Textual classification approaches are employed to categorize. The main differences among these works are the document representation methods and the machine learning techniques used for categorization. [13] extracts source code identifiers and comments as documents, then applies an EEL[18] approach to do feature selection, and makes use of SVM approach to categorize the software into predefined topic and language categories. MUDABlue [11] and LACT [12] first generate categories from source code and then use LSA and LDA approaches separately to classify software. [19] enhances LACT by integrating domain knowledge for Android applications into the original approach and improves the performance.

The second group is software categorization based on other information like API calls [20], [15]. As the source code of many commercial software are not available, McMillan et al. [20] proposed a new categorization approach based on software API calls. The basic idea is that: external APIs and methods in software are grouped by their functions, thus they can be indicators of categories for the software that use these APIs.

Most of these works categorize software based on the source code or byte code information. Considering the scale of the repositories and the complexity of software source code or byte code, such approaches are not scalable for repository-scale categorization. In this work we explore the capability of software online profile in multiple repositories for scalable and efficient categorization.

### B. Software Online Profile Mining

As more and more software projects are published with profiles, many researchers pay attention to such online data for different aims and get valuable results.

Dumitru et. al [4] study the software online feature descriptions in Softpedia to assist domain analysis. They propose an incremental diffusive clustering algorithm to discover domain-specific features in the massive amounts of software descriptions, and then recommend features for domain analysts after an initial input is provided. McMillan et. al [21] go a step further. By locating chains of function invocations in the source code, they correlate the mined features with corresponding implementation modules.

Recently, tags are widely used to describe software features at repositories. David Lo et. al study the value of software collaborative tags for different aims. In [22] they make use of co-occurrence of tags in Freecode to measure the similarities between pairs of tags and propose a  $k$ -medoids clustering algorithm to construct taxonomy of tags. In addition, they explore the software categories, license, programming languages and other related tags in SourceForge to find similar applications [23].

These works study the new types of software information, but they analyze these information independently and focus on one single repository. Differently, we aggregate the software descriptions and collaborative tags across multiple repositories for hierarchical categorization.

## III. OUR APPROACH

In this section, we first design a hierarchical categorization framework and then discuss the software online profiles for the categorization.

### A. Hierarchical Categorization Framework

1) *Category Hierarchy Definition*: As there are more than one million projects over the Internet, coarse-grained categorization is not efficient. We propose a hierarchical categorization system which includes multi-grained categories that are organized into a hierarchical structure. The category hierarchy is built according to the topics the category covers and the relations among them. For example, in SourceForge the coarse-grained category “Multimedia” is divided into several more specific subcategories like “Video”, “MP3” and so on.

In practice, a software category may belong to two or more parent categories. To simplify the problem, in this paper we model the software category hierarchy as a tree. In this tree, the relation between a pair of child and parent means “IS-A”. It has the following constraints of Asymmetric, Anti-reflexive and Transitive. Figure 1 presents a simple demonstration of the category hierarchy. The “Multimedia” is the root of this category tree, which covers all the other more specific categories like “Video”, “Sound/Audio” and so on. The categories “Video” and alike will form corresponding sub-trees. In this tree, each subcategory can have zero, one or several subcategories, but can only have one direct parent as constrained above. In this paper, we design a virtual category “Root” which is directly connected with all the first-level categories like “Multimedia” here to form a single tree.

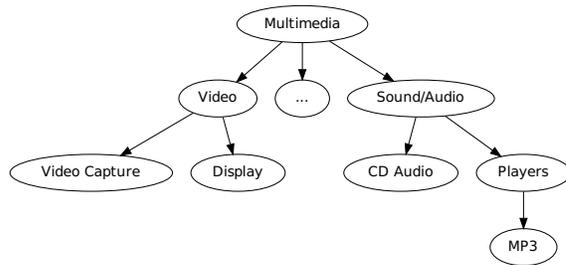


Fig. 1. Categories under “Multimedia” in the constructed category hierarchy

Because it is difficult to define a complete category hierarchy to cover all possible fine-grained topics, some software may belong to none of the given leaf categories but an internal one. It means that the most specific categories for software can be internal or leaf categories in the hierarchy. In this hierarchy, a category is allowed to have one single subcategory, just like the node “Players” in Figure 1. Because it is not mandatory for projects to be assigned with leaf categories, this setting makes sense. This is motivated by the requirement to category software as specific as possible. For projects categorized under “Players”, they will be further tested to see if can be assigned with the more specific “MP3”.

2) *Hierarchy Categorization Learning*: There are many ways to build the hierarchical categorization model, which can be mainly classified as big-bang approach and top-down approach [24]. Big-bang approach treats all the categories at once and learns a single classifier for the entire hierarchy. Such approach will reduce the total size of classification model considerably. However, as the total number of categories is often large, it is difficult to build a single accurate classifier. Top-down approach adopts a different strategy. It builds multiple local classifiers and predicts each subcategory in the hierarchy separately. In [25] it has proved that the top-down approach is superior to the big-bang approach. Thus, in this paper we adopt a top-down approach for training the model. We build local binary classifier per node in the tree except the “Root”, which is the mostly used approach in literature to construct the hierarchical categorization model [24].

To build the local classifier for each node in category hierarchy, there are two key issues to consider. The first is what classification approach to use and the second is how to

select the positive and negative examples for training. For the first issue, there are quite many classification approaches to build the classifier like SVM,  $k$ NN, Naïve Bayes and so on. In this paper we choose SVM as our basis classifier because it has been proved superior to others for software categorization repeatedly in previous works [15], [26].

For the second issue, we adopt a “sibling” strategy in our framework. For a given category node in the hierarchy, the examples of both the node and its descendants are viewed as positive examples, and those categorized with its siblings as well as the descendants of the siblings are viewed as negative ones. For those categories that have no siblings (this is possible as defined in III-A1), the negative examples consists of those that categorized with the siblings of its nearest ancestor. For example, in Figure 1 the category “MP3” has no sibling, so its negative examples are those labelled with “CD Audio” which is the sibling category of its parent. Such strategy is adopted to avoid the serious imbalance problem [25].

3) *Hierarchical Categorization Prediction*: The hierarchical software categorization problem is a non-mandatory leaf node prediction problem [27]. It means that the most specific predicted category for a testing software can be any node (i.e. internal or leaf node) in the category hierarchy except the “Root”. For example, because categories “Video Capture” and “Display” together may not cover the whole topics of “Video”, a project may belong to none of categories “Video Capture” and “Display” but their parent category “Vedio”.

For a given software, we still take a top-down approach to predict its categories. We first test the given software over all the first-level categories with corresponding local classifiers. Then we only go down to test those subcategories whose parents have been predicted positive. One thing we need to note is that one software may belong to two or more categories of the same level in practice, which is a multi-label classification problem [28]. As we adopt binary SVM as our basic local classifier which predicts the categories of the same level separately, the multi-label problem can be solved naturally.

## B. Software Profiles for Hierarchical Categorization

Different from the previous works which make use of source code or byte-code information, we leverage the software online profiles for categorization in this paper. As an alternative source information, they are less studied before.

1) *Software Online Profiles in Repositories*: In this paper, we mainly focus on two types of software profile attributes: software descriptions and collaborative tags. The software resources online are often published with brief descriptions which give high-level summarizations. For example, *MySQL* is hosted in several large software repositories and Table I in Section I lists its descriptions in SourceForge, Ohloh, Freecode. From this table we can observe that the descriptions of *MySQL* in different communities are different, each highlights some features of the software. The combination of these descriptions will give a more comprehensive summarization of it. In the descriptions, the terms like “SQL”, “database management system”, “Oracle”, which often appear more frequently in database related software, suggest the category of this software.

In addition to the high-level descriptions, collaborative tagging is widely used in software repositories to annotate

resources. For example, in Ohloh and Freecode, all registered users are allowed to label the resources in the repository based on their experiments and knowledge. Table II shows the tags of *MySQL* in Ohloh and Freecode. These annotations aggregate the crowds understanding and reflect the features of the resources, which provide useful information for software categorization [23]. In specific, these tags mainly consists of two types. The first type is about functional topics of the system, such as “database, software\_development” for *MySQL*. The second type is about the techniques features of the project like “odbc, sql, jdbc”. For the functional topics or tags, they often explicitly reflect the category of the software. While for technique words or tags, as some techniques tend to be used more often in some specific categories of software, such tags are effective indicators of the system’s category.

The software profiles present high-level and important features of the resource, which is complementary to the detailed source code or API calls. Source code identifiers or API names often reflect detailed features of the software at a granularity of method, class or package. While the software descriptions and tags provide high-level summarizations of functional or technical features over the whole software. This is the key motivation for us to explore the capability of these two types of software profile attributes for categorization.

2) *Combination of Software Profiles for Categorization*: Although software descriptions and tags are both high-level summaries of the resources, there are some differences between them. Software descriptions are often given by software managers to exhibit and popularise the resource. There are many words that are not related to software functions or techniques, which are often noise for categorization. Differently, collaborative tags are often labelled by software users, maintainers or other software stakeholders to annotate the key features of the resource. These tags covers the functional or technical aspects of the resource. Thus, they are often of better quality except some idiosyncratic or misspelling ones. Such differences are obvious as shown in Table I and II. These two types of profile attributes are complementary to each other and the combination of them will give more comprehensive summarization. However, because most of the projects are annotated with much less tags compared to the number of words in the descriptions, direct combination of them will dilute the weights of tags.

To balance the impacts of tags and descriptions on categorization, we distinguish the tags from the the common words in descriptions and design a weighted combination strategy to strengthen the weights of tags. We duplicate the tags several times before combining them with software description. The duplicate time is decided according to the ratio between the length of software descriptions and its tags. The intuition behind this is as follow. As annotated by crowds, collaborative tags present the key features of the software, and they should be equivalent to software description at summarizing the software. Thus, the overall normalized term frequency of all the tags for a software should be proximity to that of the description. So we repeat the tags many times to make the total length of them be proximate to that of software description. For tags from different repositories, as we will take steps to only retain the commonly used ones as discussed in the experiments, currently the sources of these tags are not taken

into consideration for deciding their weights. The duplication times  $\delta$  is decided according to Eq. 1.

$$\delta = \alpha * \sqrt{\frac{\sum_k t_{kj}}{\sum_m \hat{t}_{mj}}} \quad (1)$$

In Eq. 1  $t_{kj}$  represents the appearance number of term  $k$  in project  $j$  and  $\hat{t}_{mj}$  is that of tag appears in project  $j$ . In practice, some software are only annotated with very few tags which fail to cover all aspects of what description presented. To reduce the influence of these tags in such software, we take the square root over the ratio. In addition, we multiply it by another parameter  $\alpha$  to control the overall duplication times. When  $\alpha$  is set to 0, it becomes the simple combination without duplication which views tags as common words in descriptions.

In this paper, we use the traditional TF-IDF to represent the importance of a term for distinguishing a project at categorization. The traditional TF-IDF is computed for each word/tag according to Eq. 2. In this Equation,  $t_{ij}$  represents the count of appearance for term  $i$  which can be a common word in description or a tag for software  $j$ ,  $n_i$  stands for the number of software in which term  $i$  appears and  $N$  is the total number of software in the corpus. The first part of the equation is the normalized term frequency of word  $i$  in project  $j$ , and the second part is the inverse document frequency of word  $i$ .

$$w_{ij} = \frac{t_{ij}}{\sum_k t_{kj}} * \log \frac{N}{n_i} \quad (2)$$

According to Eq. 2, the duplication of tags will increase of normalized term frequency of the tags and decrease the term frequency of the words in description. Because it will not change the the inverse document frequency, the duplication of tags will increase the TF-IDF weights of tags and decrease that of description words.

#### IV. EXPERIMENTS DESIGN

In this section, we describe the experiment questions, experiment dataset and settings as well as the corresponding evaluation metrics.

##### A. Experiment Questions

To explore the effectiveness of different software profile attributes for categorization and comprehensively evaluate the proposed hierarchical categorization framework, we focus on the following four experiment questions.

- **Q1:** Which type of profile attributes is more effective for categorization, software descriptions or collaborative tags?
- **Q2:** What is the detailed performance of our approach for different level of categories?
- **Q3:** Are software online profiles as effective as API calls extracted from source code and byte code for categorization?
- **Q4:** Is profile-based categorization scalable for categorizing Internet-scale software repositories?

For experiment question Q1, we explore the effectiveness of the two types of profile attributes for categorization: software descriptions and collaborative tags. For Q2 we have a insight look at the the performance of our approach at different category levels. For Q3 we aim to compare the capability of software profiles and API calls for categorization. For Q4, we analyse the capability of our approach for scalable categorization.

##### B. Dataset and Experimental Settings

To address the above research questions and validate our approach, we focus on three large and popular open source repositories: SourceForge, Ohloh and Freecode. In this subsection we present the experiment dataset and the constructed category hierarchy.

1) *Experiment Dataset:* SourceForge is one of the largest and most popular open source community. It has predefined a hierarchical category system with 363 categories which is the basis for constructing our category hierarchy, and these projects that are categorized can be our training samples. Ohloh and Freecode are both large open source repositories which collect more than 400,000 and 45,000 software project respectively. In Ohloh and Freecode, they adopt a collaborative tagging mechanism to annotate and organise these collected resources. We crawl project profiles from Ohloh and Freecode to enrich the profile of software in SourceForge.

We construct the experiment dataset through the following steps: (1) *Software homepage crawling.* We crawl the homepage of the software in the three open source repositories and parse them to extract the profile attributes including descriptions, categories and tags. (2) *Software profile aggregation.* Among these crawled software projects, a proportion of them exist in multiple repositories, like *MySQL* listed in Table I. Such software not only have categories in SourceForge but also have tags in Ohloh or Freecode. We retain these projects and combine their profiles to train hierarchical classifiers. (3) *Profile preprocessing.* We do stop words removing and stemming over the descriptions and tags. After this, we only retain those projects which have combined descriptions of more than 10 words. In addition, to get rid of those idiosyncratic or misspelling tags, we set a threshold of 50 to filter such tags, i.e. only those tags that have been used more than 50 times in the corresponding repositories will be retained. After the three steps, we get a number of 18,032 unique software and a total of 5,429 unique tags. The detailed information of the dataset is shown in Table III.

The first part is the projects from SourceForge, each of which has at least one category. These software projects have descriptions of about 20 words and 3 categories in average; The second and third part are the projects from Ohloh and Freecode. There are 9,813 and 10,357 projects in Ohloh and Freecode which also exist in the 18,032 SourceForge projects. The total projects in Ohloh and Freecode are 2,138 more than that of SourceForge which implies that there are 2,138 projects among the dataset exist in all the three repositories. These software in Ohloh and Freecode have similar description length and average tags. The retained projects only account for a small proportion of the total software in these repositories. There are several reasons. Firstly, the software projects that

TABLE III. SUMMARIES OF THE DATASET CRAWLED FROM SOURCEFORGE, OHLOH AND FREECODE

Repository	Num. of software	Avg. description length	Num. unique categories	Avg. categories	Num. unique tags	Avg. tags
SourceForge	18,032	19.69	307	2.98	-	-
Ohloh	9,813	20.84	-	-	5,373	5.73
Freecode	10,357	25.17	-	-	940	4.85

are categorized in SourceForge and also tagged in Ohloh or Freecode take only a small proportion [29]. Secondly, we take a strict matching strategy on full-name to search projects in multiple repositories for profile aggregation. Such strategy is adopt to avoid those projects which have similar name but actually not the same.

2) *Category Hierarchy Construction*: We build the category hierarchy based on the predefined one in SourceForge. The categories in SourceForge include coarse-grained ones like “Multimedia” and “Games” as well as specific ones like “MP3” and “First Person Shooter”, which are organized in four levels. Similar categorization systems are also adopted in other communities like Rubyforge. Here we constructed our category hierarchy based on the SourceForge categorization system as follows.

- 1) *Transforming the DAG structure to tree structure*. In the original category hierarchy, there can be more than one path between two nodes. To simplify the classification process, we only retain the mostly used path in the dataset.
- 2) *Pruning the category hierarchy*. Firstly we combine those categories which are quite similar. Secondly we delete those categories which have less examples than a predefined threshold and then we lift the examples under them to the corresponding parent categories.
- 3) *Constructing the uniform category hierarchy*. After DAG transformation and hierarchy pruning, we create a virtual root to cover all the original root categories. This results in a single category hierarchy tree.

For category pruning, the thresholds for deletion are set to 500, 100, 50, 50 for the four levels categories. After the preprocessing, we constructed a uniform hierarchy which consists of 123 categories with four levels. Figure 1 presents a part of the constructed category hierarchy under “Multimedia”.

The summarization of the constructed hierarchy is presented in the Table. IV. There are 12 top categories at level one including “Multimedia, Games/Entertainment, Office/Business” and so on. Most of the categories are cramped in the second and third level, which have 61 and 41 of them respectively. The fourth level have only 9 categories. Among the 18,032 projects in the dataset, the average number of positive samples for the first level categories is 2215.33. As the category level increases, the average number of positive samples decreases dramatically to only a few hundreds, and this will affect the performance of our approach. More samples will be included in the future.

### C. Evaluation Metrics

In previous works like [15], [20], the precision, recall and F-Measure are widely used to evaluate the performance of flat categorization system. In this paper we make a slight change

TABLE IV. DETAILS OF THE CONSTRUCTED CATEGORY HIERARCHY

Category Level	Num. of categories	Avg. number of positive examples
Level 1	12	2215.33
Level 2	61	382.15
Level 3	41	196.85
Level 4	9	120.00

on these metrics and adopt hierarchical metrics including hierarchical precision( $hP$ ), hierarchical recall( $hR$ ) and hierarchical f-measure ( $hF$ ) over each category. The definition of these metrics are as follows.

$$hP = \frac{\hat{P}_i \cap \hat{T}_i}{\hat{P}_i}, hR = \frac{\hat{P}_i \cap \hat{T}_i}{\hat{T}_i}, hF = \frac{2 * hP * hR}{hP + hR} \quad (3)$$

In Eq. 3, for each category  $i$ ,  $\hat{P}_i$  is the predicted sample set for category  $i$ . It consists of software whose categories are predicted as category  $i$  or the descendants of  $i$ ,  $\hat{T}_i$  is the true sample set that consists of all projects labelled with category  $i$  and the descendants of  $i$ . The three metrics  $hP$ ,  $hR$  and  $hF$  represent the average precision, recall and F-Measure for each category over all the testing examples where the hierarchical structure is concerned.

To measure the average performance over all the categories, we make use of Micro Average on hierarchical Precision(Micro- $hP$ ), Recall(Micro- $hR$ ) and F-Measure(Micro- $hF$ ) as Eq. 4 which are similar to these used in [24], [30].

$$Micro-hP = \frac{\sum_i (\hat{P}_i \cap \hat{T}_i)}{\sum_i \hat{P}_i}, Micro-hR = \frac{\sum_i (\hat{P}_i \cap \hat{T}_i)}{\sum_i \hat{T}_i}, Micro-hF = \frac{2 * Micro-hP * Micro-hR}{Micro-hP + Micro-hR} \quad (4)$$

## V. EXPERIMENT EVALUATIONS

In the experiments, we first build the hierarchical categorization model and then test on large amounts of projects. The SVM classification algorithm is used as the basic classifier and the parameters are set as default except the parameter  $c$ . In the experiments we set  $c$  to 0.5 for which achieves the best performance after a simple test. The detailed information of the datasets used for experiments and the hierarchical categorization framework are discussed in Section IV. We first use 5-fold cross validation and hierarchical metrics to measure our approach and compare the different types of profile attributes for categorization. Then we compare our approach with a previous work which categories software based on API calls. For 5-fold cross validation, the whole dataset is broken into 5 folds randomly, and each of the folds will be tested once by using the other 4 folds to train the model.

### A. Q1: Software Descriptions and Collaborative Tags for Categorization

We explore two different types of profile attributes in this work: software descriptions and collaborative tags. These attributes have never been extensively studied for software categorization before. In this section we conduct four sets of experiments to compare the effectiveness of the different types of profiles for categorization. Firstly we only make use of SourceForge software descriptions to train and test. Then we aggregate the software descriptions from the three repositories to test. Thirdly, we do experiment on combination of software tags from Ohloh and Freecode. Finally, we simply combine software descriptions with tags from all these three repositories to do categorization.

Table V shows the experiment results. *SF descriptions*, *Combination of descriptions*, *Combination of tags* and *Combination of descriptions and tags* in the table stand for the four sets of corresponding experiments mentioned above where *SF descriptions* stands for the experiment based on SourceForge software descriptions. In this experiment, we get an overall micro precision of about 58.68% and recall of 48.37%. By aggregating the descriptions from other repositories, the categorization precision is only improved slightly while the recall is almost the same. This is because that for a large proportion of software, the descriptions of the same software in different repositories overlap each other a lot, which fails to provide much additional information. For collaborative tags, though each software project has much less tags than description words, the experiment based on combination of tags still achieves better results. The micro precision is similar to that of *Combination of description*, but the micro recall shows a significant improvement of about 13.68% and F-Measure gets an improvement of 8.50%. This is due to the better quality of collaborative tags. Different from software descriptions which often contains many words like “wide range”, “performance” for *MySQL* that reflect no technical or functional features, most of the tags have specific meaning and reflect some aspects of features for the resource. Thus, categorization based on collaborative tags achieves better results.

As shown in the fourth row, we simply combine the software descriptions and tags. In this experiment we get better precision but worse recall than that of tags. Compared to collaborative tags, such combination treats each tag as common word in description. On the one hand, such combination aggregates more information of the software; On the other hand, it introduces more noisy words and dilutes the weights of tags. Thus, it fails to improve the categorization performance over that of *Combination of tags*.

TABLE V. MICRO HIERARCHICAL PRECISION, RECALL AND F-MEASURE FOR USING DIFFERENT TYPES OF PROFILE ATTRIBUTES

Software profile	Micro-hP	Micro-hR	Micro-hF
SF descriptions	0.5868	0.4837	0.5302
Combination of descriptions	0.6213	0.4839	0.5440
Combination of tags	0.6375	0.6207	0.6290
Combination of descriptions and tags	0.6831	0.5746	0.6241

To make full use of the descriptions and tags for categorization, in Section III-B2 we propose a weighted combination strategy which aims assign more weights to more important

attributes by duplication. In Eq. 1 we design a parameter  $\alpha$  which controls the overall duplication time for tags. Figure 2 presents the performances for different values of  $\alpha$ .

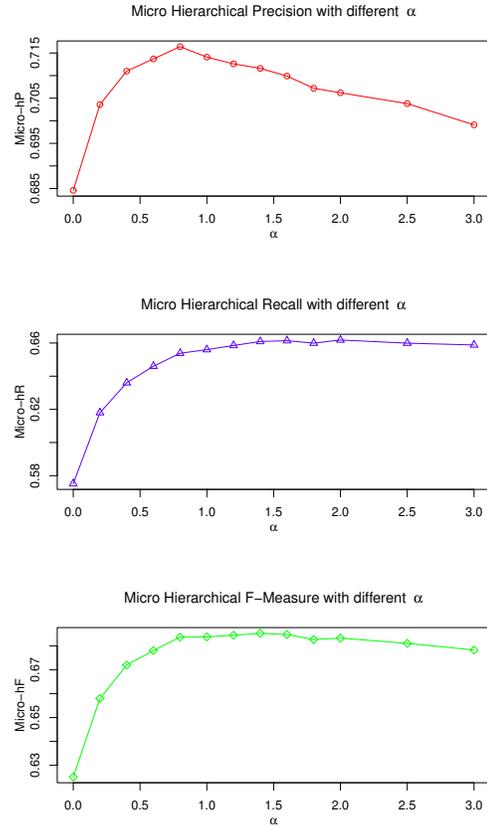


Fig. 2. Categorization performance with different values of  $\alpha$  based on weighted combination of profiles

It shows that as the  $\alpha$  increases from 0 to 1, both precision and recall have a great improvement, and the F-Measure improves from about 62.41% to about 68.38%. This verifies the effectiveness of our weighted combination strategy. While the parameter  $\alpha$  keeps on increasing from 1 to 3 and larger, the precision decreases slowly, and the recall and F-Measure keep almost the same. This is reasonable. As  $\alpha$  increases, the weights of tags will become much larger than that of words in descriptions. This leads to the result that the descriptions have only slight effect on distinguishing software categories.

Overall, from the above results we can find that collaborative tags are more effective attributes than software descriptions for categorization. Nevertheless, those two types of attributes are mutually complementary, and the weighted combination of them which assigns greater weights to tags properly will improve the overall performance. In addition, from Table V we can see that, even without the additional tags and descriptions, our hierarchical categorization framework is still applicable by only using the descriptions from SourceForge and achieves an overall F-measure of about 53.02%.

### B. Q2: The categorization performance for different category levels

In Q1 we present the average results over the whole four-level category hierarchy. As different level of the categories often have different number of positive examples, it will affect the performances of categorization. To get a comprehensive understanding of such influence, we analyze the performance for the top-k level categories where  $k$  ranges from 1 to 4. Table VI presents the detailed results based on weighted combination of profiles with  $\alpha = 1$  as discussed in Section V-A.

TABLE VI. CATEGORIZATION PERFORMANCE FOR TOP-K LEVEL CATEGORIES BASED ON WEIGHTED COMBINATION OF PROFILES WITH  $\alpha = 1$

top-k	Num. Categories	Micro-hP	Micro-hR	Micro-hF
1	12	0.8534	0.7008	0.7695
2	73	0.7767	0.6647	0.7163
3	114	0.7365	0.6567	0.6942
4	123	0.7141	0.6560	0.6838

As shown in this table, the overall performance of the first level of 12 categories achieves 85.34%, 70.08% and 76.95% for precision, recall and F-measure respectively, which is relatively high. As the category level rises, the overall performance decreases accordingly. This is because the deeper categories often have fewer examples which affect the categorization accuracy. As listed in Table IV in Section IV-B2, the average number of positive examples decreases from 2,215 at first level to about 120 at fourth level. It will be our future work to introduce more samples for training more accurate categorization model.

### C. Q3: Weighted Combination of Software Profiles and API calls for Categorization

In this section we carry out an experiment to compare the categorization performance by using software profiles and that of software API calls. APIs are grouped into packages and libraries according to their functions. Thus, the APIs invoked in a software would be good indicators of the software category. Based on this intuition, McMillan et. al [20] leverage API calls for categorization. Their case studies suggest that API calls are as effective as source code identifiers and comment terms for categorization. In their works the effectiveness of using API packages and API classes are verified respectively. We only compares their best setting (using API packages) with our approach by using profile combination with  $\alpha = 1$ .

In the experiment, we test our approach on the dataset used in their experiments. Among their SourceForge dataset, we search for the projects which are tagged in Ohloh or Freecode and get a total of 849 ones which are used as the testing set. Then we train the categorization model based on the our dataset in which the testing set are rejected. The final results for the 22 categories are shown in Table VII.

Overall, the two approaches achieve similar F-Measure over all the 22 categories as shown in the last row of the table. In specific, for some categories like “Chat”, “Communications” and “Emails”, our approach get better results. While for some others such as “Visualization” and “Front-Ends”, API-based approach achieves higher F-Measure. Such difference maybe result from the different characteristics of

TABLE VII. F-MEASURE OF EACH CATEGORY BASED ON API PACKAGES AND WEIGHTED COMBINATION OF PROFILES

Category	F-Measure	
	API calls	Software profiles
Bio-Informatics	0.743	0.583
Chat	0.742	0.815
Communications	0.601	0.723
Compilers	0.712	0.616
Database	0.689	0.664
Education	0.633	0.583
Email	0.746	0.792
Frameworks	0.727	0.806
Front-Ends	0.702	0.437
Games/Entertainments	0.728	0.703
Graphics	0.607	0.580
Indexing/Searching	0.732	0.667
Internet	0.604	0.581
Interpreters	0.635	0.679
Mathematics	0.658	0.775
Networking	0.567	0.481
Office/Business	0.598	0.617
Scientific	0.654	0.708
Security	0.622	0.727
Testing	0.730	0.678
Visualization	0.605	0.400
WWW/HTTP	0.696	0.605
AVG.	0.6696	0.6464

attributes used for categorization. The name of API packages and classes are implementation-related attributes which mainly reflect the functionality of the package or the class, which is fine-grained features. While the software profiles emphasis on high-level summaries of the resource. These two types of attributes reflect the features of the software from different perspectives with different granularity. The results suggest that the software profiles are effective alternative to software APIs for categorization. Intuitively, they are complementary to each other and a proper integration of them should be more effective. It will be our future work to have a comprehensive analysis on this.

It should be noted that we do categorization hierarchically with more finer-grained categories. In addition to the 22 categories, some of the 849 testing software in Sourceforge are originally classified with many finer-grained categories. Our approach organizes these categories hierarchically and achieves high accuracy at these specific categories as well. For example, in our constructed category hierarchy, the category “Compilers, Testing” in the 22 categories are organized under the category “Software Development”. The “Software Development” category are divided into more specific ones like “Build Tools, Object Oriented, Algorithms, Quality Assurance” and so on. Among these testing software, many of them are originally grouped under these specific categories, and our approach classifies these with high accuracy. Table VIII presents the results for the categories under “Software Development”.

### D. Q4: Scalability Analysis

Internet-based software maintenance greatly relies on efficient categorization of the massive software in repositories.

TABLE VIII. PERFORMANCE FOR CATEGORIES UNDER “SOFTWARE DEVELOPMENT” WITH WEIGHTED COMBINATION OF PROFILES

Category	hP	hR	hF
Software Development	0.8526	0.7826	0.8161
Object Oriented	0.9333	0.6667	0.7778
Build Tools	1.0	0.5833	0.7368
Algorithms	0.9000	0.4737	0.6207
Quality Assurance	0.8000	0.6667	0.7273
User Interface	0.5484	0.7083	0.6182

In this section we analyze the scalability of our approach for categorizing the Internet-scale repositories.

The cost of software categorization mainly consists of two parts. The first is training data obtaining and preprocessing and the second is the model training and prediction. For the first part, to get the profiles of the 417,344 software in Ohloh, we design a processing chain including web crawler and profile extractor. The crawler is implemented with 50 threads to crawl the homepage of the software and the profile extractor parses the homepage to get corresponding profile information. The software profiles are given in natural language which is not specific to any programming language. Thus, the profile extraction and pre-processing is simple. We deploy the crawler and extractor in a server (8\*2.13G CPUs, 16GB RAM and 2TB storage) which is connected to the Internet with network bandwidth of 100M. Totally, it takes less than 3 days to get all the software profiles in Ohloh with a total size of about 100 MB for the extracted profile attributes.

The model training and category prediction on the pre-processed dataset are quite efficient. The cost time for model training and testing mainly depends on the classification algorithm used. We do our experiments using SVM algorithm on a computer of Intel(R) Core(TM) i5-3320M CPU @ 2.6M with 4GB RAM. The total time for five-cross validation over 18,032 projects is about 620 seconds, in which the training time is about 540 seconds and testing time is about 80 seconds. Based on this we can estimate that the time for categorizing the total 417,344 software in Ohloh will take about 31 minutes once the categorization model is built.

In contrast, categorizing the repository software based on source code will be much more complex. Firstly, considering the huge amounts of projects in the repositories, to get the source code of all the software is time-consuming. Secondly, the analysis of the source code is quite complex. For example, in June 2011 *MySQL* has about 1,333,855 lines of code and 298,918 lines of comments<sup>2</sup>. To parse the source code of it for identifiers and comments is quite a time-consuming process, let alone the huge amounts of software which are implemented in various programming languages.

Based on the analysis we can conclude that our approach is capable of doing categorization for the large repositories with high efficiency.

## VI. VALIDITY

There are some threats to validity which may affect the experiment results of our approach. One is that the software

may be incorrectly categorized in SourceForge or incorrectly tagged in Ohloh and Freecode. It is very difficult to eliminate such kind of threat. In our paper, we minimize such threat by including as many samples as possible to reduce the ratio of such incorrectly categorized software.

Another threat is that the same name software we crawled from multiple repositories may not be the same software. Such mistakes will introduce incorrect profiles in the dataset. To minimize such threat, in this paper we adopt a strict full-name matching strategy to filter the false ones. In the future we will design a profile-comparison tool to compute the similarities of the software projects with same names, and thus the quality of training data will be further improved.

As discussed in Q1, the software tags have a great impact on the performance of our categorization approach. Thus, the tag shortage may affect the generality of our approach. However, for such software that have no tag, we can classify them based on their descriptions. As shown in Table V, the categorization precision based on descriptions achieves 58.68% as well. In addition, as social tagging is becoming widely accepted, more and more software will be annotated with tags, which can be re-categorized efficiently based on the new data.

## VII. CONCLUSION AND FUTURE WORKS

Internet-scale software repositories requires more scalable and finer-grained categorization. In this paper we propose a hierarchical categorization approach by mining software online profiles across multiple repositories. We categorize software with more than 120 categories which are organized into a hierarchical structured. Such categorization greatly improve the efficiency for retrieving similar software. As online profiles is not specific to any programming languages or source code, and it is easy to obtain and analyse as well. Thus, it is scalable to do categorization over the huge amounts of software in repositories efficiently. The extensive experiments on more than 18,000 software show that our approach achieves promising performance over the whole category hierarchy. Compared to the approach that using APIs for flat categorization, our approach achieves competitive results, and furthermore, our approach is capable of predicting more specific categories hierarchically.

In addition, we have developed an open source software searching and ranking system named Influx<sup>3</sup>, which has crawled OSS data from the influential repositories (such as SourceForge, Ohloh, Freecode). Currently, Influx provides some interesting services by mining OSS data, such as cross-repositories profiles, synergy analysis and software ranking in some repositories (such as OW2). A demo of the hierarchical categorization system proposed in this paper has been integrated into Influx and can be visited now.

There are several possible ways to improve the performance for categorization. Firstly, more software can be crawled from websites over the Internet to enrich the training set. In this paper, although we crawled more than 18,000 software from three software repositories, the positive samples for some categories are only about 100, which affects the accuracy for classification. In the future, more samples can be retrieved from

<sup>2</sup><https://www.ohloh.net/p/mysql>

<sup>3</sup><http://influx.trustie.net>

various repositories to improve the performance. Secondly, in this paper we treat the tags from different repositories equally. Further analysis about the characteristics of these tags will be done to explore their potential for categorization. In addition, software attributes of API calls and software profiles reflect software features at different granularity, which are complementary to each other. We will study how to make full use of the strengths of these different attributes.

#### ACKNOWLEDGMENT

This research is supported by the National Science Foundation of China (Grant No.60903043) and the National High Technology Research and Development Program of China (Grant No. 2012AA011201). We would like to thank Xiao Li, Yue Yu and Li Fang for their collaboration. We would also thank the anonymous reviewers for providing us constructive comments and suggestions.

#### REFERENCES

- [1] "Ieee standard for software maintenance," *IEEE Std 1219-1998*, 1998.
- [2] A. Kuhn, "Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code," in *Mining Software Repositories, 2009. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 175–178.
- [3] C. McMillan, "Searching, selecting, and synthesizing source code," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1124–1125.
- [4] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *Software Engineering, 2011 33rd International Conference on*. IEEE, 2011, pp. 181–190.
- [5] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sept., pp. 462–471.
- [6] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 57–62.
- [7] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 289–298.
- [8] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.
- [9] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos, "Recommending people in developers' collaboration network," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 379–388.
- [10] D. Kuang, X. Li, and C. X. Ling, "A new search engine integrating hierarchical browsing and keyword search," in *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, ser. IJCAI'11. AAAI Press, 2011, pp. 2464–2469.
- [11] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.
- [12] K. Tian, M. Reville, and D. Shyvyanyk, "Using latent dirichlet allocation for automatic categorization of software," in *Mining Software Repositories, 2009. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 163–166.
- [13] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: automatic classification of source code archives," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 632–638.
- [14] P. S. Sandhu, J. Singh, and H. Singh, "Approaches for categorization of reusable software components," *Journal of Computer Science*, vol. 3, no. 5, pp. 266–273, 2007.
- [15] C. McMillan, M. Linares-Vasquez, D. Shyvyanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 343–352.
- [16] C. McMillan, M. Grechanik, and D. Shyvyanyk, "Detecting similar software applications," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 364–374.
- [17] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "On automatic categorization of open source software," in *In 3rd Workshop on Open Source Software Engineering*, 2003, pp. 63–68.
- [18] N. Abramson, *Information theory and coding*, ser. McGraw-Hill electronic sciences series. New York, NY: McGraw-Hill, 1963.
- [19] C.-Z. Yang and M.-H. Tu, "Lacta: An enhanced automatic software categorization on the native code of android applications," *Lecture Notes in Engineering and Computer Science*, vol. 2195, no. 1, pp. 769–773, 2012.
- [20] M. Linares-Vsquez, C. McMillan, D. Shyvyanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Software Engineering*, pp. 1–37, 2012.
- [21] C. McMillan, N. Hariri, D. Shyvyanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 848–858.
- [22] S. Wang, D. Lo, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012, pp. 604–607.
- [23] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, vol. 0, pp. 600–603, 2012.
- [24] J. Silla, CarlosN. and A. Freitas, "A survey of hierarchical classification across different application domains," *Data Mining and Knowledge Discovery*, vol. 22, pp. 31–72, 2011.
- [25] G.-R. Xue, D. Xing, Q. Yang, and Y. Yu, "Deep classification in large-scale text hierarchies," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '08. New York, NY, USA: ACM, 2008, pp. 619–626.
- [26] G. A. D. Lucca, M. D. Penta, and S. Gradara, "An approach to classify software maintenance requests," in *In Proc., International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2002, pp. 93–102.
- [27] A. A. Freitas, "A tutorial on hierarchical classification with applications in bioinformatics," in *In: D. Taniar (Ed.) Research and Trends in Data Mining Technologies and Applications, Idea Group, 2007*, pp. 175–208.
- [28] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Int J Data Warehousing and Mining*, vol. 2007, pp. 1–13, 2007.
- [29] T. Wang, G. Yin, X. Li, and H. Wang, "Labeled topic detection of open source software from mining mass textual project profiles," in *Proceedings of the ACM SIGKDD Workshop on Software Mining*, ser. SoftwareMining'12. New York, NY, USA: ACM, 2012, pp. 17–24, Best Paper.
- [30] X. Li, D. Kuang, and C. X. Ling, "Active learning for hierarchical text classification," in *Proceedings of the 16th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining - Volume Part 1*, ser. PAKDD'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 14–25.

# Mining Software Repositories for Accurate Authorship

Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat  
 Computer Sciences Department  
 University of Wisconsin  
 Madison, WI 53706 USA  
 {xmeng, bart, bill, bernat}@cs.wisc.edu

**Abstract**—Code authorship information is important for analyzing software quality, performing software forensics, and improving software maintenance. However, current tools assume that the last developer to change a line of code is its author regardless of all earlier changes. This approximation loses important information. We present two new line-level authorship models to overcome this limitation. We first define the *repository graph* as a graph abstraction for a code repository, in which nodes are the commits and edges represent the development dependencies. Then for each line of code, *structural authorship* is defined as a subgraph of the repository graph recording all commits that changed the line and the development dependencies between the commits; *weighted authorship* is defined as a vector of author contribution weights derived from the structural authorship of the line and based on a code change measure between commits, for example, best edit distance. We have implemented our two authorship models as a new git built-in tool *git-author*. We evaluated *git-author* in an empirical study and a comparison study. In the empirical study, we ran *git-author* on five open source projects and found that *git-author* can recover more information than a current tool (*git-blame*) for about 10% of lines. In the comparison study, we used *git-author* to build a line-level model for bug prediction. We compared our line-level model with a representative file-level model. The results show that our line-level model performs consistently better than the file-level model when evaluated on our data sets produced from the Apache HTTP server project.

## I. INTRODUCTION

Information as to who wrote a given piece of code, authorship, is used to analyze software quality [5, 11, 32, 35, 38], perform software forensics [33], and improve software maintenance [13, 14]. Current tools approximate line level authorship by assuming that the last person to change a line is its author, while ignoring all earlier changes. In this paper, we show how to mine a code repository for the development history of a line of code to assign contribution weights to multiple authors. Using these contribution weights, we can attribute a line to the most responsible author in binary code forensics, directly apply the weights to model source code familiarity, and trace back to earlier commits to determine when bugs were introduced in software quality analysis. Our new method abstracts code repositories as a graph representing the development dependencies between commits. We perform a backward flow analysis based on the results of an enhanced line differencing tool [8] between adjacent commits to extract the development history of a line of code. We then use the

history to attribute each character of the line to the responsible author and assign contribution weights. We have implemented this new functionality as an extension to git.

The methods used by current tools (*git-blame* [11, 32], *svn-annotate* [38], and *CVS-annotate* [35]) for obtaining line level authorship loses information. A line of code may be changed multiple times by different developers to fix bugs, to conform to interface changes, or to tune parameters. These changes compose the history of a line of code. For each line of code, current tools report the last commit that changed the line and the author of that last commit. These tools take the last snapshot, while missing the earlier stages of the development history. Therefore, even when the last commit changes only a small fraction of a line of code, the author of the last commit still is credited for the entire line.

In this paper, we define the *repository graph*, *structural authorship*, and *weighted authorship* to help overcome these limitations. The repository graph is a directed graph representing our abstraction for a code repository. In the graph, nodes are the commits and edges represent the development dependencies. For each line of code, we define structural authorship and weighted authorship. Structural authorship is a subgraph of the repository graph. The nodes consist of the commits that changed that line. Development dependencies between the subset commits form the edges. Weighted authorship is a vector of author contribution weights derived from the structural authorship of the line. The weight of an author is defined by a code change measure between commits, for example, best edit distance [36]. We use these two models to extract the development history of a line of code and derive precise line level authorship.

To evaluate our new models, we implemented structural authorship and weighted authorship as a new git built-in tool: *git-author*. We conducted two experiments to show how often the new models will produce more information and whether this information is useful for analysis tools that are based on code authorship information. In the first experiment, we ran *git-author* over the repositories of five open source projects and found that about 10% of the lines were changed by multiple commits and about 8% of the lines were changed by multiple authors. Analysis tools lose information on these lines when they use the current methods for line level authorship. In the second experiment, we used *git-author* to build a new

line-level bug prediction model. We compared our line-level model with a representative file-level model [22] on our data sets derived from the Apache HTTP server project [1]. The results show that the line-level model performs consistently better than the file-level model when evaluated on effort-aware metrics [22, 25].

This work makes the following contributions:

- 1) The structural authorship model that extracts the development history of a line of code and overcomes the fundamental weakness of current tools.
- 2) The weighted authorship model that assigns contribution weights to each change of the line and produces precise line-level authorship attribution.
- 3) The tool *git-author* that is a new built-in tool in git and implements the structural authorship and the weighted authorship model.
- 4) A study of five open source projects that characterizes the number of lines changed by multiple commits and multiple authors.
- 5) A line-level bug prediction model that performs consistently better than the file-level model [22].

We provide an overview of version control systems and define our graph abstraction for code repositories in Section 2. We present the structural authorship model in Section 3 and the weighted authorship model in Section 4. We evaluate our new models in Section 5. We discuss related work in Section 6 and then conclude in Section 7.

## II. REPOSITORY ABSTRACTION

We define the repository graph to capture the fundamental capability of a version control system (VCS). With the repository graph, we can focus on the contents of development history without considering which specific VCS is used. A VCS records the development history of a project by storing all the revisions of source code and the dependent relationship between these revisions. Our graph abstraction models revisions as nodes and the relationship between revisions as edges. We are able to implement the graph structure based on any current mainstream VCS.

A VCS allows programmers to checkpoint their changes. A new revision is created when a programmer commits their modifications to the VCS. The dependencies between revisions also is recorded to maintain the relative order of commits. Current VCS's support concurrent development. Programmers can work on different branches without affecting other people's work and later combine their work by merging branches. Therefore, it is also necessary to record on which existing revision the new revision is based. In addition to these basic capabilities, a VCS often supports reverting previous changes, browsing development history, and other complementary capabilities to facilitate daily development work.

The repository graph is a directed graph  $G = (V, E, \Delta)$  used to describe the basic capability of a VCS. A node in  $V$  represents a revision or a snapshot of the project and is annotated with information about the snapshot including the author of the snapshot. The snapshot of node  $i$  is denoted

as  $s_i$ ,  $s_i \in V$ , and the author is denoted as  $a_i$ . Node  $s_0$  is a virtual node representing the empty repository before any changes are committed.  $E$  is the set of edges, representing development dependencies between revisions.  $\Delta$  is a labeling of  $E$  that represents code changes and there is a one-to-one mapping between the elements in  $E$  and  $\Delta$ . We adapt our definition of code changes from Zeller and Hildebrandt [39], where a change  $\delta$  is a mapping from old code to new code. An edge  $e_{i,j}(\delta_{i,j})$ ,  $e_{i,j} \in E$  and  $\delta_{i,j} \in \Delta$ , means that by applying the change  $\delta_{i,j}$  to  $s_i$ , we can get code snapshot  $s_j$ ; so  $\delta_{i,j}(s_i) = s_j$ . We define  $\delta_{i,j}$  to be a tuple of  $(\mathcal{D}_{i,j}, \mathcal{A}_{i,j}, \mathcal{C}_{i,j})$  where  $\mathcal{D}_{i,j}$  is the set of lines deleted from  $s_i$ ,  $\mathcal{A}_{i,j}$  is the set of lines added to  $s_i$ , and  $\mathcal{C}_{i,j}$  is the set of pairs of lines changed from  $s_i$  to  $s_j$ . For node  $s_i$ ,  $s_j$ , and  $s_k$  such that  $e_{i,j} \in E$  and  $e_{j,k} \in E$ , we define the composition of change sets as  $\delta_{i,k} = \delta_{j,k} \circ \delta_{i,j}$  meaning applying  $\delta_{i,j}$  first, and then  $\delta_{j,k}$ . Our definition implies that the operator  $\circ$  is right associative. One key property of the composition operation is that the result of composition of change sets is path independent. The result only depends on the two end nodes.

We illustrate our definition in Figure 1. The repository consists of ten revisions (ten nodes) and three developers: Alice, Bob, and Jim. The author information for a node is represented by its color. The virtual node  $s_0$  has no author information, so we leave it blank. Alice created a branch for her work and committed  $s_3$  and  $s_4$ . Later Bob merged Alice's work back to the master branch and created  $s_7$ . For the path independent property, we have  $(\delta_{4,7} \circ \delta_{3,4} \circ \delta_{2,3})(s_2) = (\delta_{6,7} \circ \delta_{5,6} \circ \delta_{2,5})(s_2) = s_7$ . The first part in the equation is the composition along Alice's branch. The second one is the composition along the master branch. The two paths yield the same overall effects, which is the third part in the equation.

Our definition of the repository graph is applicable on any current mainstream VCS. To demonstrate that, we consider how to derive the nodes, edges, and the change sets on edges in three popular version control systems: git, svn, and CVS. Git and svn store each commit as a snapshot of the repository, so the commits correspond to the nodes in the repository graph. CVS on the other hand stores commits as the change set containing added lines and deleted lines. We can derive the contents of nodes by composing consecutive change sets. All the three version control systems record branching and merging, so edges are easy to find. Git and svn provide built-in differencing tools to calculate change sets, but they are not sufficient for our definition of  $\delta$  because they report changed lines separately as added lines and deleted lines. *ldiff* [8] calculates source code similarity metrics (such as best edit distance and cosine similarity) to match added lines and deleted lines and derive pairs of changed lines. We use *ldiff* to implement our definition of  $\delta$ . Since we can implement the repository graph on any mainstream VCS, we assume a code repository is represented as a repository graph in the following sections.

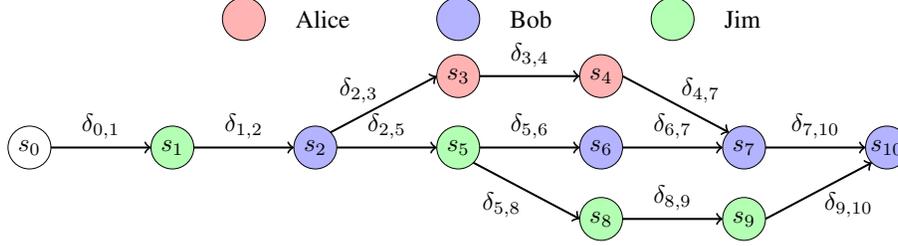


Fig. 1. An example of the repository graph. Nodes are source code revisions, denoted from  $s_0$  to  $s_{10}$ . The color of a node shows the author creating the revision. The virtual node  $s_0$  has no author information. Edges represent development dependencies between revisions.  $\delta_{i,j}$  on edge  $e_{i,j}$  is the code change from  $s_i$  to  $s_j$ .

### III. STRUCTURAL AUTHORSHIP

Structural authorship represents the development history of a line of code. We define structural authorship as a subgraph  $G_l$  of the repository graph  $G$  that includes only the revisions that change line  $l$  of code, and the development dependences between these revisions. We present a backward flow analysis algorithm on the repository graph  $G$  that extracts the structural authorship. Our analysis processes all lines in a file to provide sufficient context for programmers to view code history. After extracting structural authorship, analysis tools have access to all historical information of a line so that they are not limited to the last change of that line.

Our structural authorship model can be seen as a generalization of the current method that only reports the last change. Both our model and the current method stop searching the history of a line when the line is found to be added. The distinction is that our model can make use of the information in the set of changed lines  $\mathcal{C}$ , while the current method cannot.

#### A. Model definition

For a given line of code  $l$  appearing in a revision  $s_v$  (often the head revision), the *structural authorship* of the pair of  $(s_v, l)$  is defined to be a directed graph  $G_l = (V_l, E_l, \Delta_l)$ .  $V_l$  is the set of nodes that changed or added line  $l$ .  $E_l$  is the set of edges that represent development dependences between nodes in  $V_l$ .  $\Delta_l$  is a labeling of  $E_l$  that represents code changes. Before giving the formal definitions of  $V_l$ ,  $E_l$ , and  $\Delta_l$ , we first introduce notation to describe the relationships between nodes and then extend our definition of  $\delta_{i,j}$ .

We define  $s_i \rightarrow s_j$  if and only if there is a directed path in  $G$  from  $s_i$  to  $s_j$ . For the starting revision  $s_v$ , its ancestor set contains the potential revisions that could be in  $V_l$ . We define the ancestor set of a node  $s_i$  as

$$ance(s_i) = \{s_k \in V \mid s_k \rightarrow s_i\}$$

To determine what lines a node  $s_i$  has changed or added, we define the total effect of  $s_i$  as:

$$\begin{aligned} \delta_i &= \bigcup_{s_k \in pred(s_i)} \delta_{k,i} \\ \mathcal{D}_i &= \bigcup_{s_k \in pred(s_i)} \mathcal{D}_{k,i} \\ \mathcal{A}_i &= \bigcup_{s_k \in pred(s_i)} \mathcal{A}_{k,i} \end{aligned}$$

$$\mathcal{C}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{C}_{k,i}$$

Now we can define  $V_l$  as the set of revisions of  $s_v$  and its ancestors that add or change the line  $l$ :

$$V_l = \{s_j \in (ance(s_v) \cup \{s_v\}) \mid l \in (\mathcal{A}_j \cup \mathcal{C}_j)\}$$

An edge in  $E_l$  represents a path that does not go through nodes in  $V_l$ . For  $s_i$  and  $s_j$  such that  $s_i \rightarrow s_j$ , we define  $s_i \xrightarrow{V_l} s_j$  if and only if there exists one or more directed paths from  $s_i$  to  $s_j$  and none of the intermediate nodes on the path are in  $V_l$ . This relationship is used to describe the development dependency between two nodes in  $V_l$ . We can define  $E_l$  as:

$$E_l = \{e_{i,j} \mid (s_i, s_j \in V_l) \wedge (s_i \xrightarrow{V_l} s_j)\}$$

Note that a single  $e_{i,j}$  in  $E_l$  can result from multiple directed paths in the original  $G$ .

We now extend our definition of  $\delta_{i,j}$  to the case where  $s_i \rightarrow s_j$  so that  $\delta$  can be used to describe  $\Delta_l$ . If  $\langle s_i, s_{k_1}, \dots, s_{k_m}, s_j \rangle$  is a directed path from  $s_i$  to  $s_j$ , then

$$\delta_{i,j} = \delta_{k_m,j} \circ \delta_{k_{m-1},k_m} \circ \dots \circ \delta_{i,k_1}$$

Note that the specific choice of the path is not important because the result of composition of change sets is path independent.  $\Delta_l$  then can be defined as

$$\Delta_l = \{\delta_{i,j} \mid e_{i,j} \in E_l\}$$

We illustrate our subgraph definition with an example. In the repository graph  $G$  shown in Figure 1, suppose we have the following scenario: Line  $l$  was first introduced into the project by Bob in revision 2 ( $s_2$ ). Alice changed  $l$  in revisions 3 and 4 in her branch. Jim changed  $l$  in revision 9 in his branch. Bob merged Alice's branch in revision 7. Since Alice and Jim made independent changes to  $l$ , when Bob finally tried to merge Jim's branch, Bob had to solve the conflict by taking either Alice's change or Jim's change; we assume that Bob took Jim's change. The structural authorship  $G_l$  is shown in Figure 2.

#### B. Backward flow analysis

We calculate the structural authorship graphs in two steps. In the first step, we use a backward flow analysis to calculate

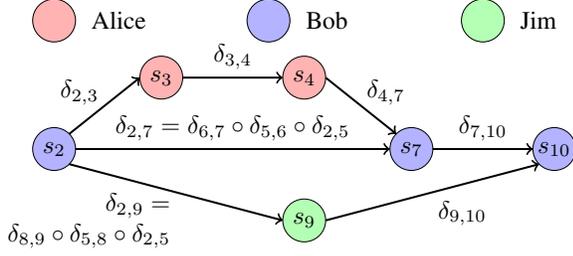


Fig. 2. An example of the structural authorship graph. Nodes in  $V_l = \{s_2, s_3, s_4, s_7, s_9, s_{10}\}$  changed or added line  $l$ . Edges represent extended development dependencies between revisions.  $\delta_{i,j}$  on edge  $e_{i,j}$  is the extended code change from  $s_i$  to  $s_j$ .

```

input :  $V, E, F$ , and  $s_v$ 
output:  $\{V_l | l \in F\}$ 
// The live lines that can reach  $s_v$ 
1  $liveLines[s_v] \leftarrow F$ ;
2 for  $s_i \in (ance(s_v) \cup \{s_v\})$  in reverse topological order in  $G$ 
  do
    // Phase 1: calculate  $\delta$  for  $s_i$ 
    3 for  $s_k \in pred(s_i)$  do
      4    $\delta_{k,i} \leftarrow ldiff(s_k, s_i, F)$ ;
      5    $\delta_i \leftarrow \delta_i \cup \delta_{k,i}$ ;
    // Phase 2: update  $V_l$ 
    6 for  $l \in liveLines[s_i]$  do
      7   if  $l \in \mathcal{A}_i \cup \mathcal{C}_i$  then
      8      $V_l \leftarrow V_l \cup \{s_i\}$ ;
    // Phase 3: pass live lines to preds
    9 for  $s_k \in pred(s_i)$  do
      10   for  $l \in liveLines[s_i]$  do
      11     if  $l \notin \mathcal{A}_{k,i}$  then
      12        $liveLines[s_k] \leftarrow liveLines[s_k] \cup \{l\}$ ;
      13    $liveLines[s_i] \leftarrow \emptyset$ ;

```

Fig. 3. *S-Author*: An algorithm that extracts  $V_l$  for all lines of code in file  $F$  starting at revision  $s_v$ .

$V_l$ . In the second step, a depth first search is used to calculate  $E_l$  and  $\Delta_l$ . In our repository graph abstraction,  $V$  and  $E$  can be directly accessed through API of the underlying VCS, but we have to use *ldiff* to calculate  $\Delta$  in our analysis.

In the first step, we use the backward flow analysis shown in Figure 3 to extract  $V_l$  from the repository graph  $G$ . We perform our analysis on all of the lines in a file  $F$  rather than an individual line  $l$  for two reasons. First, by processing all lines in  $F$  together, we can order the computation so that we neither make redundant calls to *ldiff* nor store the results of *ldiff*. Second, programmers usually want to view code history in a context, so presenting histories of several lines together is more useful.

Our algorithm calculates dataflow information for each node and adds nodes to  $V_l$ . For node  $s_i$ , its dataflow information records the live lines that can reach the starting node  $s_v$  from  $s_i$  before being deleted. We use a map *liveLines* that associates a node to a set of live lines to efficiently update the dataflow information. At the beginning, all lines in  $F$  are live (line 1).

Because  $G$  is acyclic, the traditional work list algorithm for dataflow analysis is not necessary in our case. It is sufficient

to visit each node from  $s_v$  in the reverse topological order of  $G$  (line 2). For each node  $s_i$ , there are three major phases: calculating change sets (lines 3-5), updating  $V_l$  (lines 6-8) and passing live lines to the predecessors of  $s_i$  (lines 9-12).

In phase 1, we call *ldiff* to calculate a subset of  $\Delta$  that are sufficient and necessary for the next two phases. In phase 2, for each live line  $l$ , we determine whether  $s_i$  is in  $V_l$  or not (line 7). In phase 3, we check whether the current live lines will still be live in each predecessor  $s_k$  of  $s_i$  (line 11). It is possible that  $l$  will be dead along one branch, but still be live along another branch.

The analysis finishes after it visits the virtual node  $s_0$ . As a special case, we can add  $s_0$  to  $V_l$  to represent the state where  $l$  has not yet been introduced into the repository. For any  $l \in F$ ,  $V_l$  are the nodes in the structural authorship graph.

The memory used for the results of *ldiff* in phase 1 can be freed after the phase 3 in this iteration. *ldiff* produces the  $\delta$  between two files and has a relative high time complexity, quadratic in terms of the size of the files [8]. Caching the results of *ldiff* can avoid redundant calls to *ldiff*. But we estimate that caching the results of *ldiff* on a large code repository could take a few gigabytes of memory, which is too much for a built-in tool for a VCS.

In the second step, for each node that we have determined is in  $V_l$ , we can do a depth first search in  $G$  to calculate  $E_l$  and  $\Delta_l$  according to our definitions.

The running efficiency of our algorithms both depends on the actual sizes of structural authorship graphs.  $G_l$  could be as large as  $G$  in theory. However,  $G_l$  is usually small in practice (Section 5.1) and our algorithms show good performance.

#### IV. WEIGHTED AUTHORSHIP

The structural authorship graph  $G_l$  represents the complete development history of a line of code  $l$ . However, existing analysis tools typically operate on numerical or ordinal features rather than a graph, so we wish to provide summaries of this information in a form such tools can consume. We define the *weighted authorship* of  $l$  to be a vector of author contribution weights. For each author, we can then use the weighted authorship to determine their contribution, model their familiarity of the line, or estimate their efforts spent on the line. This type of summary information is often used to analyze software quality [5, 32], help familiarize new developers [13], and estimate software development cost [26].

##### A. Model description

For a line of code  $l$ , we define the *weighted authorship*  $W_l$  as a vector  $(c_1, c_2, \dots, c_m)$ . Each element  $c_i$  is the percentage of contribution made by developer  $i$ ; elements in  $W_l$  sum to 1.  $m$  is the total number of developers that changed  $l$ . By examining  $G_l$ , we can determine the value of  $m$ . We define each  $c_i$  to be the number of characters attributed to developer  $i$  divided by the total number of characters in  $l$ . For example, if Alice, Bob and Jim are developers 1, 2 and 3,  $W_l = (30\%, 20\%, 50\%)$  means that Alice, Bob, and Jim contribute 30%, 20%, 50% of the line respectively. We use

```

input :  $l, V_l, E_l$ , and  $\Delta_l$ 
output:  $attr$ : maps a character in  $l$  to its attributed node
1 Let  $s_v$  be the last node that changed  $l$ ;
  // The live characters that can reach  $s_v$ 
2  $liveC[s_v] \leftarrow l$ ;
3 for  $s_i \in V_l$  in reverse topological order in  $G_l$  do
4   if  $|pred(s_i)| == 1$  then
5     //  $s_i$  is created by a normal commit
6     Let  $s_k$  be the element in  $pred(s_i)$ ;
7      $chars \leftarrow AC\text{-}BestEdit(l, \delta_{k,i})$ ;
8     for  $c \in liveC[s_i] \cap chars$  do
9       if ( $c \notin attr.keys()$ ) or ( $tstamp(s_i) < tstamp(attr[c])$ )
10        then
11           $attr[c] \leftarrow s_i$ ;
12           $liveC[s_k] \leftarrow liveC[s_k] \cup (liveC[s_i] - chars)$ ;
13   else
14     //  $s_i$  is created by a merge commit
15     for  $s_k \in pred(s_i)$  do
16        $chars \leftarrow AC\text{-}BestEdit(l, \delta_{k,i})$ ;
17        $liveC[s_k] \leftarrow liveC[s_k] \cup (liveC[s_i] - chars)$ ;

```

Fig. 4. *W-Author*: An algorithm calculating the attribution map for  $l$

characters as the unit of contribution because it is simple and avoids being dependent on the programming language used. While we do not consider the semantics of the code, we do collapse white space to minimize the effects of simple formatting changes. We do not isolate the affect of each of these choices, however the experiments in the following section show that these choices produce satisfactory results.

### B. Algorithm

We calculate  $W_l$  based on  $G_l$ . We first attribute each character in  $l$  to the node that introduced that character and then attribute each node to the appropriate developer. We define the attribution map  $attr$  to maintain this character-to-node attribution. The node-to-developer attribution can be done by checking the author label of each node.

We use the algorithm shown in Figure 4 to compute the attribution map  $attr$ . The idea is to attribute a character to the node in which the character is added or changed. The algorithm first finds the last revision  $s_v$  that changed  $l$ ; this  $s_v$  is the starting point of our algorithm (line 1). For each node in  $G_l$ , we maintain the live characters that can reach  $s_v$  before being deleted. All characters in  $l$  at  $s_v$  are live (line 2). We visit each node in  $G_l$  in reverse topological order. For each node  $s_i$ , we distinguish whether  $s_i$  is created by a normal commit or a merge commit by checking the number of its predecessors (line 4). In both case, we define *AC-BestEdit* to calculate the set of characters added or changed in this node (line 6 and 13). These characters are not passed to the predecessors of  $s_i$ . For a normal commit, we update the attribution map and pass the live characters (lines 5-10). For a merge commit, we only pass the live characters (lines 12-14).

*AC-BestEdit* adapts the Wagner-Fischer algorithm [36] for computing the best edit distance to calculate the set of characters in  $l$  added or changed by  $s_i$ . In the Wagner-Fischer algorithm, the best distance is defined as the minimum number of steps needed to change a source string to a target

string. Each step can be adding, deleting, or substituting a character. The algorithm computes a shortest path and returns the minimal number of steps. For an edge  $e_{k,i} \in E_l$ , the string in  $s_k$  is the source string and the string in  $s_i$  is the target string. *AC-BestEdit* calculates the shortest path to change the source string to the target string using Wagner-Fischer, and it returns the set of characters added or changed by  $s_i$ .

A normal commit has a single predecessor  $s_k$ . A character that is added or changed in this node may be also added or changed independently in other nodes (in other branches). Since characters are the unit of contribution, we do not divide the contribution of a character among the multiple commits. In this case, we attribute the character to the node with the earlier commit timestamp (line 8).

For a merge commit, we assume that the commit is either produced during an automatic merge by the VCS or manual selections from one of the multiple branches; therefore a merge commit does not introduce new characters. Since the added or changed characters in one branch actually come from other branches, we just ignore these characters in this merge commit and attribute them to other branches.

The performance of the algorithm depends on the size of  $G_l$ . As we will discuss in the next section, the size of  $G_l$  is usually small. In our experience, running this algorithm on all lines in a file finishes in around second.

## V. EVALUATION

We implemented our structural authorship model and weighted authorship model in a new git built-in tool: *git-author*. *git-author* uses a syntax similar to that of *git-blame* so has a familiar feel to current users of git. We designed two experiments to compare our new authorship models to the current model that only reports the last change to a line. In the first experiment, we ran *git-author* on five open source code repositories to study the number of lines that were changed in multiple commits and the number of lines that were changed by multiple authors. This experiment shows that *git-author* can recover more information than *git-blame* on about 10% of lines. The results show that most lines are touched only by one author in one commit and the cooperation between developers is restricted to small regions of code. We hypothesized that these small regions of code contain rich information about the software development process and that analysis tools can benefit from this extra information. We conducted our second experiment to verify this hypothesis. Our second experiment evaluated whether the additional information would be useful to build a better analysis tool. We built a new line-level model for source code bug prediction and compared it with the best previously report work on a file-level model [22]. We found that our line-level model consistently performed better than the file-level model. This demonstrates that our new authorship models can help build better analysis tools.

### A. Multi-author study

In this experiment, we ran *git-author* on the following five open source projects: Dyninst [31], the Apache HTTP

Repository	Multi. Commits	Multi. Authors	# of lines
Dyninst	53K (12.11%)	40K (9.12%)	434K
Httpd	27K (10.90%)	20K (8.15%)	247K
GCC	279K (8.08%)	217K (6.27%)	3454K
Linux	1440K (9.69%)	1072K (7.22%)	14857K
GIMP	122K (12.82%)	78K (8.12%)	955K

TABLE I  
NUMBER OF LINES CHANGED BY MULTIPLE COMMITS AND MULTIPLE AUTHORS. THE SECOND COLUMN SHOWS THE NUMBER OF LINES CHANGED IN MULTIPLE COMMITS AND THE PERCENTAGE THEY ACCOUNT FOR IN THE REPOSITORY. THE THIRD COLUMN SHOWS THE SAME INFORMATION FOR LINES THAT CHANGED BY MULTIPLE AUTHORS.

server [1], GCC [15], the Linux Kernel [24], and Gimp [16], extracting the structural authorship for each line of the code. We then counted the number of nodes and the number of authors in each structural authorship graph. Note that we did not run *git-blame* on the five projects because *git-blame* would output only one commit and one author for each line of code.

The results are shown in Table I. About 10% of lines are changed by multiple commits and about 8% of lines are changed by multiple authors. *git-author* produces more information than *git-blame* on these lines.

### B. Line-level bug prediction

Our second experiment evaluated whether the information provided by *git-author* would be helpful to build a better bug prediction model. We show that we can build a line-level bug prediction model that is more effective than the best previously reported work on a file-level model by Kamei, Matsumoto et al. [22]. To the best of our knowledge, we are the first project to try to predict bugs at the line level.

We first give an overview of bug prediction and our experiment. We then introduce our new line-level model and the file-level model we compared it to. We discuss our data sets and the metrics used to evaluate the models. Finally, we present our results.

1) *Overview*: Many research efforts have been dedicated to source code bug prediction to prioritize software testing [18, 20, 22, 29, 30]. Two comprehensive surveys are from Arisholm, Briand, et al. [2] and D’Ambros, Lanza et al. [9].

Three decisions affect the performance of a bug prediction model: the granularity of prediction, a set of bug predictors, and a machine learning technique that trains the model and predicts bugs. Using *git-author* changes the granularity of prediction to the line level and introduces new bug predictors. We do not explore the influence of different machine learning techniques as it is beyond the scope of this paper.

Most of the existing source code bug prediction models predict at the granularity of a source file [18, 28] or even a module [29, 30]. The disadvantage of coarse-grained prediction models is that, even if the prediction results are accurate, developers still have to spend effort to locate the bugs within a module or file. Predicting at a finer granularity, such as at the method level can help to reduce the problem [20, 23]. Our line-level model can locate the suspected lines and help focus

Level	Predictor name	Definition
Line	WA	Weighted authorship defined in Section 4
	NOA	# of authors
	NOC	# of commits
	LEN	Length of the line
	VAR	Variance of the length of the line across all commits in $G_l$
	FIX	# of times a line involved in a bug-fix commit
	REF	# of times a line involved in a refactoring commit
	COM	Whether a line is a comment
	AGE	The age of the line
File [22]	Codechurn	Sum of (added lines of code - deleted lines of code)
	LOCAdd	Sum of added lines of code over all revisions
	LOCDel	Sum of deleted lines of code over all revisions
	Revisions	# of revisions
	Age	The age of the file
	BugFixes	# of times a file involved in a bug-fix commit
	Refactor	# of times a file involved in a refactoring commit

TABLE II  
BUG PREDICTORS USED IN THE STUDY.

testing efforts. It uses the development history of lines of code provided by *git-author* to make prediction. Note that since the development history of a line of code produced by *git-blame* is incomplete, it is impractical to do line-level prediction with *git-blame*. We compared our line-level model to the file-level model because predicting at a file level is well understood.

Two types of bug predictors are commonly used: product predictors that summarize code in the predicting snapshot [41] and process predictors that summarize the history of the predicting snapshot [28]. The process predictors have been shown to be more effective than the product predictors [22, 28]. In our experiment, most of our predictors are process predictors.

Many machine learning techniques have been adopted for bug prediction. However, previous studies have shown that the influence of bug predictors on the final prediction results is much larger than the chosen machine learning technique [2, 22]. Therefore, we selected linear learning techniques for both our line-level model and the file-level model. We do not believe this choice will have a noticeable effect on our results.

2) *Models*: The goal of our line-level model is, given a line of code, to output the probability that the line is buggy. Based on these outputs, a developer could prioritize testing of the software to the lines with higher probabilities of being buggy. We used a linear SVM as the learning technique in our line-level model [12]. The predictors in our new model are shown in Table II. We introduce new predictors including the weighted authorship, the length of the line, the variance of the length of the line across all commits in  $G_l$ , and whether the line is a comment. The other predictors were adapted from existing file-level predictors. We compute the values of these line-level predictors from the outputs of *git-author*.

We compared our line-level model to the file-level model from Kamei, Matsumoto et al. [22]. Their model outputs the predicted fault density when given a file. They compared the prediction results of using process predictors and product

predictors with three learning techniques: linear regression [10], regression tree [7], and random forest [6]. Their results showed that using process predictors produced consistently better results than using product predictors and combining them together did not provide further advantages. Therefore, we implemented the file-level process predictors listed in Table II. We chose the logistic regression [12], one type of linear regression, as the learning technique of the file-level model to match the linear SVM used in our line-level model.

Note that when evaluating the effects of *git-author*, it would have been preferable to use the same machine learning technique in the line-level model and the file-level model. However, because the outputs of the line-level model and the file-level model are different, we cannot use the exact same learning technique. Therefore, we can only try to minimize the effects on performance from the factors rather than *git-author*.

3) *Data collection*: We are unaware of existing bug prediction data sets with line-level predictors; instead we generated new such data sets. Producing a bug prediction data set takes two steps. We first create a *bug map* from a bug record in the bug database to the pair of commits that caused the bug and fixed the bug. We then choose a time point, typically a release, and use the bug map to produce data instances for this snapshot. The second step is repeated at several different release time points so that we could do cross release prediction.

For the first step, we used the SZZ algorithm [35] to find buggy commits and the corresponding *fixing commits* that fixed the bugs in the Apache HTTP server repository. The quality of the results in this step is improved by *Relink* [37], which addresses the problem of missing valid mappings in the original SZZ algorithm [4].

In the second step, we projected the *bug map* onto the chosen snapshot. A bug is relevant to the snapshot if and only if the snapshot is inside the time interval between the buggy commit and the fixing commit. For each relevant bug, we first produced line-level data, and then summarized the data into file-level data. We assume the lines that are deleted or changed in the fixing commit are the buggy lines. Two methods can be used to summarize the line-level data. We can either count all buggy lines as a single bug or count the lines separately. The first method assumes that it takes the same effort to fix every bug, while the second method takes this factor into consideration. We adopted both methods and produced two data sets.

We collected data for seven releases in the Apache HTTP server project and produced two data sets described above. The first data set is denoted as “Bug count” and the other one is denoted as “Line count”. Table III summarizes our data sets.

4) *Evaluation metrics*: Many metrics are used to evaluate bug prediction models. The most commonly used metrics include precision and recall [29, 30], the area under the curve (AUC) of ROC curves [27, 28], and effort-aware metrics [22, 25]. Comparison studies have shown that the choices of metrics can significantly affect the performance of prediction models [2, 9]. The difference of performance on metrics does

Release	# of files	# of bugs	SLOC	# of buggy lines
2.1.1	305	129	177K	670
2.2.0	319	171	202K	746
2.2.6	320	167	205K	708
2.2.10	321	172	207K	664
2.3.0	383	179	207K	680
2.3.10	372	195	218K	747
2.4.0	362	181	223K	555

TABLE III  
SUMMARY OF THE DATA SETS. EACH ROW IN THE TABLE SUMMARIZES THE NUMBER OF FILES, BUGS, LINES OF CODE, AND BUGGY LINES IN A RELEASE SNAPSHOT OF APACHE.

not mean inconsistent results because different metrics are designed to answer different questions. We use the effort-aware metrics because they are domain specific metrics for bug prediction. They measure not only the accuracy of the predicting results but also the efforts needed to fix the bugs.

In our study, we use two effort-aware metrics:  $P_{opt}$ , which measures the closeness of a model to the optimal file level model [25] and cost-effectiveness ( $CE$ ), which measures the advantages of that model over a random prediction model [2]. The idea of effort-aware metrics is that a developer can first test or inspect the most suspicious lines or the files with largest fault densities and see how many percent of bugs can be found. The assumption is that the effort needed to test a piece of code is roughly proportional to the size of the code [2]. Using the percent of lines tested as the x-axis and the percent of bugs covered as the y-axis, we can draw a curve to visualize the performance of a model. We denote the area under the curve of a model  $m$  as  $AUC(m)$ .  $P_{opt}$  and  $CE$  can be defined as:

$$P_{opt}(m) = 1 - \frac{AUC(FileOptimal) - AUC(m)}{AUC(m) - AUC(Random)}$$

$$CE(m) = \frac{AUC(m) - AUC(Random)}{AUC(FileOptimal) - AUC(Random)}$$

In the above formulas, the file optimal model tests files in decreasing order of the fault densities. It represents the upper bound of a file level model. The random model orders the files randomly. We use the average performance of the random model in the  $CE$  formula, which is a straight line from (0, 0) to (1, 1). For both  $P_{opt}$  and  $CE$ , larger values mean better performance. When the values are larger than 1, the model  $m$  performs better than the optimal file-level model.

5) *Results*: We performed cross release prediction on our data set. We chose cross release prediction instead of cross-validation inside a release because the cross-release prediction represents the real practice of how a bug prediction model is used. We used *Liblinear* to do training and prediction on our two data sets [12]. We denote our line-level model as  $lm$ , the file-level model as  $fm$ , and the optimal file-level model as  $fm_o$ .

In the “Bug count” data set, we need to aggregate line-level prediction results into the bug count. We provide three interpretations for our line level models. The first one is that we can identify a bug as long as any line comprising bug is identified. This is the optimistic interpretation and represents the maximal benefits that can be acquired by using our line-

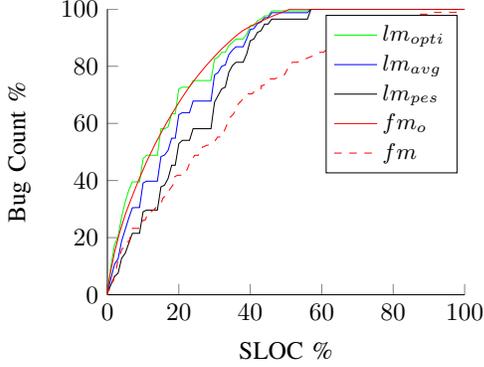


Fig. 5. Cross release prediction from 2.2.10 to 2.3.0 on “Bug count” data set. The x-axis is the percentage of source line of code to test. The y-axis is the percentage of bugs that can be identified.

level model. The second one is that we take partial credit when we identify a buggy line. For example, if we identify one buggy line for a five-line bug, we say we find 20% of a bug. This is the average interpretation and assumes that the more information about a bug is provided, the more likely the bug can be identified. The third one is that only after we identify all buggy lines of a bug, we cover the bug. This is the pessimistic interpretation. We denote the three views as  $lm_{opti}$ ,  $lm_{avg}$ , and  $lm_{pes}$ .

The results for the “Bug count” data set are shown in Table IV. Our results of  $P_{opt}(fm)$  are consistent with the results shown by Kamei, Matsumoto et al. [22]. The results of  $CE(fm)$  are slightly better but still consistent with the results shown by Arisholm, Briand et al. [2]. Therefore, we believe that our implementation of  $fm$  is comparable to other implementations and that we can compare our  $lm$  to this implementation of  $fm$ .

The optimistic interpretation and the average interpretation are consistently much better than the file model in both  $P_{opt}$  and  $CE$ . The pessimistic interpretation loses to the file model slightly in two rounds of prediction but has a much higher mean value. All the three interpretations have much smaller standard deviation than the file model, so prediction results are more stable on line level. Notice that the value of  $P_{opt}(lm_{opti})$  and  $CE(lm_{opti})$  in row “2.3.10 → 2.4.0” are larger than 1, which shows that the performance of the line level model can even exceed the upper bound of file level models.

Figure 5 shows the prediction results of training on release 2.2.10 and predicting on release 2.3.0. If we only test a small amount of code, the  $lm_{opti}$  is actually better than the  $fm_o$ , but the  $lm_{pes}$  is a little bit worse than the  $fm$ . As we test more code, the three interpretations of the line-level model are consistently better than the  $fm$ .

The “Bug count” data set assumes that every bug involves the same amount of work to fix. We use the “Line count” data set to measure how many buggy lines can be covered during testing. The overall results are shown in Table V and confirm that the line level model consistently performs better

Train → Predict	$P_{opt}$		$CE$	
	$lm$	$fm$	$lm$	$fm$
2.1.1 → 2.2.0	0.9148	0.8113	0.7925	0.5404
2.2.0 → 2.2.6	0.9425	0.7704	0.8578	0.4321
2.2.6 → 2.2.10	0.9470	0.7860	0.8658	0.4579
2.2.10 → 2.3.0	0.9153	0.8288	0.7834	0.5624
2.3.0 → 2.3.10	0.8660	0.7711	0.6590	0.4173
2.3.10 → 2.4.0	0.9343	0.8860	0.8299	0.7050
Mean	0.9200	0.8089	0.7981	0.5192
Standard Deviation	0.0271	0.0404	0.0692	0.0988

TABLE V  
RESULTS OF “LINE COUNT” DATA SET.

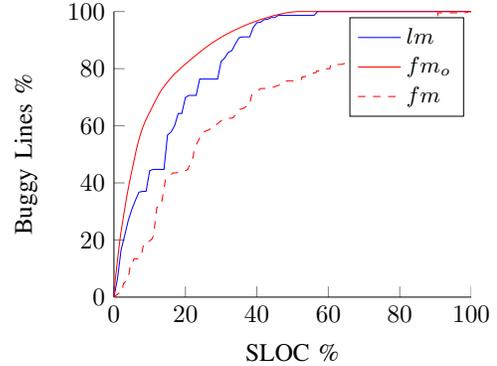


Fig. 6. Cross release prediction from 2.2.10 to 2.3.0 on “Line count” data set. The x-axis is the percentage of source line of code to test. The y-axis is the percentage of buggy lines that can be identified.

in both  $P_{opt}$  and  $CE$ . Figure 6 shows the results of training on release 2.2.10 and predicting on release 2.3.0 in the “Line count” data set. The line level model performs better than the file level model over all ranges of the curve.

In summary, our two experiments confirm the effectiveness of our new authorship models. The first experiment shows that *git-author* provides more information than *git-blame* by the structured authorship model. The second experiment shows that the information is useful to build better analysis tools.

## VI. RELATED WORK

Three types of studies are related to our work: code authorship extraction and visualization [14, 21], software quality and maintenance analysis using code authorship [5, 13, 32], and mining software repositories for histories of source code entities [3, 17, 19, 34, 40]. The first type is similar to our work in terms of the final goal that is to present authorship information to users, but the approaches and the granularity are different. The second type consumes authorship information to analyze software quality or to improve developer familiarization. The third type shares a similar approach with our work. We now discuss each type of studies in more detail.

Syde [21] is a system built on Eclipse that collects every change made by developers. Syde records changes made by developers when they try to compile the code. They then define the owner of a file as the developer making the most number of changes. With Syde’s change log, refined ownership can

Train → Predict	$P_{opt}$				$CE$			
	$lm_{opti}$	$lm_{avg}$	$lm_{pes}$	$fm$	$lm_{opti}$	$lm_{avg}$	$lm_{pes}$	$fm$
2.1.1 → 2.2.0	0.9695	0.9392	0.9023	0.8321	0.9132	0.8243	0.7220	0.5221
2.2.0 → 2.2.6	0.9884	0.9632	0.9297	0.8166	0.9664	0.8935	0.7965	0.4693
2.2.6 → 2.2.10	0.9997	0.9706	0.9339	0.8453	0.9990	0.9148	0.8082	0.5509
2.2.10 → 2.3.0	0.9647	0.9325	0.8965	0.8716	0.8956	0.8007	0.6943	0.6208
2.3.0 → 2.3.10	0.9664	0.9275	0.8848	0.8870	0.8961	0.7756	0.6433	0.6504
2.3.10 → 2.4.0	<b>1.0013</b>	0.9665	0.9245	0.9267	<b>1.0040</b>	0.8979	0.7700	0.7769
Mean	0.9817	0.9499	0.9120	0.8632	0.9457	0.8511	0.7391	0.5984
Std. Dev.	0.0154	0.0173	0.0184	0.0368	0.0460	0.0532	0.0585	0.0998

TABLE IV

RESULTS OF “BUG COUNT” DATA SET. THE TWO BOLD NUMBERS IN ROW “2.3.10 → 2.4.0” ARE LARGER THAN ONE INDICATING THAT THE PERFORMANCE OF OUR LINE LEVEL MODEL CAN EXCEED THE UPPER BOUND OF ANY FILE LEVEL MODEL.

be extracted on file level. Our work differs from Syde in two ways. First, our authorship models are on line level. Second, our models are applicable to existing repositories and do not require extra compile-time information.

Rahman and Devanbu [32] analyze the relationship between code authorship and the number of defects in four open source projects. They define a file-level authorship model that computes the percentage of lines owned by each developer using *git-blame*. Fritz, Ou and et al. [13] use code authorship data and developer interaction data to model source code familiarity. Their authorship model is at the source code element level including class, method and field. We believe these studies can benefit from our new authorship models by aggregating accurate line-level authorship information into the corresponding granularities.

Kenyon [3], APFEL [40], Beagle [17] and Hstorage [19] mine software repositories to produce the history of code entities at granularities finer than files. Their goal is to produce rich semantics for code changes including adding, deleting, modifying, renaming and moving. Our work differs from theirs in two perspectives. First, these tools use heavy-weight semantic analysis for rich semantics of code changes. Therefore, results have to be stored in a relational database for later queries. On the contrary, our tool is light-weighted and can produce results on the fly. Second, our tool is designed to be a built-in tool of git, so it is easy to use for users who are familiar with git.

Servant and Jones [34] define the *history slice* to represent the history of a line of code. Like our structural authorship, it contains the revisions that changed the line. Unlike our model, it ignores branches and assumes that a later revision is based only on a prior revision. Therefore, two independent revisions in different branches can be dependent in history slice.

## VII. CONCLUSION

We have presented two line-level authorship models: the structural authorship, which represents the complete development of a line of code, and the weighted authorship, which summarizes the structural authorship to produce author contribution weights. Our two authorship models overcome the limitations of the current methods that only report the last change to a line of code. We define the repository graph as a graph abstraction for a code repository and define a backward

flow analysis on the repository graph that derives the structural authorship. Another backward flow analysis is used on the structural authorship to compute the weighted authorship. We have implemented our two authorship models in a new git built-in tool *git-author*. We have evaluated *git-author* in two experiments. In the first experiment, we ran *git-author* on five open source projects and find that *git-author* can recover more information than *git-blame* on about 10% of the lines. In the second experiment, we built a line-level model for bug prediction based on the output of *git-author*. We compared our line-level model with a representative file-level model and found that our line-level model is consistently better than the file-level model on our data sets. These results show that our new authorship models can produce more information than the existing methods and that information is useful to build a better analysis tool.

## ACKNOWLEDGEMENTS

We thank Weiyang Chiew for his help in extracting the bug data sets. This work is supported in part by Department of Energy grants DE-SC0003922 and DE-SC0002155, National Science Foundation Cyber Infrastructure grants OCI-1032341, OCI-1032732, and OCI-1127210; and Department of Homeland Security under AFRL Contract FA8750-12-2-0289.

## REFERENCES

- [1] Apache Software Foundation. Apache http server, <http://httpd.apache.org>.
- [2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, Jan. 2010.
- [3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, Lisbon, Portugal, Sep. 2005.
- [4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, Amsterdam, The Netherlands, Aug. 2009.
- [5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don’t touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT*

- symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE), Szeged, Hungary, Sep. 2011.
- [6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.
  - [7] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.
  - [8] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*, Minneapolis, Minnesota, USA, May 2007.
  - [9] M. D’Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.
  - [10] A. L. Edwards. *Introduction to Linear Regression and Correlation*. W.H.Freeman & Co Ltd, 1976.
  - [11] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, Waikiki, Honolulu, HI, USA, May 2011.
  - [12] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, June 2008.
  - [13] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.
  - [14] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSSE)*, Lisbon, Portugal, Sep. 2005.
  - [15] GNU Project. Gcc: The gnu compiler collection, <http://gcc.gnu.org>.
  - [16] GNU Project. The gnu image manipulation program, <http://www.gimp.org>.
  - [17] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, Feb. 2005.
  - [18] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.
  - [19] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSSE-EVOL)*, Szeged, Hungary, Sep. 2011.
  - [20] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, May 2012.
  - [21] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, Vancouver, Canada, May 2009.
  - [22] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2010.
  - [23] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*, Minneapolis, Minnesota, USA, May 2007.
  - [24] Linux Kernel Project. Linux kernel, <http://www.kernel.org>.
  - [25] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE)*, Vancouver, British Columbia, Canada, May 2009.
  - [26] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, Kansas, USA, Nov. 2011.
  - [27] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan. 2007.
  - [28] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.
  - [29] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.
  - [30] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, San Jose, CA, USA, Nov. 2010.
  - [31] Paradyne Project. Dyninst: Putting the Performance in High Performance Computing, <http://www.dyninst.org>.
  - [32] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, HI, USA, May 2011.
  - [33] N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS)*, Leuven, Belgium, Sep. 2011.
  - [34] F. Servant and J. A. Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, Cary, North Carolina, Sep. 2012.
  - [35] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, May 2005.
  - [36] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan. 1974.
  - [37] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, ESEC/FSE ’11, Szeged, Hungary, Sep. 2011.
  - [38] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, Sep. 2011.
  - [39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.
  - [40] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange (eclipse)*, Portland, Oregon, Oct. 2006.
  - [41] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE)*, Minneapolis, Minnesota, USA, May 2007.

# Investigating the Impact of Code Smells on System's Quality:

## An Empirical Study on Systems of Different Application Domains

Francesca Arcelli Fontana  
*Department of Comp. Science*  
*University of Milano Bicocca*  
*Milano, Italy*  
*arcelli@disco.unimib.it*

Vincenzo Ferme  
*Department of Comp. Science*  
*University of Milano Bicocca*  
*Milano, Italy*  
*info@vincenzoferme.it*

Alessandro Marino  
*Department of Comp. Science*  
*University of Milano Bicocca*  
*Milano, Italy*  
*a.marino4@campus.unimib.it*

Bartosz Walter  
*Faculty of Computing*  
*Poznań University of Technology*  
*Poznań, Poland*  
*bartosz.walter@cs.put.poznan.pl*

Paweł Martenka  
*Faculty of Computing*  
*Poznań University of Technology*  
*Poznań, Poland*  
*pawel.martenka@cs.put.poznan.pl*

**Abstract** - There are various activities that support software maintenance. Program comprehension and detection of design anomalies and their symptoms, like code smells and anti patterns, are particularly relevant for improving the quality and facilitating evolution of a system. In this paper we describe an empirical study on the detection of code smells, aiming at identifying the most frequent smells in systems of different domains and hence the domains characterized by more smells. Moreover, we study possible correlations existing among smells and the values of a set of software quality metrics using Spearman's rank correlation and Principal Component Analysis.

**Keywords**-software maintenance; software evolution; code smell detection; metric and smell correlations; domain-dependent analysis.

### I. INTRODUCTION

A great interest has been observed in the literature towards the identification of parts of the code that need to be improved. In particular, code smells defined by Fowler have received a great attention [9].

In this paper we perform an analysis on object-oriented Java systems of different application domains, to identify possible relations between some features of the software systems and the domains they belong to. We focus our attention on the identification of code smells in different systems with the aim of answering the following question:

RQ 1) Which are the most frequent smells detected in different systems of different application domains? Is there a domain that usually contains more smells of a particular kind?

The answer to this question can provide hints how to adjust sensitivity thresholds in code smell detectors with respect to different applications domains, in order to reduce the number of misclassified smell instances and improve accuracy of the detection algorithms. Moreover, code smells in different domains can require different refactorings aimed at removing them. We believe that application domain is another factor that contributes to a more holistic approach to code smell detection and removal.

Next, we focus our attention on the investigation of the possible correlations existing among smells and the values of a of metrics defined for software quality evaluation, and we try to answer the following question:

RQ 2) What can we observe according to the metric values of systems with more smells? Is the code smell distribution correlated to the values of metrics? Do significant correlations exist among smells and metrics? Do low/high metric values always indicate code smells?

By answering this question we found interesting results that could help developers to focus their attention on the smells that contribute the most to deterioration of the system quality. Many metrics can be used as relevant indicators of good or bad quality of the system. Through our analysis, we aim to explore if some metrics could be used as good indicators of the presence of some particular smells and seen as predictors for software evolution and maintainability.

We also noticed that the available code smell detection tools ignore information related to the domain, design or the size of the system. We have outlined the importance of this aspect in the paper. Other factors important for improving

detection accuracy include the selection of metrics and the way they are combined together.

In studying the correlations between smells and metrics we followed a manual approach by analyzing sets of systems with largest and smallest numbers of smell instances. Moreover, we performed a correlation analysis using the Spearman's rank correlation coefficient [29] and Principal Component Analysis. We have detected 17 smells using different open source tools, trying to use only one tool for most of the smells. The subject for the study was a set of 68 systems selected from the Qualitas Corpus [30]. We also considered 20 metrics of different categories: size, complexity, dependency, data abstraction, cohesion and inheritance.

In the literature we found several studies on specific smells, in particular on God and Data Class smells; usually they were analyzing the relations between smells and change and defect proneness (see Section II). We argue that our study on the impact of code smells on system's quality can provide useful hints for software quality teams in their efforts to implement prevention and correction mechanisms and facilitate software maintenance.

The paper is organized as follows: in Section II we briefly describe works related to the topic; in Section III we present the domains of the 68 systems we have analyzed and the metrics we have considered; in Section IV we introduce the smells we have detected and the detectors we have used; in Section V we provide the results on the code smell detection and we outline which are the most common smells and the correlations among smells and metric values; in Section VI we discuss the threats to validity of our study. Finally, in Section VII we conclude and outline some future developments of our work.

## II. RELATED WORKS

Many reports and studies have been published in the literature on code smells and refactoring. Among the papers that analyze the role of the software domains, we found a work by Guo et al. [12], which considers domain-specific characteristics in a software project and use their own tool for code smells detection. In another work, by Marinescu [21], the author shows how the detection accuracy of two well-known code smells (*i.e.*, Data Class and Feature Envy) can be improved by taking into account particularities of enterprise applications, useful for design understanding and quality assessment. An extension of iPlasma (one of the tools we used for code smell detection), called DATES (not publicly released) is introduced to enhance the detection of some code smells in enterprise applications. Moha et al. [25] introduce an approach to automate the generation of code smell detection algorithms using a domain-specific language. In our paper we provide several remarks on the relevance and existence of some *domain and design-dependent smells* in different contexts.

Below we cite only few works directly related to empirical studies on code smells. Most of the cited works are focused only on one or two code smells and principally on their relations to changes or fault-proneness.

Olbrich et al. [26] present a study of God Classes and Brain Classes in evolution of three open source systems with the aim to investigate the extent of God Classes and Brain Classes to be changed more frequently and contain more defects than other classes. Zhang et al. [33] assert that the current empirical basis for using code smells to direct refactoring and to address "trouble" in code is not clear, and they propose a study which aims at empirical investigation of the impact of bad smells on software in terms of their relationship to faults. Vaucher et al. [32] study the "life cycle" of God Classes in two open-source systems: how they arise, how prevalent they are, and whether they are removed as the systems evolve over. They also show how to automatically detect the degree of "godliness" of classes. Khomh et al. [17] investigate if smelly classes are more change-prone than other classes. They demonstrate that in almost all releases of Azureus and Eclipse, smell-crippled classes are more change-prone than others, and the correlation varies depending on the code smell. Li et al. [20] present the results of an empirical study that investigated the relationship between code smells and fault probability in an industrial-strength open source system. Their research showed that some code smells were positively correlated with the class error probability. This finding supports the use of smells as a systematic method to identify and refactor problematic classes in this specific context. Arcelli et al. [3], using well known metrics for code and design quality evaluation, analyze the impact of code smell-driven refactoring on quality metric values, with the aim to prioritize the smells to be removed.

In this paper we consider several smells and their correlations with the values of different metrics. Our study is in line with our previous work [3]; however, here we do not focus on the refactoring of code smells, but rather examine the smells frequency in different domains and their correlation with different metric values.

## III. IDENTIFICATION OF THE ANALYZED SYSTEMS AND SOFTWARE METRICS

The selection of the systems and the metrics for the analysis is important if we want to capture the specific characteristics of software systems in different application domains. To ensure consistency of the analysis, we decided to focus on systems written in Java, due to the prevalence of Java open source systems in different application domains and the availability of metrics computing tools.

**1. Identification of the subject systems** – As the source for the subject systems we chose the Qualitas Corpus, collected and maintained by Tempero et al. [30]. The particular instance we used (20101126r) includes 106 systems written in Java, of different sizes and designations.

In this study we analyze only classes native to the systems *i.e.* excluding external libraries required to compile, run or test the system, as indicated also by Tempero et al. [30]. This is necessary for isolating classes directly related to given domains.

Next, we categorized the systems with respect to their size expressed by LOC (Line Of Code) metric, according to

the LOC ranges in Table I. Most of the systems in the corpus are of Small-Medium, Medium, and Medium-Large size. Based on this observation, we decided to focus on this ranges and selected 68 systems (composed of 36.750 classes) for further analysis out of the initial 106 systems. The set of selected systems is sufficiently homogeneous in terms of size to avoid biasing our analysis as we aggregate the metrics at the system level.

TABLE I. LOC RANGES USED TO CLASSIFY SYSTEMS

LOC Range	System Size
0...4999	Small
5000...14,999	Small-Medium
15,000...39,999	Medium
40,000...99,999	Medium-Large
100,000...499,999	Large
≥ 500,000	Very Large

TABLE II. APPLICATION DOMAINS AND THEIR CORRESPONDING CATEGORIES

Application Domain	Categories by Tempero et al.
Diagram generator/ Data visualization	GUI Design Tool
Software development	Ide, Parsers/Generators/Make, SDK, Testing
Application Software	Word Processor, Web Browser, Accounting, Graphic and Player
Client-Server Software	Database, Application Server, Middleware, CMS

Tempero et al. [30] classified the systems into 12 categories. However, for our analysis the number of systems in each category would be too small, so we assigned the selected 68 systems to 4 coarse-grained domains, which include the Tempero’s categories, as shown in the Table II. The domains are distinct with respect to the characteristics of systems that belong to them, and numerous enough for the analysis. As a result, 11 systems belong to the Diagram generator/Data visualization domain, 25 to the Software Development, 11 to the Application Software, and 21 to the Client-Server Software application domain.

**2. Selection of the metrics** – In order to perform the analysis described in Section V and we selected metrics for object-oriented software quality evaluation ([19, 24, 13]) in the following categories: size, complexity, dependency, data abstraction, cohesion and inheritance. From the metrics suggested by literature, we selected 20 of them, with at least one metric in each category. The selection was affected by availability of a tool for computing the given metric. We also rejected metrics that are included in the detection strategies of code smells (see Section IV). In Table III we report the metrics we computed on the 68 systems, along with their categories, the tools used for computing, granularity levels and the source references.

TABLE III. METRICS COMPUTED ON 68 SYSTEMS

Name (Acronym) [Ref]	Granularity	Tool
<b>Category: Size</b>		
Lines of Code (LOC) [19]	Operation	iPlasma [15]
Number of Methods (NOM) [19]	Class	
Number of Packages (NOP) [18]	System	
Number of Classes (NOC) [19]	System	
Number of equal Lines of Code (EDUPLINES) [19]	Method	
Average Line of Code per Method (ALCM) [11]	System	CodePro Analytix [11]
<b>Category: Complexity</b>		
McCabe’s Cyclomatic Number (CYCLO) [24]	Operation	iPlasma [15]
Weighted Method Count (WMC) [5, 24]	Class	
Average Method Weight (AMW) [24]	Class	
Program Volume (PV) [11]	System	CodePro Analytix [11]
<b>Category: Dependency</b>		
Abstractness (Abstr) [11]	System	CodePro Analytix [11]
Instability (I) [11]	System	
Distance from Main Sequence (DMS) [11]	System	
Changing Classes (CC) [19]	Method	iPlasma [15]
Number of Called Classes (FANOUT) [19]	System	
<b>Category: Data Abstraction</b>		
Access To Foreign Data (ATFD) [19]	Class, Method	iPlasma [15]
Locality of Attribute Accesses (LAA) [19]	Method	
<b>Category: Cohesion</b>		
Tight Class Cohesion (TCC) [19]	Class	iPlasma [15]
Lack of Cohesion Metric (LCOM) [13]	Class	Eclipse Metrics [7]
<b>Category: Inheritance</b>		
Average Depth of Inheritance Hierarchy (ADIH) [11]	System	CodePro Analytix [11]

In order to ensure consistency and comparability of results, metrics at the method and class granularity levels have been aggregated at the system level. In particular, AMW, ATFD, TCC, WMC, CC, CYCLO, LAA, LCOM have been aggregated with respect to the mean value and NOM, LOC, EDUPLINES with respect to the sum.

We selected metrics computation tools with respect to clearly documented metric definitions and their implementation. The values of all the metrics from Table III for all 68 systems are available online at [8].

## IV. CODE SMELL DETECTION

The concept of code smells was introduced by Fowler [9], who also identified and named initial 20 smells; others have been identified in the literature later [21] and new ones can be defined. Code smells, although initially thought to be captured primarily by human judgment, are now subject to

automated detection as well. Numerous tools have been developed that exploit different detection techniques. In most cases, they are based on the computation of a particular set of combined metrics [19], either standard object-oriented metrics or defined *ad hoc*.

In our study we consider a number of code smells presented in Table IV, detected on the 68 systems we have described in the previous section. Table IV provides the name of the smell, its granularity (class, method or operation), a source reference to the smell definition and the detector tool we used. Noticeably, Refused Parent Bequest code smell is reported twice, as iPlasma detector includes two versions of this smell.

TABLE IV. DETECTED CODE SMELLS

Code Smell [Ref]	Granularity
<b>Detection Tool: iPlasma [15]</b>	
God Class [9]	Class
Refused Parent Bequest [9]	Class
Refused Parent Bequest 2 [15]	Class
Intensive Coupling [19]	Method
Extensive Coupling [19]	Method
Shotgun Surgery [9]	Method
Feature Envy [9]	Method
Data Class [9, 27]	Class
Brain Method [19]	Method
Brain Class [19]	Class
Tradition Breaker [19]	Class
Schizophrenic Class [31]	Class
<b>Detection Tool: NiCad [28]</b>	
Duplicate Code [9]	Operation
<b>Detection Tool: Excel macros</b>	
Speculative Generality [9]	Class
Long Method [9]	Method
Long Parameter List [9]	Method

We selected iPlasma as the primary detector because it clearly defines (at least for 11 smells) its detection techniques and the adopted metrics thresholds [19]. In addition to that, we also used NiCad [28] for detecting Duplicate Code (in particular clones of Type I and Type III), as it allows for switching between function-/method- and block-level clones. Moreover, it allows for customization of the clone detection, which is not very common for other detectors [34].

For three code smells: Speculative Generality, Long Method and Long Parameter List, we have defined our own detection rules (reported as Excel macros Table IV). We adopted slightly changed definitions of the Long Method and Long Parameter List smells than those reported in the literature. In the latter case, they are usually based on fixed thresholds or subjective evaluation experiments [20]. Our rules adopt relative thresholds, so they depend on the context of the entire system and not just on the individual classes.

For Speculative Generality [9] we implemented a detection rule based on the definition given by Moha [25].

## V. SMELL FREQUENCY AND METRIC CORRELATION IN SYSTEMS OF DIFFERENT DOMAINS

### A. Smells distribution in different software domains

We detected code smells presented Section IV for all the 68 systems identified in Section III. Chart V shows the number of smell instances and their distribution in each application domain. To preserve consistency of results at different granularity levels, the total number of code smell instances in each application domain has been divided by:

- the total number of classes of the systems in the domain, for the code smells at the class granularity level;
- the total number of methods of the systems in the domain, for code smells at the method granularity level.

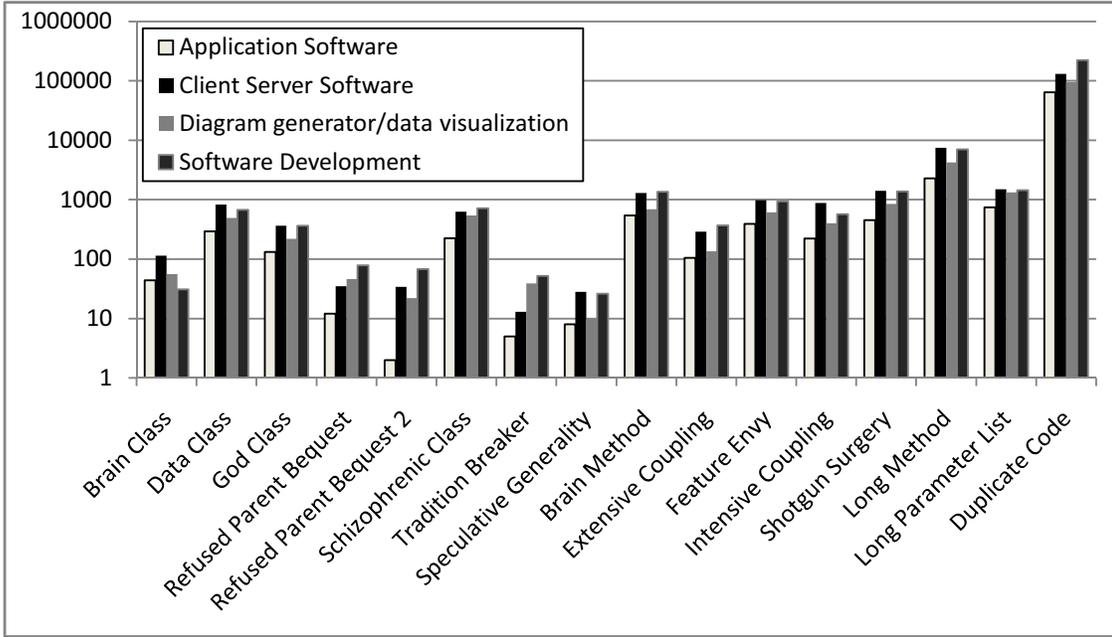
If a code entity contains several smells (e.g. a Long Method could have a Long Parameter List), they are counted separately.

When considering Duplicate Code smell, for each system the total number of duplicated lines has been divided by the total number of lines. The computation does not consider the possible relations existing between smells, in particular the relation of mutual exclusion.

Based on the results presented in Chart V, it is evident that some smells are more prevalent than others, even if (at the application domain level), significant differences are not observable. The most common smell is Duplicate Code (23.07% in the Software Development domain), followed by Data Class, God Class, Schizophrenic Class and Long Method. The most smell-crippled domain is the Diagram generator/Data visualization domain (33.87%), followed by Application Software (33.54%), Client-Server Software (31.71%) and Software Development (29.32%), although differences are not significant.

**Observation.** The distribution of code smells is approximately the same in all the considered domains. It comes from the fact that code smell detection tools are ignoring features related to the domain or design of the system. In a previous paper [3] we analyzed the impact on the refactoring of 2 smells (namely God Class and Data Class) on the values of selected metrics. We manually verified all code smells instances pointed to by automated detectors and found a high number of false positive instances. What is more important, we also found many cases of smells that can be considered *domain* or *design-dependent smells*: the numbers of false positives were different in different domains. God Classes and Data Classes were manually confirmed in 23.38% and 13.67% automatically identified cases, respectively. It means that in some domains specific smells do not point to an actual flaw, so their presence is misleading. For example, in the Client-Server Software Domain it is common to use Java beans for

CHART V. CODE SMELL DISTRIBUTION IN DIFFERENT DOMAINS



transporting data between remote layers (known as Data Transfer Objects). They are falsely detected as Data Class smells and represent a common example a domain-dependent smell. From the results we present in this section, we can conclude that other smells also exhibit similar behavior.

The same applies to design patterns, which can introduce code smells as well [16]. For example, a Composite design pattern may involve a Refused Parent Bequest smell or a Visitor pattern includes the Feature Envy and Data Class smells.

All these observations suggest that accuracy of code smells detection rules used by the tools can be improved by exploiting knowledge about both the domain and the design of a system.

### B. Manual analysis between smell frequency and metrics

In this subsection we investigate the correlation between the number of smell instances in the systems and the values of object-oriented metrics. Specifically, we want to know if low/high values of some metrics can serve as hints suggesting the presence of smells. For that purpose we have selected 10 systems (out of 68) with most outstanding number of smell instances: 5 with the smallest and 5 with the largest number of smell instances. In this analysis we reduced the number of subject systems as we found exactly 5 systems with significantly. We have performed our analysis by considering the percentage of the smells for each type of granularity level (smells for classes and for methods).

TABLE V. MANUAL ANALYSIS OF METRICS ON 68 SYSTEMS

Metric	Mean-Least Smell	Mean-Most Smell	Variation	Outlier
<b>Smell granularity: Class</b>				
WMC	7.98	21.89	+ 174.31%	0
AMW	1.37	2.22	+ 61.04%	0
ATFD	1.02	3.58	+ 250.98%	0
CYCLO	1.44	3.12	+ 116.18%	1
LAA	0.96	0.89	- 6.92%	2
ALCM	8.21	13.73	+ 67.20%	2
FANOUT	3.44	8.45	+ 145.58%	2
<b>Smell granularity: Method</b>				
CYCLO	1.64	3.47	+ 111.59%	0
WMC	13.65	27.01	+ 97.88%	0
Abstr	16.32	7.96	- 51.23%	0
FANOUT	3.46	13.41	+ 287.57%	0
AMW	1.65	2.47	+ 49.55%	1
LAA	0.96	0.87	- 9.09%	1
ALCM	8.01	14.94	+ 86.49%	2
CC	1.30	1.66	+ 27.52%	2
ATFD	2.00	4.99	+ 149.37%	2

Table VI reports the results: the name of the metric, its granularity level, the mean value for five systems with smallest and largest number of smell instances, the variation of the mean value for the systems with most

smells with respect to those with fewest smells and the number of outliers for each metric.

For all but two metrics (namely Abstr and LAA) presented in Table VI, an increase on the value of the metric implies deterioration on the quality of the system. For the Abstr and LAA metrics we observe decrease that implies respectively deterioration on the dependency and data abstraction of the quality of the system. The values of each metric in the systems with largest smell instances represent deterioration with respect to the systems with the smallest number. The strongest hints of deterioration come from the values of FANOUT (+287.57%), ATFD (+250.98%) and WMC (174.31%) metrics. Hence, we can observe significant deterioration for dependency, data abstraction and complexity, respectively. For some metrics we found outliers that correspond to the metric values (at the method/class granularity level) with different trends between the two sets of systems with the largest/smallest number of smells. The metrics without outliers demonstrate a trend that is more likely to indicate the improvement/deterioration of the system quality.

In the same way we made the analysis separately for each application domain with the aim to understand which are the metrics affected by the largest variations in the given domain. The results of the analysis indicate that the Application Software is the domain in which the metrics of complexity and dependence have the highest variations compared to the other domains, and in particular we observed a high variation of following metrics: WMC (+105.10%), CYCLO (+115.99%), AMW (+63.44%), Abstr (-80.49%), DMS (-70.58%), FANOUT (+214.57%). Furthermore, we observed that the Client-Server Software domain has a high variation of the cohesion and data abstraction metrics: ATFD (+187.15%), LAA (-4.04%), and LCOM (+55.98%). This feature outlines with good probability that for the systems in the Client-Server Software domain the presence of most smells has the greatest negative impact on cohesion and data abstraction.

### C. Analysis of the correlation between smell frequency and metric values

We used the collected data about metrics, presented in Section III, and the data about smells, presented in Section IV (Tables V and VI), to analyze the correlation between code smells frequency and selected metrics. First, we observed for each system that basically all metrics demonstrate a monotonic trend with respect to the smell frequencies. Therefore, we chose a correlation test like Spearman's rank correlation coefficient [27], as it allows to assess how the relationship between two variables can be described using a monotonic function. There are two methods to calculate Spearman's rank-order correlation, depending on whether our data does or does not have tied ranks (the one used in Pearson correlation). We used these methods in order to distinguish cases with and without duplicates.

We decided to perform a *two tails* analysis to obtain also the direction of the correlation. Table VII presents

*Spearman's rho coefficients* for all smell-metric pairs (except for those including EDUPLINES metric, which had rho equals to 0). In this table, rho greater than 0.7 or less than -0.7 with a significance level (p-value) less or equal to 5% are printed in bold and additionally their p-value is given (full data is available online at [8]). The results indicate strong correlation between 10 smell-metric pairs, including Brain Class, Feature Envy, God Class, Brain Method and Shotgun Surgery code smells.

The highest correlation is observed between Brain Method and CYCLO (rho: 0.836, p-value: 7E-17), then for Feature Envy-LAA (rho: -0.817, p-value: 2E-15), Shotgun Surgery-CC (rho: 0.806, p-value: 1E-14), Brain Method-ALCM (rho: 0.8, p-value: 3E-14), Brain Class-WMC (rho: 0.788, p-value: 2E-13), Brain Method-AMW (rho: 0.773, p-value: 1E-12), God Class-ATFD (rho: 0.758, p-value: 7E-12), God Class-WMC (rho: 0.742, p-value: 4E-11), Brain Class-CYCLO (rho: 0.722, p-value: 4E-10) and Brain Class-AMW (rho: 0.711, p-value: 1E-09). Noticeably, all but one pairs demonstrate positive correlation (with the exception of the Feature Envy-LAA pair). In all these cases the p-value is negligibly small, which shows high significance of the obtained correlation. Another 10 smell-metric pairs have rho coefficient around 0.6.

However, some of the strongly correlated pairs can be questioned, as they include code smells that are detected with direct use of the same metrics they are correlated with (see Table VIII, column "Metrics involved"). For the Brain Class-WMC, Brain Class-CYCLO, God Class-WMC, God Class-ATFD, Brain Method-CYCLO, Feature Envy-LAA, Feature Envy-ATFD and Shotgun Surgery-CC pairs, it is actually the case. For other pairs: Brain Class-AMW, God Class-AMW, Brain Method-WMC, Brain Method-AMW and Intensive Coupling-FANOUT, the observed high correlation is more interesting because the metrics are not directly used in the smell detection rule, even if they belong to the same category as the ones used in the detection rule, or directly related to that. For example, for the God Class-AMW pair, the metric is not used in the detection, but it is closely related to WMC. This one, in turn, is used in the detection strategy of God Class. On the other hand, for the God Class-FANOUT and Feature Envy-FANOUT pairs we observe only medium rho correlation (with high p-value), but the FANOUT is not used in the God Class and Feature Envy detection, even if it is indirectly related to ATFD metric, which is explicitly used in the detection rules of those smells.

But in some cases there exists no such relation between a code smell and a metric. The following pairs are the most interesting (with respect to the rho value): Brain Method-ALCM (rho: 0.8, p-value: 3E-14), Extensive Coupling-CC (rho: 0.697, p-value: 4E-09), Shotgun Surgery-PV (rho: 0.653, p-value: 2E-07), Shotgun Surgery-LOC (rho: 0.642, p-value: 4E-07), Brain Class-ALCM (rho: 0.613, p-value: 3E-06), Shotgun Surgery-FANOUT (rho: 0.612, p-value: 2.9E-06). In these cases the metrics and code smells do not share common roots or belong to a different category. For

some smells, their correlation with corresponding metrics or similar metrics in the same category is obvious (e.g. a Data Class smell should be correlated with the Data

Abstraction metrics, or God Class smell with the complexity metrics), but this is not true in all cases.

TABLE VI. SPEARMAN’S RANK CORRELATION BETWEEN CODE SMELL FREQUENCY AND METRICS

Code Smell	Metrics																		
	LOC	NOM	NOP	NOC	ALCM	CYCLO	WMC	AMW	Abstr	I	DMS	CC	FANOUT	ATFD	LAA	TCC	LCOM	PV	ADIH
Brain Class	-0.065	-0.169	-0.264	-0.343	0.613	<b>0.722 4E-10</b>	<b>0.788 2E-13</b>	<b>0.711 1E-09</b>	-0.128	-0.123	-0.216	0.132	0.443	0.481	-0.155	0.132	0.051	-0.04	-0.018
Data Class	-0.247	-0.293	-0.124	-0.376	0.256	0.239	0.257	0.35	-0.197	-0.092	-0.227	-0.022	0.218	0.42	-0.229	0.109	0.383	-0.244	-0.229
God Class	0.031	-0.043	-0.206	-0.243	0.563	0.521	<b>0.742 4E-11</b>	0.641	-0.23	-0.106	-0.214	0.433	0.601	<b>0.758 7E-12</b>	-0.495	0.066	0.215	0.058	-0.005
Refused Parent Bequest	0.443	0.482	0.352	0.45	-0.058	-0.004	0.101	0.034	-0.083	0.146	-0.028	0.295	0.305	-0.004	-0.104	-0.182	-0.011	0.495	0.258
Refused Parent Bequest 2	0.37	0.392	0.222	0.327	-0.014	0.2	0.218	0.212	-0.029	0.065	0.026	0.197	0.24	0.028	-0.001	-0.115	0.075	0.431	0.199
Schizophrenic Class	-0.112	-0.075	-0.105	-0.024	0.004	0.099	-0.135	-0.05	-0.132	0.186	0.036	-0.196	-0.098	-0.113	-0.131	0.231	-0.206	-0.102	0.115
Tradition Breaker	0.234	0.241	0.033	0.133	0.066	0.187	0.35	0.054	-0.194	0.121	-0.08	0.126	0.268	0.035	-0.135	-0.076	-0.059	0.307	0.214
Specularity Generality	-0.122	-0.024	-0.05	-0.044	-0.274	-0.277	-0.241	-0.217	0.459	0.165	0.537	-0.092	-0.202	-0.256	0.355	-0.12	-0.139	-0.099	-0.156
Brain Method	-0.083	-0.334	-0.234	-0.389	<b>0.8 3E-14</b>	<b>0.836 7E-17</b>	0.681	<b>0.773 1E-12</b>	-0.28	-0.091	-0.384	0.102	0.517	0.51	-0.315	0.205	0.224	-0.071	-0.015
Extensive Coupling	0.422	0.33	0.338	0.292	0.2	0.151	0.264	0.292	0.072	-0.188	-0.046	<sup>0.697</sup> <sub>4E-09</sub>	0.564	0.31	-0.264	-0.102	0.277	0.411	-0.112
Feature Envy	0.127	0.005	0.06	-0.029	0.389	0.382	0.39	0.395	-0.463	0.074	-0.4	0.375	0.63	0.629	<b>-0.817 2E-15</b>	0.165	0.229	0.157	-0.016
Intensive Coupling	0.158	0.01	0.162	-0.03	0.37	0.332	0.373	0.435	-0.218	0.092	-0.195	0.405	0.681	0.552	-0.458	0.049	0.45	0.166	-0.26
Shotgun Surgery	0.642	0.549	0.492	0.535	0.078	0.038	0.08	0.154	-0.101	-0.035	-0.088	<b>0.806 1E-14</b>	0.612	0.292	-0.486	0.102	0.172	0.653	-0.011
Long Method	0.12	0.079	0.226	0.168	-0.135	-0.191	-0.26	-0.149	-0.061	0.079	-0.042	-0.155	-0.158	-0.241	0.093	0.041	0.128	0.078	-0.077
Long Parameter List	0.175	0.045	0.062	0.014	0.205	0.295	0.271	0.182	-0.105	-0.059	-0.114	-0.015	0.23	0.019	-0.042	0.075	0.117	0.19	0.139
Duplicated Code	0.144	0.024	-0.062	-0.053	0.200	0.212	0.274	0.106	0.193	-0.202	0.147	0.123	0.104	0.120	0.079	0.006	-0.064	0.103	-0.232

In order to assess the actual impact of particular metrics on a specific code smell, we employed Principal Component Analysis (PCA). For that purpose we defined separate data sets for every code smell, composed of the smell and all metrics relevant to the smell’s granularity level. PCA transforms a multidimensional space of objects into a corresponding space of eigenvectors, ordering them according to the amount of explained variance in data. Thus, it can be used for both reducing dimensionality of the original space and assessing the importance of the original attributes (metrics, in our case) for the resulting eigenvectors.

In order to find the most influential metrics we considered only Principal Components that explain no less than 0.05 of the variance and include no less than 0.3 of the code smell value. The code smells and metrics conformant to these conditions are presented in Table VIII (column “Metrics related”). Due to lack of values for several metrics from Table III at the method and class level, the “Metrics related” column includes only those metrics that were available and could be analyzed.

As we can see, metrics extracted by the PCA partially overlap with the ones strongly correlated with code smells. Based on the presented results we can assert e.g. that the presence of the smell Shotgun Surgery is a valid indicator

of the increasing complexity and size of the systems and the presence of Brain Class and Brain Method smells is an indicator of considerable size of the system. These three

TABLE VII. METRICS RELATED TO CODE SMELLS

Code smell	Metrics included in detection strategy	Metrics related
Brain Class	LOC, WMC, TCC	AMW
Data Class	NOC	NOC, ATFD
God Class	WMC, TCC, ATFD	NOM, WMC
Refused Parent Bequest	NProtM, BUR, BOvR	AMW
Refused Parent Bequest 2	AMW, WMC, NOM	AMW
Schizophrenic Class	(unknown)	AMW
Tradition Breaker	NAS, AMW, WMC, NOM	AMW
Speculative Generality	(N/A, structure-based detection)	
Brain Method	LOC, CYCLO, MAXNEST, NOAV	LOC, LAA
Extensive Coupling	(unknown)	LAA, NOP, LOC
Feature Envy	ATFD, LAA, FDP	LOC, LAA, NOP
Intensive Coupling	MAXNEST, CINT, CDISP	LAA, LOC
Shotgun Surgery	CM, CYCLO	CYCLO
Long Method	NOP	LAA, LOC
Long Parameter List	NOP	NOP

smells capture different features of object-orientation with respect to their aims. Detecting numerous instances of these smells points to design flaws strictly connected to the complexity and size of the system.

To formally support the hypothesis that an element affects another one, we have to perform other analyses. One of the possible approaches could be refactoring the involved smell and then verifying if the operation effectively improves (or not) the metric values of size and complexity. In [3] we described of 12 systems that had been refactored, removing God Class, Data Class and Duplicate Code smells. We noticed that removing the God Class smell in many cases yields an improvement on WMC and ATFD metric. Therefore, we can conclude that some of the highly correlated smell-metric pairs could indicate problems that can be solved or diminished with highest priority through refactoring.

The results reveal no significant correlation of Duplicated Code with any of the metrics (the highest rho-value - 0.274 - was observed for WMC). This shows that clones do not directly affect the structural properties of the system, and possibly impact metrics closely related with maintenance effort.

We performed all the analyses described in this section also considering separately the subsets of 68 systems belonging to each of the four different application domains of Table I. Some of the smell-metric pairs reveal domain specific features related to quality evaluation. Correlation matrix for each domain is available online at [8].

The Diagram generator/Data visualization domain is the only one in which the presence of the Long Parameter List smell is correlated with WMC (rho: 0.764, p-value: 0.61), ALCM (rho: 0.691, p-value: 1.85), and CYCLO (rho: 0.627, p-value: 3.89) metrics. According to the definition of the detection rule for this smell (described in Section IV), the reason for this observation is probably due to the fact that numerous methods from this domain accept many primitive parameters as input. These methods probably provide graphics functions that are very adaptable and versatile, according to the needs of the developer. Such methods are usually complex and have a fairly high Average Number of Lines of Code metric.

Another interesting property of this domain is high correlation between Tradition Breaker code smell and ATFD (rho: 0.779, p-value: 0.47), FANOUT (rho: 0.748, p-value: 0.81) and LAA (rho: -0.674, p-value: 2.29) metrics. Starting from the definition of Tradition Breaker and considering the heavy use of graphics libraries in the considered domain, we can expect that the presence of this smell is a consequence of a strong dependency between the classes (in particular between system classes and libraries) and high data abstraction of the system.

Noticeably, the strength and direction of correlation of code smells and metrics values varies in different domains. We can observe e.g. a significant positive correlation between PV and Long Method for the Application Software domain (rho: 0.627, p-value: 3.896), which in turn becomes slightly negative in the Software

Development domain (rho: -0.211, p-value: 31.133). It turns out that methods in Application Software domain become more complex along the growth of entire system's complexity, whereas for the other domain they do not or even exhibit the opposite relation. It is difficult to provide a definite explanation for that observation; probably it is due to more intense maintenance activities conducted on the tools in Software Development domain than for the Application Software systems.

For the Client-Server domain, CC metric is strongly correlated with Extensive Coupling (rho: 0.761, p-value: 0.01), while for the Software Development domain this relation is weaker (rho: 0.445, p-value: 2.58). The ATFD metric is correlated with the Brain Class code smell in the Client-Server and Application Software domains (rho: 0.640 and 0.706, p-value: 0.178 and 1.51, respectively), whereas it is substantially weaker for the Software Development domain (rho: 0.343, p-value: 9.32). Again, it is probably the result of more advanced maintenance applied to systems in Software Development domain.

These facts document the observation that code smells have various impact on metrics (as further, on software quality) in different domains. We can draw a conclusion that domain of an analyzed system is an important factor that should be included in the detection strategies for various code smells.

## VI. THREATS TO VALIDITY

This section summarizes several factors that pose threats to validity for the results of the study.

Internal validity of a study deals with demonstrating a casual relation between analyzed variables and identifying unknown factors that may affect the relation. In our case, several elements account for that. Correctness of results provided by code smell detectors is a key issue for the validity of the paper findings. Identification of code smells depends on the adopted definition and thresholds of the metrics involved in the detection procedure. Thus, different tools could detect different code smell instances, which significantly influences the results. For example, replacing NiCad as a clone detector with another tool can identify completely different clone suspects due to various definitions of clones and near-clones. We did not perform cross-validation of the smell detection tools, as it was beyond the scope of the study. However, this is an important threat to validity. Similarly, we computed every metrics value only with single tool. As metrics are formally defined, there should be no difference in results obtained with different measurement software, but in many cases interpretation of the definitions could vary.

Another important issue is the intrinsic dependency between metrics and code smells. Code smell detectors use metrics as the primary source of data, so using metrics also as reference quality measurements poses risk to validity of the results. We thrived to reduce the risk, but to some extent it is still present.

External validity is preserved if the results are true regardless of the variations in experimental settings and

procedures. The main doubt in our study is if the sample of projects selected from Qualitas Corpus is representative for software in general (open source, commercial, or both). We analyzed systems of different sizes in four different domains, avoiding intentional narrowing the scope of the systems. But it would be interesting to perform our analysis also on larger sample of systems and compare the results.

## VII. CONCLUDING REMARKS AND FUTURE WORKS

In this paper we described an empirical study on code smells detection and an analysis on the frequency of smells in systems of different domains. We have considered values of different metrics to analyze possible correlations among smells and metrics. We argue that our study could be useful to improve software quality, maintainability and evolution, taking into account the frequency and the relevance of smells, and their correlations to software quality metrics and to the domains and the systems they belong.

We have considered two questions:

RQ1) Which are the most frequent smells detected in different systems of different application domains? Is there a domain that usually contains more smells of a particular kind?

RQ2) What can we observe according to the metric values of systems with more smells? Is the code smell distribution correlated to the values of metrics? Do significant correlations exist among smells and metrics? Do low/high metric values always indicate code smells?

Addressing RQ1, we identified Duplicate Code, Data Class, God Class, Schizophrenic Class and Long Method as the most common smells in general, but at the application domain level, significant differences among code smells are not observable. In some cases we observed that the code smell detector could return false positive results; hence, some smells appear domain- or design-dependent.

Looking for an answer to RQ2 in Section V-B, we identified a set of metrics whose values, for the systems mostly crippled with code smells, deteriorate in comparison to systems with a small number of smells; hence, we identified interesting correlations between code smell frequency and metric values. In particular, we observed a high deterioration for dependency, data abstraction and complexity. We observed that Application Software is the domain in which the metrics of complexity and dependence have the highest variations compared to the other domains. In the Client-Server domain the presence of most smells has the strongest negative relation with cohesion and data abstraction, among all the application domains. In this case, we observed the importance of detecting and recognizing the existence of possible domain-dependent or design-dependent smells to be removed.

Analyzing RQ2 in Section V-C, we examined correlation between code smells and quality metrics used as a reference. For that purpose we used two methods:

Spearman rank correlation and Principal Component Analysis. The analysis revealed significant correlation between several code smells and selected metrics. Some of the discovered relations, however, result from metrics being directly included in the code smell detection strategy. Analyzing the relations between metrics and code smells we were able to identify those cases and focus on the non-trivial smell-metric pairs. Similar analysis was performed in each domain separately. It revealed that strength of correlation varies with respect to the domain.

We suggest that the discovered relations can be useful to implement prevention mechanisms that facilitate software maintenance, by taking into account the frequency of the smells, and their correlations to software quality metrics and to the domains and the systems they belong.

For the future developments, we plan to extend our investigation on the domain-dependent code smells. Features of the domain or the system where we detect the smells, as we observed before for the LM and LPL smells, are important to capture the real nature of the smell, if it actually points to a flaw or it is an acceptable solution in that domain. This study requires a deep and careful manual analysis of the detected code smells. Manual analysis could also help investigating some of the code smell instances to verify the detection results and improve the reliability of our study. Another future investigation we plan is towards the refactoring of the smells strongly correlated with metrics measuring deterioration of the software quality. Removal of such code smells should improve the overall system quality and could be used to verify the relations discovered with the Spearman analysis and PCA. This should allow for more accurate identification of the actually harmful smells and prioritizing those to be removed. Moreover, identifying and refining the most useful set of metrics and also considering the design evolution metrics as used by Kpodjedo for defect prediction [18], is another direction of future research. Finally, we would like to better explore the correlations existing among smells. A code smell can contain other smell, and the presence on one smell could imply the existence or not existence of other smells. We started the investigation on these topics exploiting data-mining techniques in our previous paper [1].

## REFERENCES

- [1] F. Arcelli Fontana, M. Zanoni, B. Walter, and P. Martenka, "Code smells, micro patterns and their relations". ERCIM News, 88:33, January 2012. Special theme: Evolving Software. URL: <http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf>
- [2] F. Arcelli Fontana, P. Braione and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment". Journal of Object Technology (11:2), 2012, pp. 1-38.
- [3] F. Arcelli Fontana, V. Ferme and S. Spinelli, "Investigating the Impact of Code Smells Debt on Quality Code Evaluation". Proc. of the 4th Workshop the Third Workshop on Managing Technical Debt, in conjunction with ICSE 2012, IEEE, Zurich, June 5 2012.
- [4] H. Barkmann, R. Lincke and W. Löwe, "Quantitative Evaluation of Software Quality Metrics in Open-Source Projects".

- International Conference on Advanced Information Networking and Applications Workshops, WAINA '09, 2009.
- [5] S.R. Chidamber and C.F. Kemerer, "A metric suite for object-oriented design". IEEE Transactions of Software Engineering, vol. 20, June 1994, pp. 476-493, doi:10.1109/32.295895.
- [6] M. Dagpinar and J.H. Jahnke, "Predicting Maintainability with Object-oriented Metrics. An empirical Comparison". IEEE Proc. of the WCRE 2003 Conference, Canada 2003.
- [7] Eclipse Metrics plugin: <http://eclipse-metrics.sourceforge.net>
- [8] ESSERE Lab: Evolution of Software SystEms and Reverse Engineering, <http://essere.disco.unimib.it/reverse/MetricSmellCorrelations.html>
- [9] M. Fowler et al, "Refactoring: Improving the Design of Existing Code". Addison-Wesley. Inc., Boston, MA, USA, 1999.
- [10] M. Fowler, "Patterns of Enterprise Application Architecture". Addison Wesley, 2002.
- [11] Google CodePro Analytix: <http://code.google.com/intl/it-IT/javadevtools/codepro/doc/index.html>
- [12] Y. Guo, C. Seaman, N. Zazworka and F. Shull, "Domain-specific tailoring of code smells: an empirical study". Proc of 32nd International Conference on Software Engineering (ICSE 10), ACM/IEEE, May 2010, pp. 167-170, doi:10.1145/1810295.1810321
- [13] B. Henderson-Sellers, "Object-Oriented Metrics: Measures of Complexity". Prentice Hall, 1996.
- [14] InFusion: [www.intooitus.com/inFusion.html](http://www.intooitus.com/inFusion.html)
- [15] iPlasma: <http://loose.upt.ro/iplasma/index.html>
- [16] S. Jancke, "Smell Detection in Context". Master Thesis, University of Bonn, 2010.
- [17] F. Khomh, M. Di Penta and Y. Gaël Guéhéneuc, "An exploratory study of the impact of code smells on software change proneness". Proc of 16th Working Conference on Reverse Engineering (WCRE'09), France, October 2009, pp 75-84, doi:10.1109/WCRE.2009.28.
- [18] S.Kpodjedo, F.Ricca, P. Galinier, Y.Gaël Guéhéneuc and G. Antoniol, "Design evolution metrics for defect prediction in object oriented system". Empirical Software Engineering (2011) 16, pp. 141-175, doi 10.1007/s10664-010-9151-7.
- [19] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Springer-Verlag, 2006.
- [20] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". Journal of Systems and Software, Vol. 80(7), 2007, pp. 1120-1128. doi:10.1016/j.jss.2006.10.018.
- [21] M. Mäntylä, and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study". Journal of Empirical Software Engineering, Vol. 11, No. 3, 2006, pp. 395-431.
- [22] C. Marinescu: Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications. Proc. of ICPC 2006.
- [23] R. Marinescu and D. Rău, "Quantifying the Quality of Object Oriented Design: The Factor-Strategy Model". Proceedings of the 11th Working Conference on Reverse Engineering, WCRE '04, 2004.
- [24] T.J. McCabe, "A measure of complexity". IEEE Transactions on Software Engineering, Vol. 2(4), December 1976, pp. 308-320.
- [25] N. Moha, Y. Gaël Guéhéneuc, A. Françoise Le Meur, L. Duchien and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells". Formal Aspects of Computing Springer, Vol. 22, N. 3-4, doi: 10.1007/s00165-009-0115-x, 2009.
- [26] S. M. Olbrich, D. S. Cruzes and D. I. K. Sjöberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems". Proceedings of International Conference on Software Maintenance (ICSM 10), IEEE, Sept. 2010, pp. 1-10, doi:10.1109/ICSM.2010.5609564
- [27] A. Riel, "Object-Oriented Design Heuristics". Addison-Wesley, Boston MA, 1996.
- [28] C. Roy and J. Cordy, "NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization". Proc. of the 16th IEEE International Conference on Program Comprehension (ICPC 2011), 2011, pp. 172-181.
- [29] C. Spearman, "The proof and measurement of association between two things". The American Journal of Psychology, 100(3/4), 1987.
- [30] E. Tempero et al, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". Proc. of 17th Asia Pacific Software Engineering Conference (APSEC 10), IEEE., 2010, pp. 336-345, doi:10.1109/APSEC.2010.46
- [31] A. Trifu, "Towards Automated Restructuring of Object Oriented Systems". IEEE Proceedings of Int. Conference of Software Maintenance and Reengineering, 2008.
- [32] S. Vaucher, F. Khomh, N. Moha, and Y. Gaël Guéhéneuc, "Tracking design smells: Lessons from a study of god classes". Proc. of 16th Working Conference on Reverse Engineering (WCRE '09), Lille, France, October 2009, IEEE Computer Society, pp. 145-154. doi:10.1109/WCRE.2009.23.
- [33] M. Zhang, T. Hall, N. Baddoo, and P. Wernick, "Do bad smells indicate "trouble" in code?". In Proc. of the 2008 workshop on Defects in large software systems, DEFECTS'08, Seattle, 2008. ACM, pp. 43-44, doi:10.1145/1390817.1390831.
- [34] C.K. Roy, J.R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". Science of Computer Programming, Vol. 74/1 (2009), pp. 470-495.

# Predicting Bugs Using Antipatterns

Seyyed Ehsan Salamati Taba<sup>1</sup>, Foutse Khomh<sup>2</sup>, Ying Zou<sup>3</sup>, Ahmed E. Hassan<sup>1</sup>, and Meiyappan Nagappan<sup>1</sup>

<sup>1</sup> *School of Computing, Queen's University, Canada*

<sup>2</sup> *SWAT, École Polytechnique de Montréal, Québec, Canada*

<sup>3</sup> *Department of Electrical and Computer Engineering, Queen's University, Canada*

*taba@cs.queensu.ca, foutse.khomh@polymtl.ca, ying.zou@queensu.ca, ahmed@cs.queensu.ca, mei@cs.queensu.ca*

**Abstract**—Bug prediction models are often used to help allocate software quality assurance efforts. Software metrics (*e.g.*, process metrics and product metrics) are at the heart of bug prediction models. However, some of these metrics like churn are not actionable; on the contrary, antipatterns which refer to specific design and implementation styles can tell the developers whether a design choice is “poor” or not. Poor designs can be fixed by refactoring. Therefore in this paper, we explore the use of antipatterns for bug prediction, and strive to improve the accuracy of bug prediction models by proposing various metrics based on antipatterns. An additional feature to our proposed metrics is that they take into account the history of antipatterns in files from their inception into the system. Through a case study on multiple versions of Eclipse and ArgoUML, we observe that (i) files participating in antipatterns have higher bug density than other files; (ii) our proposed antipattern based metrics can provide additional explanatory power over traditional metrics, and (iii) improve the *F-measure* of cross-system bug prediction models by 12.5% in average. Managers and quality assurance personnel can use our proposed metrics to better improve their bug prediction models and better focus testing activities and the allocation of support resources.

**Keywords**—bug prediction, antipattern, software quality

## I. INTRODUCTION

Software systems are pervasive in our society and play a vital role in our daily lives. We depend on software systems for our transportation, communication, finance, and even for our health. Therefore, correct functioning of software systems is essential. However, identifying and fixing errors in software systems is very costly. It is estimated that 80% of the total cost of a software system is spent on fixing bugs [1]. To reduce this cost, many bug prediction models [2], [3], [4] have been proposed by the research community to identify areas in software systems where bugs are likely to occur. The vast majority of these bug prediction models are built using product (*e.g.*, code complexity [5]) and process (*e.g.*, code churn [3]) metrics, most of which are not actionable. For example, Nagappan and Ball [3] have used code churns to predict bugs in software systems. Yet, it is unclear how developers should act on the churn values of a class to reduce the risk of future bugs occurring.

Different from metrics, antipatterns [6] which identify “poor” solutions to recurring design problems can tell developers whether the design of a class is “poor” or not, and how to improve it using refactorings [7]. Antipatterns are usually introduced in software systems by developers lack of knowledge or experience to solve a particular problem.

Although antipatterns do not usually prevent a program from functioning, they indicate weaknesses in the design that may increase the risk for bugs in the future. In other words, antipatterns indicate a deeper problem in a software system. Previous work by Khomh et al. [8] have found that classes with antipatterns are more prone to bugs than other classes. Antipatterns can be removed from systems using refactoring. If we can predict bugs using antipatterns information, development teams will be able to use refactorings to reduce the risk for bugs in systems. In this paper, we explore the possibility of predicting bugs using antipatterns and strive to improve the accuracy of state-of-the-art bug prediction models by proposing various metrics based on antipatterns. We use statistical modeling to establish and inspect dependencies between our proposed metrics and bugs counts. Using antipatterns and bug information from multiple versions of two open source software systems of Eclipse<sup>1</sup> and ArgoUML<sup>2</sup>, we address the following three research questions:

*RQ1) Do antipatterns affect the density of bugs in files?*

We find that files with antipatterns tend to have higher bug density than the others.

*RQ2) Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?*

We find that our proposed antipattern based metrics (ANA, ACM, and ARL) can provide additional explanatory power over the traditional metrics LOC, PRE and Churn. Among these metrics, ARL shows significant improvement in terms of AIC and  $D^2$ .

*RQ3) Can we improve traditional bug prediction models with antipatterns information?*

We find that ARL can also improve bug prediction models across systems. It has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of *F-measure*.

The remainder of this paper is organized as follows. First, we summarize the related literature on antipatterns and bug prediction models in Section II. Next, we describe the experimental setup of our study in Section III and report our findings in Section IV. In Section V, we discuss threats to

<sup>1</sup><http://www.eclipse.org/>

<sup>2</sup><http://argouml.tigris.org/>

the validity of our work. Section VI concludes our work and outlines avenues for future works.

## II. RELATED WORK

In this section, we discuss the related literature on antipatterns and bug prediction models.

### A. Antipatterns and Code Smells

The first book on antipatterns in object-oriented development was written in 1995 by Webster [9]. Fowler et al. [7] defined 22 code smells that are bad structures in source code. They mentioned that these smells indicate implementation issues that can be solved using refactoring. Moreover, They claim that code smells have detrimental effects on software. However, little empirical evidence was provided to support this claim.

The literature related to antipatterns and code smells generally fall into two categories. The first one focuses on detecting antipatterns and code smells (e.g., [10]). The second category concentrates on investigating the relation between antipatterns and software quality (e.g., [11]). Our work in this paper has the same aim as these studies (i.e., the improvement of software quality). Li and Shatnawi [11] investigate relationships between 6 code smells and class error probability in three different versions of Eclipse. They report that classes with antipatterns, such as: God Class, God Method and Shotgun Surgery are positively associated with higher error probability. Moreover, Khomh et al. [8] show that there is a relation between antipatterns and the bug-proneness of a file. These studies provide empirical evidences on the relation between antipatterns and bugs. In this paper, we build up on these previous works to investigate the possibility of predicting bugs in software systems using antipattern information.

Olbrich *et al.* [12] study the evolution of two different code smells (i.e., Shotgun surgery and God class) over time in the development process of two software systems. They conclude that the relative number of components having code smells does not decrease over time meaning that not a lot of refactoring activities are performed on the systems. We also observe this behavior (as shown in Figure 2) on our studied systems; the density of antipatterns does not increase significantly overtime. Peters *et al.* [13] studied the lifespan of 5 different code smells over different releases, and the refactoring behaviour of developers in seven open source systems. They conclude that given the low number of refactorings performed by developers, the number of long-living code smell instances increases over time. These studies show that code smells and antipatterns mostly remain in systems. In this study, we investigate the link between the persistent antipatterns and post release bugs in software systems.

### B. Bug Prediction Models

Researchers have tried to uncover the possible reasons for software bugs using different classes of software metrics, such as process and product metrics [14], [15] or entropy of changes [16]. However, their primary goal has been established

on improving the accuracy of bug prediction (localization) models. Zimmermann *et al.* [14] conducted an empirical study on three different versions of Eclipse to show that a combination of complexity metrics can predict bugs. They conclude that large files (i.e., high LOC values) are more prone to bugs than others. Another case study performed using 85 versions of 12 releases of Apache projects [17] show how and why process metrics are better indicators of bugs with respect to performance, portability and the stability of the model. Moreover, Kamei *et al.* [18] and Chen *et al.* [19] introduce metrics based on the effort and topics in software systems to improve bug prediction models.

Following the same line of work, in this paper, we propose antipatterns as another factor to enhance the accuracy of bug prediction models. More specifically, we propose four new metrics based on the history of antipatterns in files, and perform a case study to verify whether the proposed metrics provide additional explanatory power to bug prediction models built using traditional product and process metrics.

## III. STUDY DESIGN

This section presents the design of our case study, which aims to address the following three research questions:

- 1) Do antipatterns affect the density of bugs in files?
- 2) Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?
- 3) Can we improve traditional bug prediction models with antipatterns information?

### A. Data Collection

Our work studies bug prediction using 12 versions of Eclipse and 9 versions of ArgoUML. Eclipse is a popular IDE used both in open-source communities and in industry. It has an extensive plugin architecture. ArgoUML is an open source UML-based system design tool. These systems encompass different domains and have different sizes. Eclipse is close to the size of real industrial systems (e.g., release 3.3.1 is larger than 3.5 MLOCs), while ArgoUML is a smaller project. Table I shows descriptive statistics of the systems.

### B. Data Processing

Figure 1 shows an overview of our data processing steps. First, we mine the source code repositories of Eclipse and ArgoUML to compute product and process metrics. Next, we detect antipatterns in the two software systems. Then, we mine the bug repositories of the systems to extract information about bugs. Finally, we use statistical models to analyze the collected data and answer our three research questions. The remainder of this section elaborates on each of these steps.

1) *Mining Source Code Repositories:* We download 12 versions of Eclipse and 9 versions of ArgoUML from their respective CVS repositories. We use the Ptidej tool [20] to compute metrics on the source code of each downloaded version. We also use a perl script developed for the purpose of this study to calculate code churn metric values.

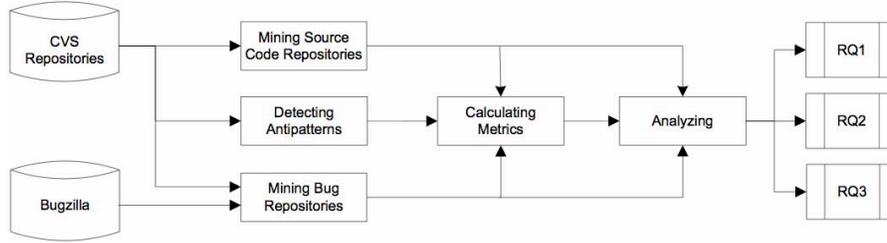


Figure 1. Overview of our data collection process.

Table I  
SUMMARY OF THE CHARACTERISTICS OF THE ANALYSED SYSTEMS

Systems	Releases(#)	Total Number of Antipatterns	Churn	Total Number of Post Bugs	Total Number of Pre Bugs	LOCs
Eclipse	2.0 – 3.3.1(12)	273,766	148,454	27,406	23,554	26,209,669
ArgoUML	0.12 – 0.26.2(9)	15,100	21,427	2,549	2,569	2,025,730

2) *Detecting Antipatterns*: We use the DECOR method proposed by Moha *et al.* [10] to specify and detect antipatterns in our subject systems. DECOR is based on a thorough domain analysis of code smells and antipatterns in the literature, and provides a domain-specific language to specify code smells and antipatterns and methods to detect their occurrences automatically. Moha *et al.* [10] reported that DECOR’s antipatterns detection algorithms achieve 100% recall and an average precision greater than 60%. In this study, we focus on the 13 antipatterns described in Table II. We choose only these antipatterns due to the following reasons: (i) they are well-described by Brown *et al.* [6] and Fowler [7]; and (ii) we could find enough of their occurrences in several releases of our subject systems.

Figure 2 shows the density of antipatterns over the different releases of our subject systems. We define density of antipatterns for a version as the total number of antipatterns over the total number of files in that version. As shown in Figure 2, the density of the antipatterns is quite stable during the evolution of the systems. Our premise in this work is that acting on these antipatterns can help reduce the risk for bugs in the systems.

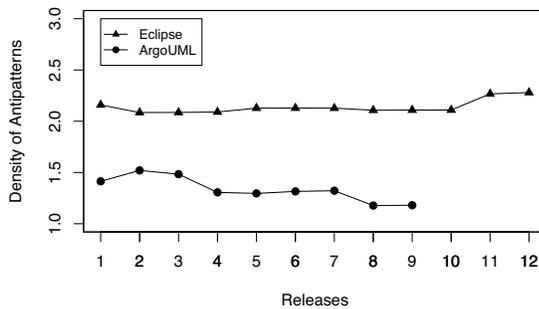


Figure 2. Density of Antipatterns over ArgoUML and Eclipse projects.

3) *Mining Bug Repositories*: For each version of our studied systems, we extract the change logs of all commits performed after the version is released and download bug reports from the bug tracking system (*i.e.*, Bugzilla). We parse the change logs and apply the heuristics proposed by Fisher *et*

Table II  
ANTIPATTERN DEFINITION

Antipatterns	Description
<b>AntiSingleton</b>	A class that provides mutable class variables, which consequently could be used as global variables.
<b>Blob</b>	A class that is too large and not cohesive enough. It monopolises most of the processing, and takes most of the decisions.
<b>ClassDataShould-BePrivate (CDSBP)</b>	A class that exposes its fields, thus violating the principle of encapsulation.
<b>ComplexClass</b>	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
<b>LargeClass</b>	A class that has grown too large in term of LOCs.
<b>LazyClass</b>	A class that has few fields and methods.
<b>LongParameter-List (LPL)</b>	A class that has (at least) one method with a too long list of parameters in comparison to the average number of parameters per methods in the system.
<b>LongMethod</b>	A class that has (at least) a method that is very long, in term of LOCs.
<b>MessageChain</b>	A class that uses a long chain of method invocations to realise (at least) one of its functionality.
<b>RefusedParent-Bequest (RPB)</b>	A class that redefines inherited method using empty bodies, thus breaking polymorphism.
<b>SpaghettiCode</b>	A class declaring long methods with no parameters and using global variables.
<b>SwissArmyKnife</b>	A class that has excessive number of method definitions, thus providing many different unrelated functionality.
<b>Speculative-Generality</b>	A class that is defined as abstract but that has very few children, which do not make use of its methods.

*al.* [21] to identify bug fixes locations. We retain only bugs for which a “bug ID” is found in CVS commits and the Resolution field is set to “FIXED” or the Status field set to “CLOSED”. We refer to the CVS commits as bug fixing commits and extract the list of files that are changed to fix the bug.

4) *Analysis Methods*: We investigate the possibility of using antipatterns to predict bugs in software systems.

a) *Analyzing the relation between the occurrences of antipatterns and the density of future bugs*: We use the Wilcoxon rank sum test [22] to compare the density of

future bugs of classes with and without antipatterns. We define density of future bugs in a file as the total number of bugs over the total LOCs in the file. The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods do not make assumptions about the distributions of assessed variables.

*b) Exploring bug prediction using antipatterns information:* As mentioned before, state of the art metrics can be classified into product metrics (e.g., Lines of Code (LOC)[23]) which are static, and process metrics (e.g., Code Churn [3]) which require historical information on a system. To investigate the use of antipatterns in bug prediction models, we propose new metrics that capture antipatterns information in a system. Then, we build logistic regression models to compare each new antipattern based metric to respectively LOC, PRE, Code Churn and the combination of them. We select LOC, PRE and Code Churn as our baseline metrics since previous studies have found them to be good predictors of bugs in software systems [3], [15], [24], [25]. A similar decision is made in studies by Bird *et al.* [26] and Chen *et al.* [19].

We create the models following a hierarchical modelling approach: we start with our baseline metrics and then build subsequent models by adding step by step, our proposed antipatterns metrics (*i.e.*, APMetric). We chose to follow a hierarchical modelling approach because contrary to a step-wise modelling approach, the hierarchical approach has the advantage of minimizing the artificial inflation of errors and therefore the overfitting [27]. For each model, we compute the variance inflation factors (VIF) [28] of each metric to examine multi-collinearity between the variables of the model. We remove all variables with  $VIF > 2.5$ .

We report for each statistical model the percentage of deviance explained  $D^2$  [29] and the Akaike information criterion (AIC)[30] of the model. The deviance of a model  $M$  is defined as  $D(M) = -2 * LL(M)$ , where  $LL(M)$  is the log-likelihood of the model  $M$ . The deviance explained (*i.e.*,  $D^2$ ) is the ratio between  $D(Bugs \sim Intercept)$  and  $D(M)$ . A higher  $D^2$  value generally indicates a better model fit. AIC is used to compare the fitness of different models. A lower AIC score is better. For each subsequent model  $M_{Base+APMetric}$  derived from a model  $M_{Base}$ , we also test the statistical significance of the difference between  $M_{Base+APMetric}$  and  $M_{Base}$ . We report the corresponding  $p$ -values.

#### IV. STUDY RESULTS

This section presents and discusses the results of our three research questions.

##### **RQ1: Do antipatterns affect the density of bugs in files?**

**Motivation.** Previous work by Khomh *et al.* [8] have shown that files participating in antipatterns are more likely to have bugs than other files. Moreover, in this research question, we examine the density of bugs in files with antipatterns. We want to know when bugs occur in files with antipatterns, they occur in larger number compared to other files or not.

**Approach.** We apply DECOR [10] to specify and detect antipatterns in all the versions of our subject systems as described in Section III-B2. For each version, we classify the files in two groups: a group of files with at least one antipattern, and a group of files without antipatterns. For each file from the two groups, we compute the number of post release bugs in the file as described in Section III-B3. Since previous studies (*e.g.*, [14], [15], [24]) have found that the size of code is related to the number of bugs in a file. To control for the confounding effect of size, we divide the number of future bugs of each file by the size of the file. We obtain the density of future bugs for each file. We test the following null hypothesis:

$H_{01}^1$ : *there is no difference between the density of future bugs of the files with antipatterns and the other files without antipatterns.*

Hypothesis  $H_{01}^1$  is two-tailed since it investigates whether antipatterns are related to a higher or a lower density of bugs. We perform a Wilcoxon rank sum test [22] to accept or refute  $H_{01}^1$ , using the 5% level (*i.e.*,  $p$ -value  $< 0.05$ ). We also compute and report the difference between the average bug densities in the two groups of files with and without antipatterns (*i.e.*,  $D_A - D_{NA}$ ).

Table III  
WILCOXON RANK SUM TEST RESULTS FOR THE BUG DENSITY IN FILES WITH AND WITHOUT ANTIPATTERNS

Eclipse			ArgoUML		
Version	$D_A - D_{NA}\%$	$p$ -value	Version	$D_A - D_{NA}\%$	$p$ -value
2.0	-5.78	<0.05	0.12	-10.75	0.58
2.1.1	-4.36	<0.05	0.14	63.26	<0.05
2.1.2	3.43	<0.05	0.16	8.09	<0.05
2.1.3	19.74	<0.05	0.18.1	19.58	<0.05
3.0	11.60	<0.05	0.20	36.78	<0.05
3.0.1	3.01	<0.05	0.22	72.93	<0.05
3.0.2	13.60	<0.05	0.24	-22.33	0.07
3.2	3.98	<0.05	0.26	-18.71	0.71
3.2.1	-1.82	<0.05	0.26.2	50.75	<0.05
3.2.2	4.23	<0.05			
3.3	19.81	<0.05			
3.3.1	-13.22	0.10			

**Findings. In general, the density of bugs in a file with antipatterns is higher than the density of bugs in a file without antipatterns.** The Wilcoxon rank sum test was statistically significant for 11 out of 12 versions of Eclipse and 6 out of 9 versions of ArgoUML (see Table III). In 8 versions of Eclipse and 6 versions of ArgoUML (highlighted in Table III), the density of bugs in files with antipatterns is significantly higher than the density of bugs in other files.

*Overall, we reject  $H_{01}^1$  and conclude that the occurrence of an antipattern in a file is not only related to a higher risk for bugs in the file (as reported by Khomh *et al.* [8]), but also to a higher density of bugs.*

**RQ2: Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?**

**Motivation.** Antipatterns presented in Table II are detected in software systems using thresholds defined over source code metrics and other lexical information [8]. Since antipatterns refer to specific design and implementation problems in software systems, they are likely to be better indicators than metrics for developers. Indeed, an antipattern can tell developers whether a design implementation is “poor” or not, by means of thresholds defined over metrics and other lexical information; while without antipatterns knowledge, developers would have to judge by themselves which metric values are problematic. Results of **RQ1** show that antipatterns are related to higher numbers of bugs. By acting on antipatterns (e.g., using refactorings), it may be possible to reduce post release bugs in a system. In this question, we investigate this hypothesis in details. We want to understand which proportion of the deviance (i.e., model fitness) in post release bugs can be explained by antipatterns information.

**Approach.** We introduce the following four metrics to capture antipatterns information in a software system.

Let’s  $S^i, i \in \{1 \dots n\}$  be the list of consecutive versions of a system  $S$ , i.e.,  $S^1$  being the first released version and  $S = S^n$ . For each file  $f \in S$ , we denote by  $f^i$  the version of  $f$  in  $S^i$ , i.e.,  $f^i \in S^i, i \in \{1 \dots n\}$  and  $f = f^n$ .

Let’s  $\chi^i$  be the indicator function defined on  $S^i$  by:

$$\chi^i(f^i) = \begin{cases} 1, & \text{if } f^i \text{ contains one or more bugs.} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Let’s  $n_{AP}(f^i)$  be the total number of antipatterns in  $f^i$ ,  $i \in \{1 \dots n\}$ . To capture the distribution of antipatterns in past buggy versions of a system, we introduce the function  $N_{AP}$  defined on  $(\cup S^i)_{i \in \{1 \dots n\}}$  as follows:

$$N_{AP}(f^i) = \begin{cases} \chi^i(f^i) * n_{AP}(f^i), & \text{if } 1 \leq i < n \\ n_{AP}(f^i) & \text{if } i = n . \end{cases} \quad (2)$$

Using  $N_{AP}$ , we define the Average Number of Antipatterns (ANA) metric to capture the distribution of antipatterns in previous buggy versions of a file following Equation (3). For each file  $f \in S$ ,

$$ANA(f) = \frac{1}{n} * \sum_{i=1}^n N_{AP}(f^i), \quad (3)$$

Where  $n$  is the total number of versions in the history of  $f$  and  $f = f^n$ .

To capture the distribution of antipatterns across the files of a specific version  $i \in \{1 \dots n\}$ , we compute the Shannon entropy [31] of antipatterns in  $S^i$  following Equation (4).

$$H^i = - \sum_{k=1}^m p(f_k^i) * \log_2 p(f_k^i), \quad (4)$$

Where  $p(f_k^i) \geq 0, \forall k \in 1, \dots, m$ ;  $m$  is the total number of files in  $S^i$  and  $p(f_k^i)$  is the probability of having antipatterns in file  $f_k^i$ ;  $p(f_k^i)$  is computed following Equation (5).

$$p(f_k^i) = \frac{n_{AP}(f_k^i)}{\sum_{l=1}^m n_{AP}(f_l^i)}, \quad (5)$$

Using the entropy of antipatterns in  $S^i$ , we introduce the Antipattern Complexity Metric (ACM) following Equation (6). This metric is similar to the HCM metric proposed by Hassan et al. [16] to capture the complexity of source code changes.

$$ACM(f) = \sum_{i=1}^n p(f^i) * H^i, \quad (6)$$

Where  $n$  is the total number of versions in the history of  $f$  and  $f = f^n$ .

To capture the consecutive occurrence of antipatterns in a file, we introduce the Antipattern Recurrence Length (ARL) metric following Equation (7).

$$ARL(f) = rle(f) * e^{\frac{1}{n} * (c(f) + b(f))}, \quad (7)$$

Where  $n$  is total number of versions in the history of  $f$ ,  $c(f)$  is the number of buggy versions in the history of  $f$  in which  $f$  has at least one antipattern,  $b(f) < n$  is the ending index of the longest consecutive stream of antipatterns in buggy versions of  $f$ , and  $rle(f)$  is the maximum length of the longest consecutive stream of antipatterns in the history of  $f$  (see [32] for a detailed definition of  $rle$ ).

To illustrate this metric let’s consider a file  $f$  that has 5 previous versions in its history. Let’s assume that the distribution of  $N_{AP}(f^i)$  among these 6 versions (i.e., 5 previous versions and the current version) is as follows:  $\{3, 4, 0, 2, 1, 3\}$ . The value of  $ARL(f)$  is 18.76. Where the value of  $rle(f)$  is 3,  $n$  is 6, the number of buggy versions having antipatterns (i.e.,  $c(f)$ ) is 5, and the ending index of the longest consecutive stream of antipatterns (i.e.,  $b(f)$ ) is 6.

Our last metric is the Antipattern Cumulative Pairwise Differences (ACPD) metric, which aim is to capture the growth tendency of the antipatterns in a file over time. ACPD is computed following Equation (8).

$$ACPD(f) = \sum_{i=1}^n [N_{AP}(f^{i-1}) - N_{AP}(f^i)], \quad (8)$$

Where  $n$  is the total number of versions in the history of  $f$  and  $f = f^n$ . Positive values of ACPD reflect a decreasing tendency of the number of antipatterns over the history of a file.

Using these four metrics (i.e., ANA, ACM, ARL, and ACPD) we build four sets of logistic regression models for every version of our subject systems, following the method described in Section III-B4.

**Findings. Among ANA, ACM, ARL, and ACPD metrics, ARL has the most significant improvement over traditional metrics LOC, PRE, Code Churn.** Figure 3 and Figure 4 show the percentage of deviance explained ( $D^2$ )

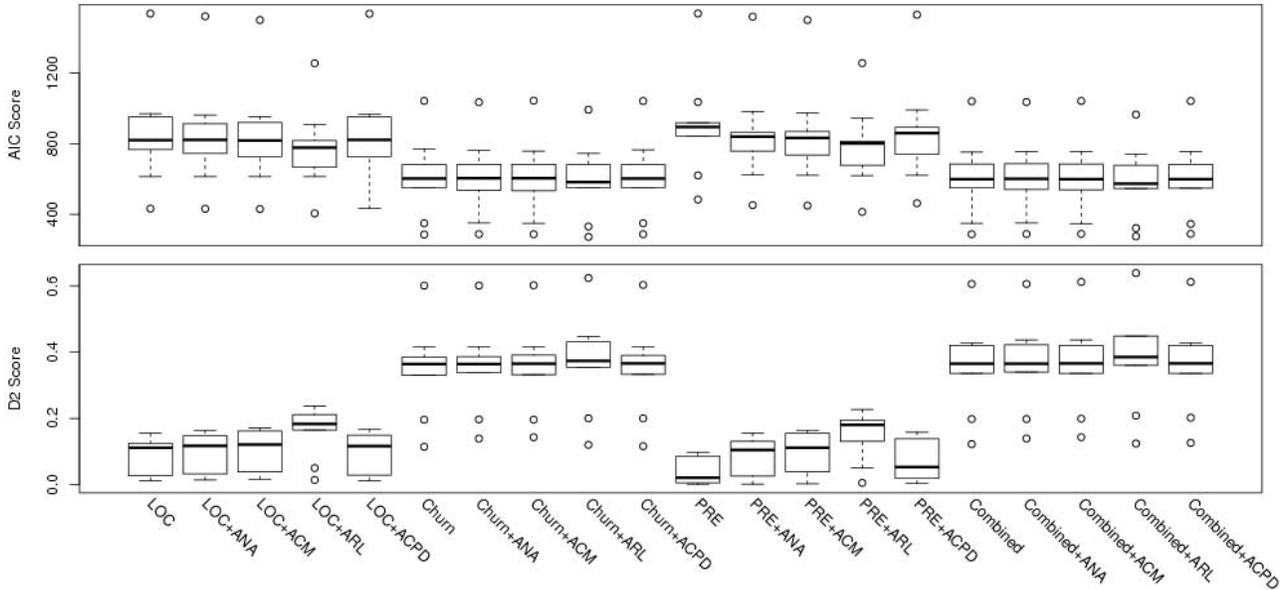


Figure 3.  $D^2$  and AIC scores over 9 versions of ArgoUML by adding proposed metrics to historical software metrics. Each boxplot shows the distribution of  $D^2$  and AIC scores over the different versions of ArgoUML.

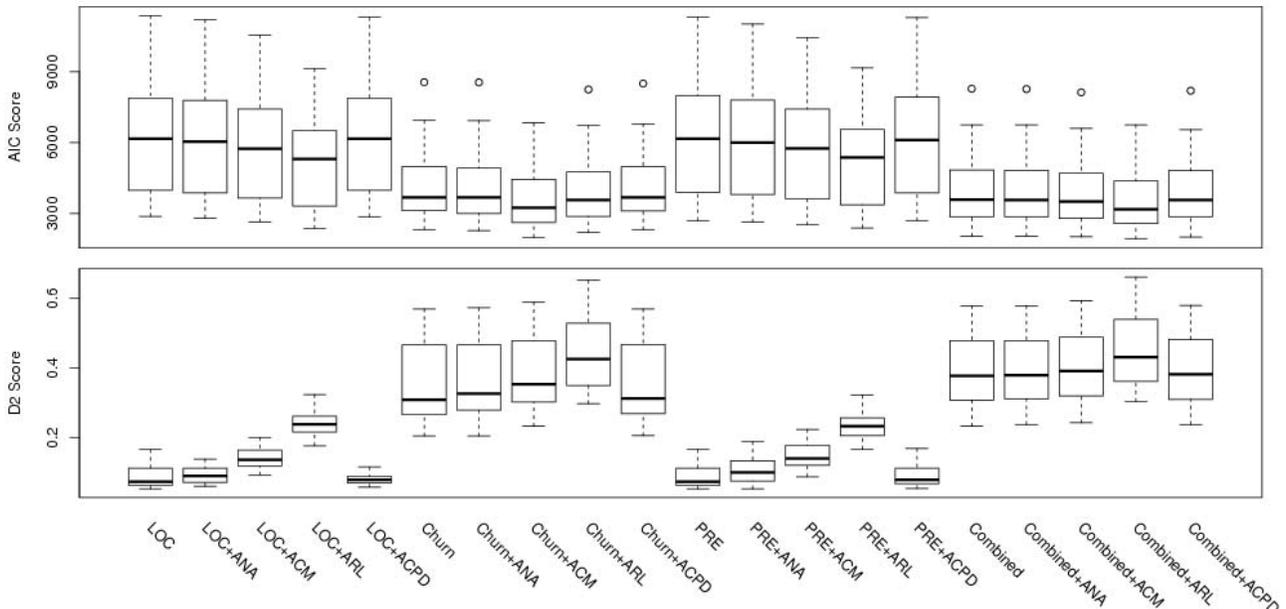


Figure 4.  $D^2$  and AIC scores over 12 versions of Eclipse by adding proposed metrics to historical software metrics. Each boxplot shows the distribution of  $D^2$  and AIC scores over the different versions of Eclipse.

and the Akaike information criterion (AIC) of the model built for each version of ArgoUML and Eclipse from our data set. For each model, a high  $D^2$  score and a low AIC score is desirable. A high  $D^2$  score (respectively a low AIC score) indicates a better model fit. For both ArgoUML and Eclipse versions, we observe that including ANA, ACM, and ARL provides additional explanatory power about the bug-proneness of files over existing traditional product and process metrics (LOC, PRE, Code Churn). The biggest improvement is obtained with the ARL metric, both in terms of AIC and

$D^2$  (i.e., 20% decrease of AIC and 300% increase of  $D^2$ ). Therefore we answer our research question positively. Since ARL captures the proportion of antipatterns with past bugs in streams of consecutive persistent occurrences of antipatterns, we recommend that development teams take the necessary steps to refactor persistent antipatterns that exhibited bugs in the past. Files with long streams of consecutive occurrences of antipatterns should also be refactored. ACM shows the second biggest improvement of explanatory power after ARL. We also recommend that development teams refactor complex

distributions of antipatterns across files since they are likely to be related to a higher risk for bug. Since we observed no improvement with the metric ACPD, it seems that a temporary increase of the number of antipatterns in a system does not necessarily translate into a high risk for bugs. The persistence of antipatterns and the complexity of their distribution across classes in a software system seem to be the main factors behind the increased risk for future bugs observed in systems with antipatterns.

*In conclusion, we found that including the antipattern based metrics ANA, ACM, and ARL provides additional explanatory power about the bug-proneness of files over the following existing traditional product and process metrics LOC, PRE, Code Churn.*

Table IV  
MEASURED PROCESS AND PRODUCT METRICS

	Metric names	Description
Product metrics	LOC	Source lines of codes
	MLOC	Executable lines of codes
	PAR	Number of parameters
	NOF	Number of attributes
	NOM	Number of methods
	NOC	Number of children
	VG	Cyclomatic complexity
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion of methods
	NOT	Number of classes
	WMC	Number of weighted methods per class
Process metrics	PRE	Number of pre-released bugs
	Churn	Number of lines of code added modified or deleted

**RQ3: Can we improve traditional bug prediction models with antipatterns information?**

**Motivation.** In RQ2 we observe that including antipattern information provides additional explanatory power to bug prediction models built using the traditional product and process metrics, LOC, PRE, and Code Churn. The proposed antipattern based metrics ANA, ACM, and ARL are able to increase the deviance explained of models by up to 300%. However, in practice, bug prediction models are not built using only LOC, PRE, and Code Churn. They take into account a variety of other metrics, including those presented in Table IV. Hence, it is interesting to further investigate how our proposed metrics perform in comparison to a more larger collection of metrics.

Another important aspect of bug prediction models is the possibility to apply them across systems. This aspect is particularly important because training data is often not available for software systems from small companies or software systems in their first release (*i.e.*, for which no past data exists). In such situations, development teams attempt to predict bugs using models built and trained on systems from other companies. In this research question, we investigate to what extent one can use cross-system antipattern information to predict bugs. More specifically, we examine whether our proposed antipattern

based metrics can improve traditional bug prediction models within and across systems.

**Approach.** To answer this research question we build two different sets of models: intra-system models and cross-system models.

*A. Intra-system Models*

Intra-system models are built using different versions of the same system. Logistic regression models are generally used for this purpose. In our case, the independent variables are our proposed metrics and a collection of code and process metrics that have been used in previous studies from the literature [33], [15], [14]. Table IV describes the metrics. The dependent variable of our models is a two-value variable that represents whether or not a file has one or more bugs. We broke the process of our analysis into two parts following the approach from Shihab et al. [33]. In the first part, we perform a step-wise analysis to remove statistically insignificant independent variables for each version. This process is repeated until we reach to a model that contains only statistically significant independent variables. Second, we remove highly collinear independent variables from the logistic regression model by controlling the levels of variance inflation. At the end, the remaining independent variables in the model will be statistically significant and minimally collinear. The following sections elaborate more on these steps.

1) *Removing Independent Variables:* In this part we build a multivariate logistic regression model based on our dependent and independent variables, and then in an iterative process we remove the independent variables that are statistically insignificant. We do this process until all the variables are statistically significant. To this end, we use the threshold  $p$ -value  $< 0.1$  to determine whether an independent variable is statistically significant or not.

2) *Collinearity Analysis:* Multicollinearity exists whenever two or more independent variables in a regression model are moderately or highly correlated. Multicollinearity does not reduce the predictive power or reliability of the model as a whole, at least within the sample data themselves, but the problem with multicollinearity is that as the independent variables become highly correlated, it becomes more difficult to determine which independent variable is actually producing the effect on the dependent variable. Tolerance and Variance Inflation Factor (*VIF*) is often used to measure the level of multicollinearity of models. A variance inflation factor (*VIF*) quantifies how much the variance is inflated. A tolerance value close to 1 means that there is no correlation among the independent variables of the model, and hence the variance of our model is not inflated at all. In this paper, we set the maximum *VIF* value to be 2.5, as suggested in [27]. A *VIF* value that exceeds 2.5 requires further investigation, while *VIFs* exceeding 10 are signs of serious multicollinearity that require correction [34]. In this work we use the *VIF* command in the *car* package of R toolkit [35] to examine the *VIF* values of all independent variables used to build the multivariate logistic regression model.

Table V  
P-VALUES OF STATISTICALLY SIGNIFICANT AND MINIMALLY COLLINEAR INDEPENDENT VARIABLES FOR ARGOUML MODELS.

	0.12.0	0.14.0	0.16.0	0.18.1	0.20.0	0.22.0	0.24.0	0.26.0	0.26.2
Churn	2.49e-13***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***
PRE		0.00352**		0.01439*		0.01418*	0.012635*		
LOC					9.17e-07***			0.00553**	
MLOC	0.0596.					7.58e-08***	0.014052*	0.02727*	
NOT							0.068654.		0.0087**
NOF	0.0230*						0.048091*		
NOM						0.00840**			0.0234*
ACM			7.26e-12***						6.77e-06***
ACPD			0.000768***			0.00122**			
ARL		4.68e-05***	<2e-16***	6.6e-05***	3.03e-07***	5.74e-15***	0.000577***		1.63e-10***
AIC	528.73	567.95	911.04	577.12	721.53	546.99	669.29	251.64	288.73
D <sup>2</sup>	0.16	0.42	0.42	0.39	0.37	0.47	0.22	0.50	0.68

Table VI  
P-VALUES OF STATISTICALLY SIGNIFICANT AND MINIMALLY COLLINEAR INDEPENDENT VARIABLES FOR ECLIPSE MODELS.

	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
Churn	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***
PRE	<2e-16***	1.58e-05***	0.000108***		1.17e-09***	2.26e-06***			1.12e-12***		0.00036***	6.65e-07***
LOC									0.00081***			
MLOC								6.46e-11***	2.63e-05***			
NOT	7.25e-05***								0.094773.	0.003649**	1.08e-05***	0.01944*
NOF	0.00054***	0.00980**		0.029776*	0.009207**		0.0417*				0.003904**	
NOM	0.032206*		0.030065*	0.001938**		0.00320**	0.0422*	0.0275*	0.014777*			0.04749*
ACM								8.99e-06***	6.05e-08***			
ACPD						0.00139**		1.42e-06***	6.18e-15***	<2e-16***	0.011604*	
ARL	<2e-16***			<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***		
AIC	3235.8	2111.7	1788.7	2332	2752	4321	6169	2797	4110	3445	2561	5020
D <sup>2</sup>	0.35	0.40	0.45	0.46	0.38	0.36	0.47	0.56	0.54	0.57	0.68	0.58

We narrow down our list of independent variables to consider only those that are statistically significant and minimally collinear with each other (*i.e.*,  $VIF = 2.5$ ). We use these variables to build the final logistic regression model.

**Findings. Among our proposed metrics, ARL remained statistically significant and had a low collinearity with other metrics in almost all the versions of two studied systems.** Table V and Table VI present the results for different versions of ArgoUML and Eclipse. Each column, represents a version of the studied systems. For each version, we report the  $p$ -value of the metrics that remained significant after the first aforementioned iterative process and that have a low collinearity with other independent variables (*i.e.*,  $VIF < 2.5$ ). As one can see, ARL is statistically significant and have a low collinearity with other independent variables for 7 out of 9 versions of ArgoUML, and for 8 out of 12 versions of Eclipse. However, the frequency of occurrence of the other three antipattern based metrics ANA, ACM and ACPD is not considerable. This result shows that ARL captures a different aspect of bug-proneness than the metrics from Table IV. Therefore, it is helpful to include ARL in a model for predicting bugs. Among the metrics from Table IV, *Churn* and *PRE* also have a high impact in predicting bugs. The product metrics *NOF* and *NOM* also contribute significantly but their occurrence in the different models is not persistent over different versions and systems. In conclusion, we recommend that software development teams make use of ARL to improve their bug-prediction models.

## B. Cross-system Models

Cross-system bug prediction is defined as the process of building a model from one system and applying that model to another system in order to successfully predict bugs [4]. To investigate the extent to which cross-system antipattern information can be used to predict bugs, we analyze cross-system bug prediction models built on 12 different versions of Eclipse and 9 versions of ArgoUML, using our antipattern based metrics and metrics from Table IV. We built models for all possible combinations across the systems. Each model was trained on one version of a system and tested on one version of the other system and vice versa. In total we obtained 216 different models. For each pair, we built a logistic regression model using the process and product metrics from Table IV as independent variables, and a two value variable which indicates whether a file has at least one bug or not, as our dependent variable. We calculate the  $F$ -measure of the model by training the model on its corresponding training data (*e.g.*, Eclipse 2.0) and testing it on its corresponding testing data (*e.g.*, ArgoUML 0.12). Next, we add the ARL metric to the previous independent variables and compute the  $F$ -measure of the new model using the same training and testing data. This enables us to measure the potential benefit of ARL for cross-system prediction. The  $F$ -measure is the harmonic mean of *precision* and *recall*. The *precision* of a model is the proportion of bugs that are predicted correctly by the model, while the *recall* is the proportion of real bugs that are predicted successfully by the model.

**Findings. ARL can improve cross-system bug prediction on the two studied systems by an average of 12.5% in terms of  $F$ -measure.** Figure 5 shows distributions of  $F$ -measure for the  $108 \times 2 = 216$  models that were built. On this figure, AE (respectively EA) means that the models were trained on ArgoUML (respectively Eclipse) and tested on Eclipse (respectively ArgoUML). AE\_base (respectively AE\_base + ARL) refers to models built using the metrics from Table IV (respectively the metrics from Table IV and ARL).

The average improvement observed on the  $F$ -measure of models trained on ArgoUML (respectively Eclipse) and tested on Eclipse (respectively ArgoUML), when ARL was added to the models is 10.71% (respectively 12.5%). This result reinforces our previous finding that ARL captures a different aspect of bug-proneness than the metrics from Table IV. Additionally, this result shows that the information captured by ARL is transferable across systems.

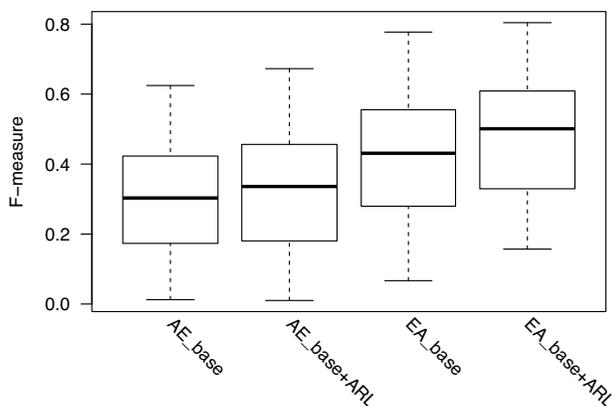


Figure 5. Performance comparison of the models for cross-project prediction when adding ARL to the traditional metrics (Table IV).

*In summary, we observed that our proposed antipattern based metric (ARL) can improve bug prediction models both within and across systems. ARL has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of  $F$ -measure.*

## V. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines for empirical studies [36].

*Construct validity threats* concern the relation between theory and observation. In this study, they are mainly due to measurement errors. For ArgoUML, issues dealing with fixing bugs are marked as “DEFECT” in the issue tracking system. For Eclipse, we mitigated the use of possibly erroneous bugs by discarding issues explicitly labeled as “Enhancements” and focusing on issues marked as “FIXED” or “CLOSED” because they required some changes. It is unlikely, in Eclipse, that hard-to-fix issues would stay longer “OPENED” than others, because Eclipse is being backed up by IBM, which

strives to offer a stable product. To identify bug fix locations, we mine CVS logs and apply the heuristics by Fisher *et al.* [21]. Although this technique may not be a hundred percent accurate, it has been used satisfactorily in many previous studies, *e.g.*, [8], [21]. For the sake of simplicity, we assumed to have one class per file. This assumption could introduce an error in case of non-public top-level classes and inner classes. We did not find any inner class participating in any antipattern in the analysed versions of the systems. Non-public top-level classes are rare and did not participate in any antipattern.

*Threats to internal validity* concern our selection of subject systems, tools, and analysis method. The accuracy of DECOR impacts our results since the number of antipatterns computed with DECOR is used to compute our proposed metrics. Other antipattern detection techniques and tools should be used to confirm our findings.

*Conclusion validity threats* concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models; in particular we used non-parametric tests that do not require any assumption on the underlying data distribution.

*Reliability validity threats* concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets [37]. To mitigate these threats we tested our hypotheses over 12 versions of Eclipse and 9 versions of ArgoUML. Two systems from different size and from different domains. Also, we attempt to provide all the necessary details to replicate our study. The source code repositories and issue-tracking systems of Eclipse and ArgoUML are publicly available to obtain the same data.

*Threats to external validity* concern the possibility to generalize our results. We have studied multiple versions of two systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. In this study, we used a particular yet representative subset of antipatterns. Future work using different antipatterns are desirable.

## VI. CONCLUSION

In this paper, we provided empirical evidence that antipatterns can help predict bugs. To begin with, we show that a file that has antipatterns tends to have a higher density of bugs than other files. Then, we proposed four metrics based on the history of antipatterns in a file to capture antipatterns information in software systems.

We performed a detailed case study using two large real-world software systems (*i.e.*, Eclipse and ArgoUML), to investigate the possibility to predict bugs using the four proposed metrics. The highlights of our analysis include:

- Files that have antipatterns tend to have higher density of bugs than the others (**RQ1**).
- Our proposed metrics can provide additional explanatory power over traditional metrics such as LOC, PRE, Churn.

Among the four proposed metrics, ARL shows the biggest improvement both in terms of AIC and  $D^2$ , i.e., 20% decrease of AIC and 300% increase of  $D^2$  (RQ2).

- ARL can also improve bug prediction models both within and across systems. It has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of  $F$ -measure (RQ3).

In future work, we plan to replicate this study on other systems than Eclipse and ArgoUML to assess the generality of our results. We also plan to investigate more metrics based on antipatterns and clone genealogies.

#### REFERENCES

- [1] N. I. of Standards & Technology, "The economic impacts of inadequate infrastructure for software testing," May 2002, uS Dept of Commerce.
- [2] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 69–82, Feb. 2006.
- [3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.
- [4] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09, 2009, pp. 91–100.
- [5] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec.
- [6] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1<sup>st</sup> ed. John Wiley and Sons, March 1998. [Online]. Available: [www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase\\\_theanti\\\_patterngr/103-4749445-6141457](http://www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase\_theanti\_patterngr/103-4749445-6141457)
- [7] *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [8] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.
- [9] B. F. Webster, "Pitfalls of object-oriented development." 1995, pp. I–X, 1–256.
- [10] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.
- [11] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.
- [12] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09, 2009, pp. 390–400.
- [13] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12, 2012, pp. 411–416.
- [14] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07, 2007, pp. 9–.
- [15] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceeding of the 7th Conference on Mining Software Repositories*, 2010, pp. 31–41.
- [16] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 78–88.
- [17] F. Rahman and P. Devanbu., "How, and why, process metrics are better."
- [18] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, 2010, pp. 1–10.
- [19] T.-H. Chen, S. Thomas, M. Nagappan, and A. Hassan, "Explaining software defects using topic models," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012, pp. 189–198.
- [20] Y.-G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 667–684, Sept.-Oct.
- [21] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept., pp. 23–32.
- [22] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.
- [23] S. G. Crawford, A. A. McIntosh, and D. Pregibon, "An analysis of static metrics and faults in c software," *J. Syst. Softw.*, vol. 5, no. 1, pp. 37–48, Feb. 1985. [Online]. Available: [http://dx.doi.org/10.1016/0164-1212\(85\)90005-6](http://dx.doi.org/10.1016/0164-1212(85)90005-6)
- [24] J. Rosenberg, "Some misconceptions about lines of code," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, Nov, pp. 137–142.
- [25] S. Biyani and P. Santhanam, "Exploring defect data from development and customer usage on software modules over multiple releases," in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, Nov, pp. 316–320.
- [26] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 4–14. [Online]. Available: <http://doi.acm.org/10.1145/2025113.2025119>
- [27] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 864–878, Nov.-Dec.
- [28] M. Kutner, C. Nachtsheim, and J. Neter, *Applied Linear Regression Models*. 4<sup>th</sup> International Edition, McGraw-Hill/Irwin., September 2004.
- [29] J. Nelder and R. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, p. 370384, 1972.
- [30] H. Akaike, "A new look at the statistical model identification," *Automatic Control, IEEE Transactions on*, vol. 19, no. 6, pp. 716–723, 1974.
- [31] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, Jan. 2001. [Online]. Available: <http://doi.acm.org/10.1145/584091.584093>
- [32] G. Held and T. R. Marshall, *Data compression: techniques and applications, hardware and software considerations (2nd ed.)*. New York, NY, USA: John Wiley & Sons, Inc., 1987.
- [33] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 4:1–4:10.
- [34] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10, 2010, pp. 124–133.
- [35] "R toolkit," 19-Dec-2012. [Online]. Available: <http://www.r-project.org>
- [36] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.
- [37] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

# The Evolution of Project Inter-Dependencies in a Software Ecosystem: the Case of Apache

Gabriele Bavota<sup>1</sup>, Gerardo Canfora<sup>1</sup>, Massimiliano Di Penta<sup>1</sup>, Rocco Oliveto<sup>2</sup>, Sebastiano Panichella<sup>1</sup>

<sup>1</sup>University of Sannio, Benevento, Italy

<sup>2</sup>University of Molise, Pesche (IS), Italy

{gbavota, canfora, dipenta}@unisannio.it, rocco.oliveto@unimol.it, spanichella@unisannio.it

**Abstract**—Software ecosystems consist of multiple software projects, often interrelated each other by means of dependency relations. When one project undergoes changes, other projects may decide to upgrade the dependency. For example, a project could use a new version of another project because the latter has been enhanced or subject to some bug-fixing activities. This paper reports an exploratory study aimed at observing the evolution of the Java subset of the Apache ecosystem, consisting of 147 projects, for a period of 14 years, and resulting in 1,964 releases. Specifically, we analyze (i) how dependencies change over time; (ii) whether a dependency upgrade is due to different kinds of factors, such as different kinds of API changes or licensing issues; and (iii) how an upgrade impacts on a related project. Results of this study help to comprehend the phenomenon of library/component upgrade, and provides the basis for a new family of recommenders aimed at supporting developers in the complex (and risky) activity of managing library/component upgrade within their software projects.

## I. INTRODUCTION

Software ecosystems [2], [16] are groups of software projects that are developed and co-evolve in the same environment. These projects share code, depend on one another, reuse the same code, and can be built on similar technologies. Examples of ecosystems—investigated in previous studies about software evolution—can be the plugins developed for a specific platform, such as the universe of Eclipse plug-ins [3], the programs developed with a specific programming language [25], or even using domain-specific language (see for instance the R ecosystem studied by German *et al.* [8]).

When one project undergoes changes and issues a new release, this may or may not lead other projects to upgrade their dependencies. On the one hand, using up-to-date releases of libraries/components may result useful, because these releases can contain new and useful features, and/or possibly some faults may have been fixed. On the other hand, the upgrade of a component may create a series of issues. For example, some APIs may have changed their interface, or might even be deprecated [25], which makes necessary the adaptation of its client. In addition, let us suppose a program uses multiple libraries, namely  $lib_1$  and  $lib_2$ , and  $lib_1$  depends on  $lib_2$ . It can happen that if one upgrades  $lib_2$ , then  $lib_1$  no longer works because does not support the new release of  $lib_2$ . Last, but not least, a library/component might have changed its license making it legally incompatible with the program using it [6]. All these scenarios suggest that *managing the upgrades of libraries/components in large ecosystems is a complex and daunting task, which requires to ponder several factors*. In principle, the problem is dealt with update management tools

available in many operating systems—e.g., Windows, Linux, MacOS—however such update tools either work with entire applications or with operating system related upgrades. Also, they are not able to decide when performing the upgrade and when it might be avoided or postponed.

This paper presents the results of an exploratory study aiming at (i) investigating how dependencies between projects change among the Java subset of the Apache ecosystem; and (ii) exploring and understanding the likely reasons and consequences of such changes. Specifically, the paper investigates:

- 1) how the projects composing the ecosystem evolve and how the dependencies between them change. In the context of this study, we limit our attention to dependencies related to API usage and/or framework usage through extension;
- 2) to what extent are dependencies upgraded (i.e., to a new release of the target project), and what are the drivers of such upgrades;
- 3) how the upgrade of a dependency impacts on the source code of a project.

The investigated ecosystem contains software projects generally related to the domain of Web application (and not only) development, ranging from JSP/Servlet engines (e.g., Tomcat) to Web service containers (Axis), XML parsers (Xerces), and various kinds of support library (e.g., Apache commons or log4j). Overall, *we observed the evolution of 147 projects over a period of 14 years, resulting in a total number of 1,964 releases*.

Results indicate a tangible increase of the dependencies over time. When a new release of a project is issued, in 69% of the cases this does not trigger an upgrade. When, instead, this happens, the likely reasons have to be found in major changes (e.g., new features/services) as well as in large amount of bug fixes. Instead, developers are reluctant to perform an upgrade when some APIs are removed. The impact of upgrades is generally low, unless it is related to frameworks/libraries used in crosscutting concerns.

The paper is organized as follows. Section II describes the study definition and planning, while results are reported in Section III. Section IV discusses the threats that could affect the validity of the results achieved. Section V relates our study with existing literature about the evolution of software ecosystems and evolution/adaptation of APIs. Finally, Section VI concludes the paper and outlines directions for future work.

## II. STUDY DEFINITION AND PLANNING

The *goal* of our study is to analyze how project inter-dependencies evolve in a software ecosystem, with the *purpose* of understanding the likely reasons and consequences of such changes. The *quality focus* is software maintainability, which could be improved by understanding the phenomenon of library/component upgrade. The *perspective* is of researchers interested in understanding when and why developers upgrade dependencies in software ecosystems.

The *context* of our study consists of the entire history of the Java subset of the Apache ecosystem, that represent the vast majority of it (75% of the projects). To date, the entire Apache ecosystem is composed of 195 software projects spread in 23 different categories (e.g., big-data, FTP, mobile, library, testing, XML) and developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases. The size of the ecosystem in the analyzed period of time ranges from 32 up to 28,584 KLOCs, while the number of classes (methods) ranges from 113 to 114,000 (1,386 to 780,731). For sake of clarity, in the following we refer to the project having a dependency toward another project as the “client project”.

### A. Research Questions

The study aimed at providing answers for the following three research questions:

- **RQ<sub>1</sub>**: *How does the Apache ecosystem evolve?* This research question is preliminary to the other two, and aims at providing a picture of the context of our study. Specifically, we analyzed how the number of projects, their size, the dependencies among them, and the declared software licenses changed in the Apache ecosystem during time. Such information represents the foundation for the other research questions.
- **RQ<sub>2</sub>**: *What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on?* Our conjecture is that the client project does not always upgrade a project it depends on when a new release of the latter is available. In this research question we not only aimed at verifying our conjecture, but we also tried to understand what are the reasons driving a client project to upgrade (or not) toward a new available release of a project it depends on. We analyzed as possible factors (i) *structural* changes, captured by analyzing changes in source code of the used project (major/minor); (ii) *functional* changes, captured by analyzing release notes, and (ii) *legal* changes, i.e., those occurring in the declared licenses, that might result in legal incompatibilities between the client project and the project it uses.
- **RQ<sub>3</sub>**: *What is the impact on the client project code of an upgrade of a dependency toward a new available release of a project it depends on?* This research question aims at quantitatively investigating the impact on the source code of the client project when it upgrades a project it uses toward a new available release.

### B. Data Extraction Process

To answer our research questions we first *downloaded the source code* of the 1,964 software releases considered in our study. We used a crawler and a code analyzer developed in the context of the Markos European project<sup>1</sup>. The crawler was able to identify for a given project of interest the list of available releases with their release date as well as its svn address. This information was extracted by crawling DOAP (Description Of A Project) files<sup>2</sup> available on the Internet.

Using the information extracted by the crawler, the code analyzer checked-out the SVN repository and identified the folder containing each of the project releases identified by the crawler. This was done by exploiting the SVN tag mechanism. In other words, the versioning system of Apache projects has a separate directory for each release (where files belonging to such a release are stored), besides keeping the project history in the SVN main trunk. In case the code analyzer did not identify any folder containing a particular release, it reported the problem. During data extraction, such an issue happened for 278 releases that were manually downloaded from the Apache archives, available online for each project<sup>3</sup>.

Once downloaded all the software releases, we *extracted dependencies* existing between them. Note that in this study we focus on dependencies existing between Java Apache projects, ignoring those toward projects external to the Apache ecosystem or not written in Java. Also in this case, the Markos code analyzer has been used. The identification of the inter-project dependencies was performed in different steps. Given a set of software releases, the code analyzer searched—in each folder release—for files that explicitly reported inter-project dependencies. These files in the Apache ecosystem are generally of three types: `libraries.properties`, `deps.properties`, or the Maven `pom.xml` file. Note that the dependency information reported in these files is generally detailed (i.e., both the name of the project as well as the used release are reported) and reliable.

When the code analyzer was not able to find none of these files, it searched for all jar files contained in the release folder and tried to match each of those files with one of the other software releases provided. This is done by computing the Levenshtein distance [17] between the name of the jar file and the name of each provided release. The output of the code analyzer is a list of candidate dependencies between the set of provided software releases.

In our study, we assumed the dependencies extracted by parsing the files `libraries.properties`, `deps.properties`, and `pom.xml` as correct. Instead, when the dependencies were extracted by analyzing jar files in the release folder, we manually validated all candidate dependencies classifying them as *true dependencies* or as *false positives*. This operation was done by two of the authors that analyzed a total of 3,742 dependencies, classifying as correct 832 of them. Overall, the final number of dependencies found in the analyzed 14 years and considered in our study is 3,514.

<sup>1</sup><http://markosproject.berlios.de>

<sup>2</sup><http://projects.apache.org/doap.html>

<sup>3</sup>An example of archive for the Ant project can be found here <http://archive.apache.org/dist/ant/source/>

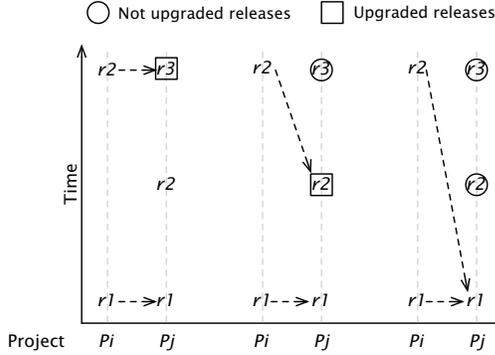


Fig. 1. Process used to divide *upgraded* and *not upgraded* releases.

All the extracted data is not enough to answer all research questions. Indeed, to answer  $\mathbf{RQ}_1$  and  $\mathbf{RQ}_2$  we also identified the software licenses declared in the downloaded software releases. To this aim we used Ninka<sup>4</sup> [10], a lightweight license identification tool for source code that consists on a sentence-based matching algorithm that automatically identifies license from *license statements*. We ran Ninka on each file contained in the 1,964 software releases considered in our study obtaining as output the license type and version declared in its licensing statement (if present).

As for the other factors considered in  $\mathbf{RQ}_2$ , we computed the bug-fixed in each software release and the changes performed between each pair of subsequent releases of the same project. As for the bug-fixing we mined bug-tracking systems of the various projects, extracting only the bugs fixed in each specific release, while the Markos code analyzer was used to extract changes performed among two subsequent releases of each project. We extracted the number of (i) added and deleted classes; (ii) added and deleted public methods; and (iii) changes in existing methods (by distinguishing between public and non public methods).

To classify releases from a functional point of view, we manually analyzed release notes of all the analyzed releases, and classified them using the following tags: *minor* (only improvement of existing features), *major* (new features added), and *bug fixing*. Clearly, the tag *minor* excludes the tag *major*, while the tag *bug fixing* is orthogonal to *minor/major*, and can be assigned to any release note talking about fixed bugs, despite it underwent minor or major changes. This classification has been performed by two of the authors who individually analyzed and tagged the release notes. Then, they performed an open discussion to resolve any conflicts and reach a consensus on the assigned tags.

To answer  $\mathbf{RQ}_3$ , we identified—using again the Markos code analyzer—the source code potentially impacted when an upgrade of a dependency is performed by a client project. The impacted source code is overestimated considering as candidate impact set all the classes of the client project importing at least one class of the upgraded project.

### C. Analysis Method

In order to answer  $\mathbf{RQ}_1$  we analyzed the history of the Apache ecosystem, considering snapshots captured every month. In particular, starting from June 1999, we compute, for each month: (i) the number of existing projects; (ii) the size of the ecosystem in terms of KLOCs; (iii) the dependencies existing between projects; (iv) the software licenses declared in source files.

Concerning the quantitative analysis performed to answer  $\mathbf{RQ}_2$ , and given the dependencies existing between the different releases during time, we verified if releases that are upgraded by client projects (hereby referred as *upgraded releases*) have more changes and/or bug-fixing than releases ignored by client projects (hereby referred as *not upgraded releases*).

To create the two sets of releases (i.e., *upgraded releases* and *not upgraded releases*) we adopted the process depicted in Fig. 1. For each pair of Apache projects,  $P_i$  and  $P_j$ , having at least one dependency between their releases, when  $P_i$  upgrades the dependency toward  $P_j$ , we determined whether  $P_i$  upgrades the dependency toward the last existing release of  $P_j$  or to another release. In the former case, we put the upgraded  $P_j$  release in the set *upgraded releases*. Instead, when the upgrade was not toward the last available release we still put the upgraded  $P_j$  release in the set *upgraded releases*, however we also put the newer ignored releases of  $P_j$  in *not upgraded releases*.

To better understand how we computed such sets, Fig. 1 shows three different evolution scenarios of dependencies between two projects  $P_i$  and  $P_j$ . Let us assume that the release  $r1$  of  $P_i$  depends on the release  $r1$  of  $P_j$ . Then, a new version of project  $P_i$  is released ( $r2$ ). In the first scenario, when  $r2$  for  $P_i$  is released, its dependency is upgraded to  $r3$  of  $P_j$ , the last available  $P_j$  release. In this case,  $r3$  is included in the set *upgraded releases*, while no releases are added to the set *not upgraded releases*, since  $P_i$  correctly upgraded its dependency to the last available  $P_j$  release. In the second scenario (reported in the middle of Fig. 1), the release  $r2$  of  $P_i$  upgrades its dependency to the release  $r2$  of  $P_j$ , even if a newer release (i.e.,  $r3$ ) is available. In this case the release  $r3$  of  $P_j$  has been “ignored” by  $P_i$  and thus, it is added to the set *not upgraded releases*, while release  $r2$  of  $P_j$  is added to the set *upgraded releases*. In the third and last case,  $P_i$  does not upgrade at all the dependencies toward  $P_j$ , i.e., the new release of  $P_i$  continues to use the release  $r1$  of  $P_j$ , despite the availability of more recent releases (i.e.,  $r2$  and  $r3$ ). In this case,  $r2$  and  $r3$  are added to the set *not upgraded releases*, while no releases are added to the set *upgraded releases*.

Note that, if a release  $r_i$  of a project  $P_j$  belongs to the set of *upgraded releases* when analyzing dependencies between  $P_i$  and  $P_j$ , and the same release belongs the set of *not upgraded releases* when analyzing dependencies between a project  $P_s$  and  $P_j$ , the release  $r_i$  is removed from both sets, and not considered any longer in the comparison between *upgraded releases* and *not upgraded releases*. This is done (i) to avoid overlap between the two sets, does not allowing their fair comparison; and (ii) to strongly isolate only releases that are generally upgraded (and not) by client projects.

Besides comparing descriptive statistics, we also used the

<sup>4</sup><http://ninka.turingmachine.org/>

Mann-Whitney test [4] to compare the distribution of changes and bug-fixing for the above described two sets of releases (information extracted through the process described in Section II-B). We assumed a significance level of 95%. We also estimated the magnitude of the difference between the number of changes for the two considered groups of releases (upgraded and not upgraded by clients) using the Cliff’s Delta (or  $d$ ), a non-parametric effect size measure [14] for ordinal data. We followed the guidelines in [14] to interpret the effect size values: small for  $d < 0.33$  (positive as well as negative values), medium for  $0.33 \leq d < 0.474$  and large for  $d \geq 0.474$ .

Finally, to answer **RQ<sub>3</sub>** we report descriptive statistics of the impacted client code in terms of percentage of impacted classes, and percentage of impacted LOCs.

#### D. Replication package

The study described in this section can be replicated using the replication package available online<sup>5</sup>. Such a replication package includes information to download all analyzed projects, as well as working data sets used to answer the study research questions.

### III. ANALYSIS OF THE RESULTS

This section discusses the results achieved aimed at answering the three research questions formulated in Section II-A.

#### A. RQ1: How does the Apache ecosystem evolve?

Fig. 2 reports the evolution over time of the Java Apache ecosystem, in terms of size measured in KLOCs (see Fig. 2(a)), number of projects (black line in Fig. 2(b)) and number of dependencies existing between them (gray line in Fig. 2(b)). As expected, during the analyzed 14 years, the size of the Apache ecosystem grows up exponentially (model fitting resulted in an adjusted  $R^2 = 0.56$ ). From the single Java project existing in 1999 (i.e., Apache ECS<sup>6</sup>) the Apache ecosystem grows up to the 147 Java projects existing today (reflecting also the new developers that started to work in the Apache project teams). Such a growth is linear (adjusted  $R^2 = 0.98$ ). With the increasing of the number of projects also the size—see Fig. 2 (b)—of the entire ecosystem grows, by reaching almost 30 Million LOCs in April 2013. A very strong peak in the size of the ecosystem can be observed between the end of 2006 and the begin of 2007, where the size to the Apache ecosystem redoubled. In this period, several new and big projects have been added to the ecosystem, e.g., Apache UIMA<sup>7</sup> with its 2 millions of LOCs.

Also, dependencies between projects increase continuously during evolution. Similarly to the size, but differently from the number of projects, dependencies follow an exponential trend (adjusted  $R^2 = 0.56$ ). In fact, until 2003 (when about 25 projects were in the ecosystem) there were few dependencies between the projects. After 2003, dependencies sensibly grow in the following years. This is mainly due to the fact that several projects added after 2003 are projects implement reusable components—like those belonging to the Apache

Commons<sup>8</sup>—that are used as libraries by several Apache projects. For example the number of client projects for Apache Commons Compress<sup>9</sup> grows up to 20 (April 2013).

To get a better view on how the Apache software projects and the dependencies between them evolved during time, Fig. 3 shows snapshots of the Apache ecosystem from 2002 to 2013. We ignored the years before 2002 since, as reported in Fig. 2(b), the number of projects (and dependencies) is quite low. In the graphs of Fig. 3, each node represents a project, while an edge connecting two nodes represents a dependency between two projects. By looking at the figure it is clear as the net of dependencies in the ecosystem grows during evolution. Also, focusing on the 2013 snapshot, several *hub projects*, i.e., projects having a lot of client projects, can be noticed. Besides the previously cited Apache Commons project, other *hub projects* are for example Apache Log4j<sup>10</sup> (having 31 client projects), Apache Geronimo<sup>11</sup> (30), and Apache Ant<sup>12</sup> (29). It is worth noting that all these projects implement quite general and reusable features, useful for software projects having different purposes.

As explained in Section II-C, we also analyzed the evolution of the software licenses declared by the Apache projects during time. From 1999 until 2003 we found *Apache Software License (ASL) v1.1* as the only license present in all source code files of all the existing projects. Starting from 2004 all projects started to migrate toward ASL v2.0 and, by the end of 2004, 86% of the source code files in the Apache ecosystem already completed such a migration, leaving the remaining 14% to v1.1. This migration was complete in 2008. In addition to these two licenses, we just found one Apache project (i.e., ApacheTapestry<sup>13</sup>) containing in the majority of its source code files a different license, namely *BSD 3*. However, this does not create any legal issues for potential client projects interested in using ApacheTapestry as library. In fact, the ASL is largely inspired to the *BSD* license and, contrarily to the *GPL* one, source code files having a *BSD* license can be used by source code files having an ASL. Given that the changes in terms of licenses observed during the Apache ecosystem history cannot generate legal issues, in our **RQ<sub>2</sub>** we will not analyze licenses as a possible factor motivating the upgrade of a dependency for client projects.

#### B. RQ2: What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on?

Regarding (**RQ<sub>2</sub>**), we first verified if *not upgraded* releases exist in the Apache ecosystem history. Among the 1,964 releases considered in our study, 950 have been involved in at least one dependency (as client or as library) during their history. Of these 950, 140 releases belong to the 38 projects that have been used as “library project”, i.e., have at least one client project using them. Thus, these are the 140 releases that we classified as *upgraded* or as *not upgraded* by client projects, following the process described in Section II-C. It

<sup>5</sup><http://distat.unimol.it/reports/icsm-apache/>

<sup>6</sup><http://projects.apache.org/projects/eecs.html>

<sup>7</sup><http://uima.apache.org/>

<sup>8</sup><http://commons.apache.org/>

<sup>9</sup><http://commons.apache.org/proper/commons-compress/>

<sup>10</sup><http://logging.apache.org/log4j/>

<sup>11</sup><http://geronimo.apache.org/>

<sup>12</sup><http://ant.apache.org/>

<sup>13</sup><http://tapestry.apache.org/>

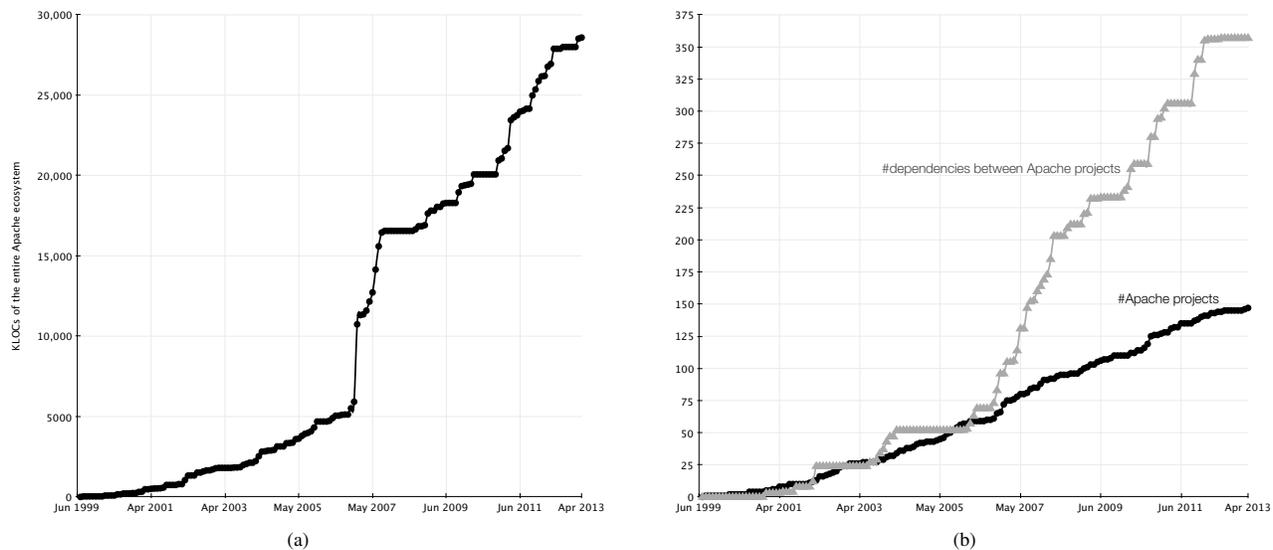


Fig. 2. Evolution of the size (a) and of the projects and dependencies (b) in the Apache ecosystem.

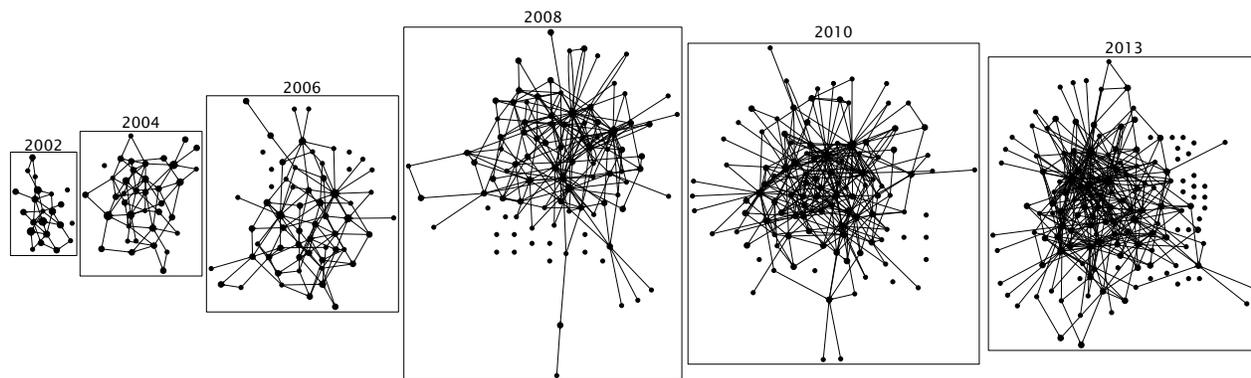


Fig. 3. Snapshots of projects and their dependencies in the Apache ecosystem history.

is worthwhile to note that 14 of these releases have not been assigned to one of the two sets, due to the fact that they are upgraded by some client projects and not upgraded by other clients. Af for the other releases, 87 belong to the *not upgraded* set, while 39 have been assigned to the *upgraded* set. This means that 69% of new releases of Apache software projects are “ignored” by client projects that depend on such projects.

Fig. 4 reports the boxplots for different type of changes for releases that are ignored by client projects (i.e., *not upgraded releases*), and releases used by client projects to upgrade their dependencies (i.e., *upgraded releases*). Moreover, Table I reports the results of the Mann-Whitney test (p-value) and the Cliff’s  $d$  effect size when comparing the distributions for the different types of changes performed on *upgraded* and *not upgraded* releases. On average, in *upgraded releases* there are 25 times more added classes than in *not upgraded releases* (125 vs 5)—see Fig. 4(a). As shown in Table I, this difference is statistically significant (p-value  $<0.0001$ ) with a large effect size (0.62). From Fig. 4 it is clear as, generally, there are no deleted classes (with respect to the previous release) in both kinds of releases—see Fig. 4(b)—and thus, no statistically significant difference.

As for the changes applied to existing methods (i.e., methods already present in a previous release of the project), we observed almost three times more changes for the *upgraded releases* when analyzing all methods in the system (705 vs 217)—see Fig. 4(c)—as well as when just focusing on public methods (that are those used by the client projects), 527 vs 160—see Fig. 4(d). For both types of changes, results in Table I highlight statistically significant differences between *upgraded* and *not upgraded* releases, with a large effect size (0.48) when considering all methods, and a medium effect size (0.46) when just focusing on public methods. Also the number of added public methods is higher in the *upgraded releases* (six times more) than in *not upgraded releases*—see Fig. 4(e)—with statistically significance and a large effect size (0.57).

All these results quantitatively highlight that *upgraded releases* contain changes affecting the interfaces and substantial changes, as compared to the *not upgraded releases*. This is particularly evident when focusing on added classes (29 times more) that are likely related to new features provided by the new project release, and on added public methods (six times more than for *not upgraded releases*), that represent new services available to the client projects. Also, the higher number of overall method changes (three times more than

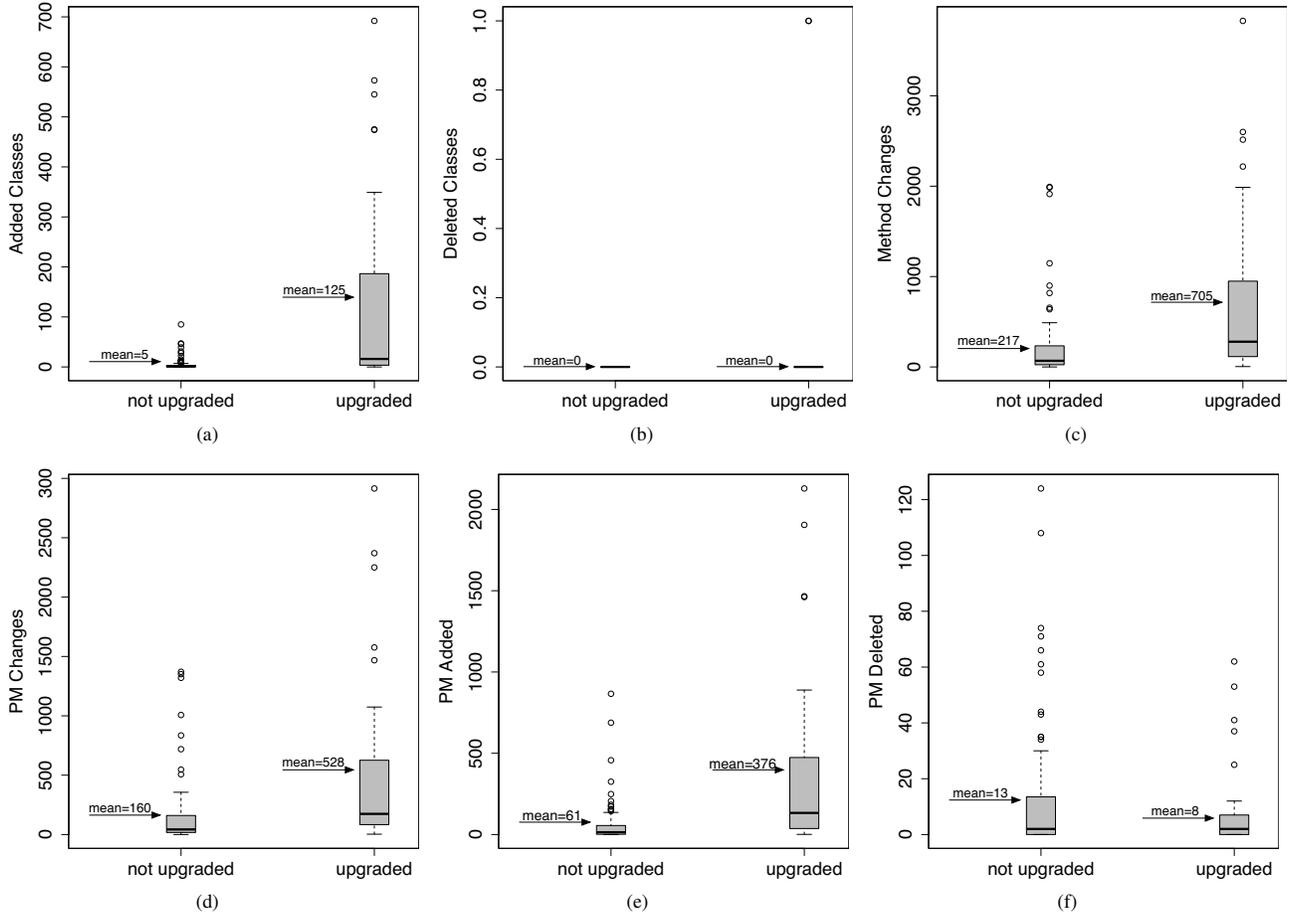


Fig. 4. Changes in upgraded and not upgraded releases.

TABLE I. CHANGES AND FIXED BUGS IN UPGRADED AND NOT UPGRADED RELEASES: MANN-WHITNEY TEST (ADJ. P-VALUE) AND CLIFF'S  $d$ .

Tested	p-value	$d$
Added Classes	<0.0001	0.62 (Large)
Deleted Classes	0.51	0.05 (Small)
Method Changes	<0.0001	0.48 (Large)
PM Changes	<0.0001	0.46 (Medium)
PM Added	<0.0001	0.57 (Large)
PM Deleted	0.48	-0.01 (Small)
Fixed Bugs	<0.0001	-0.35 (Medium)

*not upgraded releases*) highlights substantial changes in the *upgraded releases* as compared to the *not upgraded releases*. The only change for which we did not observe a higher proportion in the *upgraded releases* are the deleted methods (-63%)—see Fig. 4(f). Note that deleted public methods mean removed services for the client projects. Thus, it is reasonable to think that client projects using the removed services tend to not upgrade the dependency towards the new release until they fix the client code in order to properly works with the new release. This could explain the lower number of deleted methods for *upgraded releases*, compared to *not upgraded releases*. However, this difference is not statistically significant (see Table I).

Concerning the number of bugs fixed in *upgraded* and *not upgraded* releases, Fig. 5 reports their distribution. On average,

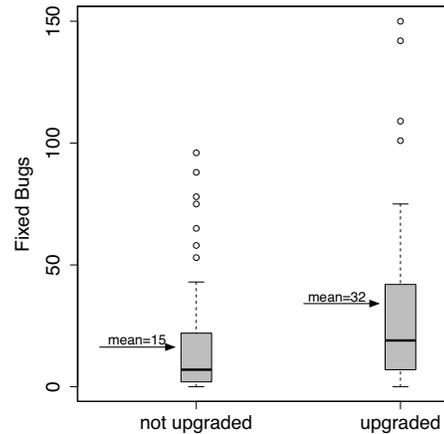


Fig. 5. Fixed bugs in upgraded and not upgraded releases.

the number of bugs fixed in the *upgraded releases* is more than two times greater than for *not upgraded releases* (32 vs 15). Also, this difference is statistically significant with a medium effect size (-0.35)—see Table I.

Summarizing, the results achieved highlighted that on average *upgraded releases* as compared to *not upgraded releases*:

TABLE II. ANALYSIS OF RELEASE NOTES FOR UPGRADED AND NOT UPGRADED SETS OF RELEASES.

Release type	Minor	Major	Bug fixing
<i>not upgraded</i>	79%	21%	82%
<i>upgraded</i>	59%	41%	92%

- 1) *include more new classes and public methods*, likely indicators of new features and services provided by the new release;
- 2) *underwent to more changes* performed on already existing methods, highlighting major changes in the new release;
- 3) *underwent to more bug fixing activities*, removing possible issues experienced by client projects when using the previous release;
- 4) *exhibit less deleted methods*, reducing compatibility problems for developers using them in the client code.

As explained in Section II-C, to provide further evidence to the results reported above, we inspected the release notes of both *upgraded releases* and *not upgraded releases* to understand what are the changes generally declared by developers when releasing both types of releases.

First, we found release notes of *upgraded releases* much longer than those of *not upgraded releases*. For instance, Apache log4j release notes for the six *not upgraded releases* considered in our study are composed, on average, of 676 words each, against the 2,339 words of the five *upgraded releases*. The same difference can be observed between the release notes of the four Apache Ant *not upgraded releases* having an average length of 1,417 words and those of the eight *upgraded releases* with an average of 10,476 words. This suggests that release notes for *upgraded releases* have a longer content, which often means (as confirmed by a manual analysis) describing much more novelties, improvements, and bug-fixes. For example, Apache log4j releases from 1.2.5 to 1.2.8 (4 releases), plus 1.2.11 and 1.2.12 belong to the *not upgraded releases* set. Their six release notes describe, in total, 29 bug fixes and one perfective maintenance activity, the latter being a different option to initialize the system. Instead, release notes for the five *upgraded releases* (i.e., 1.2.9, 1.2.13, 1.2.14, 1.2.16, and 1.2.17) include 123 fixed bugs, two perfective maintenance activities, and one new feature.

Among the six log4j *not upgraded releases*, five have been tagged as *minor* (83%) while one (i.e., 1.2.12) as *major* (17%). Also, all of them have been tagged as *bug fixing*. Concerning the five *upgraded releases*, two (i.e., 1.2.9 and 1.2.13) have been tagged as *minor* (40%), while three as *major*. Also in this case, all five releases have been also tagged as *bug fixing*.

The classification of the inspected release notes is reported in Table II. As we can see, 79% of *not upgraded releases* have been tagged as *minor*, against 59% of the *upgraded releases*, while 41% of *upgraded releases* have been tagged as *major*, against 21% of the *not upgraded releases*. Concerning the bug-fixing activities declared in release notes, overall 82% of the release notes for *not upgraded releases* have been tagged as *bug fixing*, against 92% of the *upgraded releases*.

Overall, the inspection of the release notes confirms that *client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including bug-fixing activities*.

TABLE III. IMPACTED SOURCE CODE COMPONENTS IN CLIENT PROJECTS.

	#Classes (%)	#KLOC (%)
Mean	58 (5%)	65 (6%)
Median	6 (1%)	12 (1%)
St. Dev.	122 (9%)	14 (12%)
Max	518 (41%)	77 (62%)
Min	1 (0%)	39 (0%)

*C. RQ3: What is the impact on the client project code of an upgrade of a dependency toward a new available release of a project it depends on?*

Table III reports descriptive statistics of the impacted source code of client projects upgrading one of their dependencies. The values are reported in terms of impacted number (percentage) of classes and number (percentage) of KLOCs of the client project.

On average, the impacted source code of the client project is quite limited, about 5% of the total number of classes and 6% of the KLOCs. This is quite expected, since most the dependencies a client project has are just due to few classes exploiting the services provided by this dependency. For instance, all the dependencies toward the Apache Commons projects are generally due to few methods in the client code exploiting the offered services, like the `compressors` and `archivers` services provided by the Apache Commons Compress project to manipulate archive files, or the collection of I/O utilities available in the Apache Commons IO project. Since these services support the implementation of specific tasks, it is expected that they just impact on classes having such tasks among their responsibilities.

Instead, there are some projects offering very wide services, representing crosscutting concerns exploited by a great part of the client project source code. This consideration is derived by the analysis of the row “Max” in Table III, reporting the maximum value of impacted client source code we measured in our study. This value is referred to a dependency that the project Apache Accumulo<sup>14</sup> (client project) has toward the project Apache Hadoop. Accumulo is a database system, while Hadoop is a framework supporting distributed processing of large data sets across clusters of computers using simple programming models. Accumulo exhibits dependencies toward Hadoop in 518 of its 1,263 classes (41%) for a total of 77 KLOCs impacted (62% of the total size). Other projects exhibiting an high impact on the client code are Apache Tomcat<sup>15</sup>, impacting on average 23% of the client projects KLOCs, and Apache MINA<sup>16</sup> with an average of 10%. Again, both projects offer very generic services that could be reasonably exploited by several classes in the client projects. In fact, Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages (JSP) technologies, while Apache MINA is an application framework helping users in developing high performance and high scalability network applications.

On summary, results of **RQ3** highlight that *the proportion of source code of client projects impacted by changes in the projects they depend on is quite limited, around 5%. However, there are specific dependencies, generally toward*

<sup>14</sup><http://accumulo.apache.org/>

<sup>15</sup><http://tomcat.apache.org/>

<sup>16</sup><http://mina.apache.org/>

frameworks/libraries offering very wide, crosscutting services, that could strongly impact the client project source code when a dependency is upgraded.

#### IV. THREATS TO VALIDITY

This section discusses the threats that can affect the validity of the results achieved. Threats to *construct validity* concern the relation between the theory and the observation. They can be mainly due to imprecisions in the measurements we performed. This is a summary of the main sources of imprecision:

- the mapping between dependencies declared within a project and other projects was performed using a set of heuristics, as explained in Section II-B. To cope with the imprecision of such heuristics, results were manually verified;
- the analysis of change impact done in **RQ<sub>3</sub>** includes all client classes importing an API class that underwent a change. To determine whether a changed method was used or not, a fine-grained analysis would have been necessary. However, this was not our intent. Instead, we were interested to determine the potential set of clients for the changed API class, i.e., a set of classes that might need some verification/testing activities;
- analysis of licensing relies on the precision of Ninka, which is deemed to be higher than 90% [10].

Threats to *internal validity* concern factors internal to the study that could influence our results. Such kind of threats typically do not affect exploratory studies like the one in this paper. The only case worthwhile of being discussed is about **RQ<sub>2</sub>** (reasons for upgrades) and to some extent **RQ<sub>3</sub>** (why some changes in libraries have more impact than others). In the first case, although we have found some correlation between certain kinds of changes and upgrades decisions, we cannot claim there is a cause-effect relation. Nevertheless, we manually inspected release notes to support our findings. Similar considerations apply to **RQ<sub>3</sub>**, where the cases of large impact were fairly limited—i.e., to framework such as Accumulo and Hadoop—and it was possible to manually verify our findings.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. The analyses performed in this paper mainly have an observational nature, although we used, where appropriate (**RQ<sub>2</sub>**), statistical procedures and effect size measures to support our claims.

Threats to *external validity* concern the generalizability of our findings. Such a generalizability is clearly limited to the ecosystem being analyzed, i.e., Apache, and specifically Java projects of the Apache ecosystem. Also, in terms of assessing dependency upgrades, such assessment is confined to *within-ecosystem* dependencies, as we are not interested to analyze dependencies to projects that are not part of the ecosystem. Future studies need to be done to investigate upgrades with respect to external dependencies too, and to repeat the study on other ecosystems.

#### V. RELATED WORK

In recent and past years several papers have analyzed software ecosystems to investigate how and why a single

project became an ecosystem of more than one software project. Authors of these works focused their attention on methods to analyze the evolution of software projects, as well as methods to extract dependencies between projects belonging to the ecosystem. In this section, we discuss studies aimed at analyzing software ecosystems. Also, we discuss studies that observed changes/deprecations of APIs and their impact on software evolution and stability.

##### A. Analysis of Software Ecosystems

During the software development/evolution of a software project, the complexity and dimension of the project increase in terms of (i) the number of components the software system is composed by; and (ii) number the developers teams. For example, the size of the Debian ecosystem doubles in size every 2 years [11]. In addition, large projects evolve rapidly through the evolution of a set of depending sub-projects [9], [11], [12]. This means that a software system, during its evolution, becomes part of a larger software ecosystem, developed in the context of an organization or an open-source community [18].

Software ecosystems have been studied in the last decade from several different perspectives. Lungu [18], [19] show how reverse engineering an ecosystem is a natural and complementary extension to the traditional system reverse engineering. In a previous work [20], Lungu *et al.* focused their effort on reverse engineering a software ecosystem by generating high-level views capturing various aspects of its structure and evolution.

Gonzalez-Barahona *et al.* [12] studied and analyzed the Debian Linux distribution founding that large source code data does not necessarily involve an ecosystem. However, in large software systems, knowing the dependencies between modules or components is critical to assess the impact of changes. In software ecosystems, which is composed by a collections of software projects, recover all the dependencies is not a simple problem. For this reasons several authors [21], [23] focused their effort on methods for the extraction of dependencies between projects in an ecosystem. In our study, we use some specific heuristics (see Section II-B) to identify the dependencies in the Apache ecosystem.

Grechanik *et al.* [13] have studied the structural characteristics of the source code of 2080 randomly chosen Java open source projects, by answering 32 research questions related to (i) classes and packages, (ii) constructors and methods, (iii) fields, (iv) statements, (v) exceptions, (vi) variables and basic types, and (vii) evolution/maintenance activities occurred on the projects.

The Eclipse ecosystem has been studied by several authors from different perspectives. Wermelinger *et al.* [26] identified a stable core of Eclipse plugins whose dependencies have remained stable over time. Other studies analyze the evolution of Eclipse of both core [22] and third-party Eclipse plugins [3]. In particular, Mens *et al.* [22] found that the Eclipse core plugins adhere to the laws of continuing change and growth, but not to the law of increasing complexity. Businge *et al.* [3] instead, analyzed the dependencies and the survival of 467 Eclipse third-party plugins, altogether having 1,447 versions.

They found how plugins depending on only stable and supported Eclipse APIs have a very high source compatibility success rate, compared to those that depend on at least one of the non-APIs that are those that depend on at least one of the potentially unstable, discouraged and unsupported Eclipse non-APIs. This means that third-party plugins that depend from the Eclipse ecosystems (stable and supported Eclipse APIs) have a higher source compatibility success rate than discouraged and unsupported Eclipse non-APIs. In addition, they found that the majority of plugins hosted on SourceForge do not evolve beyond the first year of release.

Recently, German *et al* [8] analyzed the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that the ecosystem of user-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages, yet each individual package remains stable in size. In our study, similar to the work by German *et al.* [8], we analyzed the evolution of the Apache ecosystem. Differently from R, in Apache the changes of a sub-project (package in R) happen very often during the year. This recall the need to study how and why a client project upgrades a dependency toward a new available release of a project it depends on.

Recently, Annosi *et al.* [1] proposed a framework to support developers in the upgrade of third-party components. The decision is driven by various factors, partially related to the kind of change occurred in the component (as mined from release notes or issue trackers), partially on expert judgements collected within the company. The work presented in this paper is complementary to the work of Annosi *et al.*, because it helps to identify what are the factors and events that trigger component upgrades in a large software ecosystem.

### B. Analysis of API Changes

Theoretically, the API of a component should never change. In practice, when a new version of a software component is released, it is very likely that its interface changes. This requires projects that use the component to be changed before the new release of the component can be used. How and why API changes during software evolution has been studied by several authors. Dig *et al.* [7] studied the changes between two major releases of four frameworks (one proprietary and three open-source) and one library written in Java. They found that on average 90% of the API breaking changes<sup>17</sup> are represented by refactoring operations.

Hou *et al.* [15] analyzed the evolution of AWT/Swing at the package and class level. They found that—during 11 years of the JDK release history (i.e., since JDK 1.0 to Java SE 6)—the number of changed elements was relatively small as compared to the size of the whole API, and the majority of them happened in release 1.1. Thus, the main conclusion of their study is that the initial design of the APIs contributes to the smooth evolution of the AWT/Swing API. Raemaekers *et al.* [24] studied changes in APIs to measure the stability of the Apache Commons library. Their findings indicated that a relatively small number of new methods were added in each

snapshot to the “Commons Logging” library, and there is more work going on in new methods of “Common Codec” than in old ones.

Recently, Robbes *et al.* [25] observed how much the API of a framework (or library) changes. They studied API deprecations that led to ripple effects across an entire ecosystem. The results showed that a number of API changes caused by deprecation can have a very large impact on the ecosystem and consequently on projects or developers that are impacted by the change, or the measure of the overall number of changes.

Changes in APIs and frameworks require the adaptation of clients, that can, sometimes, be automated. To this aim, Degenais and Robillard [5] proposed SemDiff, a tool to recommend client adaptation required when the used framework evolve. The authors evaluated SemDiff on the evolution of the Eclipse-JDT framework and three of its clients.

We share with the aforementioned papers the need for studying how the evolution of projects used as libraries in software ecosystems impacts on the evolution of client projects. However, instead of proposing how client projects should be adapted, we aimed at analyzing to what extent are dependencies upgraded—i.e., towards a new release of the target project—and what are the drivers of such upgrades. Our study provides some insights on the design of recommendation systems for supporting developers in the activity of library/component upgrade.

## VI. CONCLUSION AND FUTURE WORK

In this paper we analyzed the evolution of dependencies between projects belonging to the Java subset of the Apache ecosystem. Our study aims at providing some insights on (i) how the dependencies between projects composing the ecosystem change during software evolution; (ii) to what extent are dependencies upgraded, and what are the drivers of such upgrades; and (iii) how the upgrade of a dependency can impact on the source code of a project. In the context of our study, we observed the evolution of 147 projects over a period of 14 years, resulting in a total number of 1,964 releases.

Results of our study indicate that projects and their dependencies increase continuously during evolution. However, dependencies follow a different trend as compared to the number of projects of the ecosystem. Specifically, while the trend of number projects is linear, the number of dependencies between them grows exponentially. As for the upgrade of dependencies, we observed that, when a new release of a project is issued, in 69% of the cases this does not trigger an upgrade. Client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including major bug-fixing activities, change to API interfaces or addition of new API and features. Instead, developers are reluctant to perform an upgrade when some APIs are removed. The impact of upgrade is generally low, unless it is related to components providing features which is used in different points of a project.

On the one hand, the findings of this study allow to understand the phenomenon of software ecosystem evolution and, specifically, of library/component upgrade. Because of the addition of new features, the relationship between projects

<sup>17</sup>API breaking changes would cause an application built with an older version of the component to fail under a newer version.

in the ecosystem become more and more complex, and the upgrade management becomes cumbersome. On the other hand, achieving a deep understanding on when upgrades were performed and when not is useful to pose the basis for the development of smart upgrade management systems that, instead of just triggering updates whenever a new version of a component is available, are able to support the software engineer in the decision of whether to (i) perform the upgrade immediately (e.g., in case of a major release or important bug-fixing on the component), (ii) postpone it (minor fixes, that would just require to allocate effort for performing change impact analysis without any major advantage), or (iii) even avoid to perform it, e.g., when APIs are deprecated or when the upgrade can create licensing incompatibility issues.

In future work, we plan to continue studying how dependencies evolve in software ecosystems, considering other factors. For instance, the social/community aspects of ecosystems could play an important role in the evolution of dependencies between projects, i.e., the presence of joint development teams, and/or communication between teams could be important drivers for dependency upgrade as well as to promote code reuse between projects.

#### ACKNOWLEDGEMENTS

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, and Sebastiano Panichella are partially funded by the EU FP7-ICT-2011-8 project Markos, contract no. 317743. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

#### REFERENCES

- [1] M. Annosi, M. Di Penta, and G. Tortora. Managing and assessing the risk of component upgrades. In *Product Line Approaches in Software Engineering (PLEASE)*, 2012 3rd International Workshop on, pages 9–12, 2012.
- [2] J. Bosh. From software product lines to software ecosystems. In *Proceedings of the 13th International Conference on Software Product Lines (SPLC)*, pages 111–119, 2009.
- [3] J. Businge, A. Serebrenik, and M. van den Brand. Survival of eclipse third-party plug-ins. In *28th IEEE International Conference on Software Maintenance (ICSM 2012)*, Trento, Italy, Sep 23–28, 2012, pages 368–377. IEEE Computer Society, 2012.
- [4] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.
- [5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *30th International Conference on Software Engineering (ICSE 2008)*, Leipzig, Germany, May 10–18, 2008, pages 481–490. ACM, 2008.
- [6] M. Di Penta, D. M. Germán, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010*, pages 145–154. ACM, 2010.
- [7] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:83–107, 2006.
- [8] D. German, B. Adams, and A. E. Hassan. Programming language ecosystems: the evolution of r. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 243–252, Genova, Italy, 2013.
- [9] D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07*, pages 140–149, Washington, DC, USA, 2007. IEEE Computer Society.
- [10] D. M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10*, New York, NY, USA, 2010. ACM.
- [11] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 131–140, Washington, DC, USA, 2000. IEEE Computer Society.
- [12] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Softw. Engg.*, 14(3):262–285, 2009.
- [13] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16–17 September 2010, Bolzano/Bozen, Italy*. ACM, 2010.
- [14] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates, 2nd edition edition, 2005.
- [15] D. Hou and X. Yao. Exploring the intent behind api evolution: A case study. In *18th Working Conference on Reverse Engineering (WCRE'11)*, Limerick, Ireland, Oct 17–20, 2011, pages 131–140, 2011.
- [16] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *31st International Conference on Software Ecosystems, New and Emerging Research Track*, pages 187–190, 2005.
- [17] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–716, 1966.
- [18] M. Lungu. Towards reverse engineering software ecosystems. In *24th IEEE International Conference on Software Maintenance (ICSM 2008)*, September 28 - October 4, 2008, Beijing, China, pages 428–431. IEEE, 2008.
- [19] M. Lungu. Reverse engineering software ecosystems. Technical report, University of Lugano, 2009.
- [20] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Sci. Comput. Program.*, 75(4):264–275, 2010.
- [21] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *In Proceedings of ASE 2010*, pages 309–312. ACM Society Press, 2010.
- [22] T. Mens, J. Fernández-Ramil, and S. Degrandt. The evolution of eclipse. In *24th IEEE International Conference on Software Maintenance (ICSM 2008)*, September 28 - October 4, 2008, Beijing, China, pages 386–395. IEEE, 2008.
- [23] J. Ossher, S. K. Bajracharya, and C. V. Lopes. Automated dependency resolution for open source software. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE)*, Cape Town, South Africa, May 2–3, 2010, *Proceedings*, pages 130–140. IEEE, 2010.
- [24] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM'12)*, Trento, Italy, Sep 23–28, 2012, pages 378–387, 2012.
- [25] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 56:1–56:11, New York, NY, USA, 2012. ACM.
- [26] M. Wermelinger and Y. Yu. Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 133–136, New York, NY, USA, 2008. ACM.

# Stakeholders' Information Needs for Artifacts and their Dependencies in a Real World Context

Sebastian C. Müller  
 Department of Informatics  
 University of Zurich, Switzerland  
 smueller@ifi.uzh.ch

Thomas Fritz  
 Department of Informatics  
 University of Zurich, Switzerland  
 fritz@ifi.uzh.ch

**Abstract**—In the evolution of software, stakeholders continuously seek and consult various information artifacts and their interdependencies to successfully complete their daily activities. While a lot of research has focused on supporting stakeholders in satisfying various information needs, there is little empirical evidence on how these information needs manifest themselves in the context of professional software development teams of real world companies. To investigate the information needs of the different stakeholder roles involved in software evolution activities, we conducted an empirical study with 23 participants from two professional development teams of one company. The analysis of the gathered data shows that information needs exhibit a crosscutting nature with respect to stakeholder role, activity, artifacts and even fragments of artifacts. We also found that the dependencies between information artifacts are important for the successful performance of software evolution activities, but often not captured explicitly. The lack of an explicit representation of these interdependencies often result in difficulties identifying dependent artifacts and additional communication effort. Based on our findings, we suggest ways to better support stakeholders with their information needs.

**Index Terms**—information needs, crosscutting nature, stakeholder role, activity, artifact, fragment, missing link

## I. INTRODUCTION

Software evolution requires the coordination and collaboration of multiple stakeholders performing a variety of activities (e.g., [1], [2]), where many of these activities deal with already existing code, such as analyzing and fixing bugs in a system. In this process, large amounts of information are continuously evolved and recorded in various kinds of artifacts, such as source code, bug reports or requirements documents. These information artifacts and their dependencies are then continuously consulted by the stakeholders of the system, to successfully complete their activities. For instance, a software developer might have to understand a bug reported by a software tester and find the right part in the source code to fix the bug, or a software tester might have to examine the requirements specified by the requirements engineer to see if his newly created test cases actually cover them.

Although there is a lot of research to support stakeholders with their information needs, even across multiple artifacts (e.g., [3], [4]) or to recover the often implicit dependencies between artifacts (e.g., [5], [6]), there is little evidence on how these information needs manifest themselves in the workspace of real world companies. Existing evidence on information

needs mainly stems from retrospective analyses of repositories (e.g., [7], [8], [9]) and studies predominantly conducted in a lab setting that either focus on a single kind of information artifact, a single activity or a single stakeholder role (e.g., [10], [11], [12], [13], [14], [15]).

To investigate stakeholders' information needs and their manifestation in a real world context, we conducted an empirical study with 23 participants from two different software development teams of one European company<sup>1</sup>. This study was composed of two parts, a diary study and a follow-up interview. In this study, we focused on the following two research questions:

- What are the characteristics of information artifacts needed by the different kinds of stakeholder roles involved in the daily software evolution activities?
- How are these artifacts interdependent?

We found that information needs exhibit a crosscutting nature with respect to stakeholder role, activity, artifacts and even fragments of artifacts. Furthermore, we found that the dependencies between various information artifacts are important for stakeholders to successfully perform their activities but are often not explicitly represented, causing difficulties for individuals to understand and identify dependencies as well as additional communication effort.

This paper makes the following contributions:

- it identifies characteristics of the information artifacts needed by stakeholders in the context of a real world company;
- it identifies characteristics of the dependencies between these artifacts needed by stakeholders for their daily activities; and
- it provides a discussion and implications of the crosscutting nature of information needs.

## II. RELATED WORK

Work related to our approach can broadly be categorized into three major areas: first, studies investigating stakeholders' activities in the software evolution process as well as the information sharing and communication practice during these activities, second, research focusing on the information needs

<sup>1</sup>Due to intellectual property, we are not able to disclose more specific information about the company.

of stakeholders, and third, research looking at the dependencies between information artifacts that are relevant for the activities of a stakeholder. In the following, we will provide an overview over each of these areas by sampling the work in each area.

Various studies have looked at stakeholder activities in the software evolution process. For instance, Perry *et al.* [1] conducted two studies to investigate how software developers spend their time and found that software developers spend more than half of their time on non-coding activities. LaToza *et al.* [16] provide an overview on the typical activities of software developers as well as the practices and tools they use to perform these activities based on a set of surveys and interviews. Singer *et al.* [17] performed four different studies to investigate the daily activities of software developers and found that they spend a lot of time on reading or writing documentation, interacting with the source code and various search queries. Schröter *et al.* [18] investigated the communication between software developers in a software project, focusing on the factors that influence the communication behavior around change sets. Similarly, Aranda *et al.* [8] investigated the coordination needs and patterns of developers by looking at the bug reports of resolved bug fixes. Finally, De Souza *et al.* [19] conducted a field study to investigate the interdependencies in the process of software development, in particular, assessing the approaches a software development team uses to coordinate their work flow. Different to our work, all these studies do not examine the characteristics of the information needed to perform an activity and are often limited to the software developers or a single activity rather than looking at the multiple stakeholders involved in software evolution.

Other research has focused on the information needs and the questions of individual stakeholder roles, in particular software developers<sup>2</sup>. This research area can be further divided into studies conducted with software developers and retrospective analysis of repositories. In the studies with software developers, Sillito *et al.* [10], for instance, observed software developers while performing change tasks and identified a catalogue of common source code related questions that developers ask themselves. LaToza *et al.* [12] investigated more specifically the reachability questions that developers ask for the code and that are difficult to answer. Ko *et al.* [11] identified 21 more general types of questions that software developers ask themselves on a daily basis, the information developers seek to answer these questions and the difficulties developers have to acquire them. Fritz *et al.* [13] conducted a study to specifically identify the questions that require multiple kinds of information artifacts and are often difficult or infeasible to answer. To investigate a stakeholder's information needs using a repository, Breu *et al.* [9] looked at bug reports and investigated the information needs of software developers documented in these bug reports. They identified

<sup>2</sup>With the term software developer we will refer to the stakeholder role that is focused on the coding aspect, often also called a programmer.

eight question categories from the questions asked in bug reports and found that the information needs evolve with the bug life cycle. Erdem *et al.* [7] examined messages posted to the Usenet newsgroup to analyze what information people ask for and came up with a question classification scheme. In a more recent study, Treude *et al.* [20] looked at Stack Overflow, analyzed the tags that are used there for questions and found ten categories of questions, such as error and how-to questions. All of these studies mostly paid little to no attention on how the information needs manifest themselves in the team context of real world companies and either focused on a single stakeholder role or a single activity. Studies that examined professional developers were conducted by Roehm *et al.* [14] and Seaman [15]. In their study, Roehm *et al.* [14] investigated the strategies that developers follow to comprehend software as well as the information artifacts and tools developers use in the comprehension process. Seaman [15] paid more attention on software maintainers' information needs and conducted a survey to assess how software maintainers acquire the information they need to perform their maintenance tasks. Different to these two studies, we looked at multiple stakeholder roles.

Dependencies between information artifacts is a widely discussed topic, particularly concerning the recovery of missing dependencies between various information artifacts for traceability reasons. There are a variety of approaches using information retrieval methods to recover the links between multiple information artifacts, such as documentation and source code (*e.g.* [5], [21], [22]). In most of these papers, the authors assume that stakeholders consider these interdependencies as important, since they support stakeholders while performing a number of typical software evolution and maintenance tasks. However, to the best of our knowledge, we do not know of any research that investigates these interdependencies and their manifestations in a real world context.

### III. EMPIRICAL STUDY DESIGN

The goal of this study is to investigate the information needs of the multiple stakeholders, such as software developers and requirements engineers, in the context of a real world company. In particular, we were interested in the following two question:

- What are the characteristics of information artifacts needed by the different kinds of stakeholder roles involved in the daily software evolution activities?
- How are these artifacts interdependent?

To study these questions in the context of evolving software, we conducted an empirical study with 23 participants from two professional software development teams of a big European company<sup>3</sup>. We chose these two teams due to the access we were granted to all stakeholders. The study was composed of a diary study to gather mainly quantitative data and a follow-up interview for more qualitative data. During the follow-up interviews we asked questions to get more detailed insights

<sup>3</sup>Due to intellectual property, we are not able to disclose more specific information about the company.

about the data gathered during the diary study. We decided to conduct a diary study and against performing another kind of study, such as observations, since we wanted to find out more about stakeholders' typical software evolution activities without being too intrusive.

#### A. Team Structure and Subjects

All study participants were part of two software development teams of a big engineering company. The company employs tens of thousands of people all over the world, not all in software development. Each of the two teams consisted of one line manager, responsible for a product line and overseeing several projects, two project managers, responsible for a single project and his project team, one to two requirements engineers, six to seven software developers and two to four software testers. These roles are a subset of the fairly extensive stakeholder roles identified by Acuna *et al.* [23] and Yilmaz *et al.* [24]. The size of the teams, 13 and 16, is comparable to other software development teams in industry as reported by [25] and [26]. Except for one of the line managers and some software testers, all members of each team shared an office space.

Initially, we contacted all 29 members from both teams. To get an unbiased sample of stakeholders, we did not preselect any of the team members. 23 of the 29 team members were willing to participate in our study. The other six team members did not participate, mainly due to lack of available time. The 23 participants included two project managers, two line managers, three requirements engineers, five software testers, and eleven software developers. Given the original role distribution on the teams, these participants presented a representative sample of teams and stakeholder roles on the teams involved in the software development process at this company, and also a representative sample of stakeholder roles involved in software development teams in general. From the 23 participants in the diary study, three subjects did not have time to participate in the follow-up interview.

An overview of all participants is presented in Table I. Participants had an average of 12.2 years (ranging from 1.5 to 25 years) of experience in the software engineering domain and an average of 7.9 years (ranging from 1 to 21 years) of experience performing their specific stakeholder role.

#### B. Development Process

Both teams we observed during our study followed the same agile software development process that divides the development into release cycles typically lasting three months. A release cycle is further split into several iterations each lasting several weeks. During each iteration, new features, feature improvements as well as bug fixes are integrated into the software. At the end of each iteration, an internal beta release is created for internal testing purposes. Finally, each iteration has several milestones, each specifying a list of planned change requests to be completed for the milestone. These milestones are reviewed for quality, performance and completeness reasons.

TABLE I  
OVERVIEW OF STUDY PARTICIPANTS (*Role* INDICATING THE PARTICIPANTS' AVERAGE EXPERIENCE IN THEIR ROLE; *SE* THEIR AVERAGE EXPERIENCE IN SOFTWARE DEVELOPMENT; \* INDICATING THAT THE SUBJECT DID NOT PARTICIPATE IN THE FOLLOW-UP INTERVIEW)

Role	Subjects	Experience (in years)	
		Role	SE
<b>Team 1</b>			
Software developer (SD)	D1, D2, D5, D8, D10*	12.8	15.0
Software tester (ST)	T3	5.0	8.0
Requirements engineer (RE)	R1	8.0	15.0
Project manager (PM)	P2*	1.0	12.0
Line manager (LM)	M2	1.5	1.5
<b>Team 2</b>			
Software developer (SD)	D3, D4, D6, D7, D9, D11*	9.8	13.2
Software tester (ST)	T1, T2, T4, T5	4.9	10.8
Requirements engineer (RE)	R2, R3	5.5	16.5
Project manager (PM)	P1	10.0	15.0
Line manager (LM)	M1	2.0	7.0

All code changes during software development are based on change requests. A change request can be a bug report, a feature request or an improvement. These change requests are reported from all stakeholders that are involved in the development of the software project as well as people using the software in the field. At least once a week these requests are analyzed, prioritized and assigned to software developers to resolve them.

#### C. Study Methods

Our study was composed of a diary study with a follow-up interview. We asked the participants to complete an online survey, a diary, at the end of each day over a period of six workdays. The survey questions focused on participants' daily activities, the information they worked with to perform these activities and where they retrieved the information. The online survey contained a total of 10 questions and took participants an average of 14.22 minutes to complete. After completing the diary study, we scheduled and conducted an in-person interview with each participant, except for the three participants that were not available for the interview. The interview took an average of 39 minutes per participant and was designed to gather more detailed insights on a stakeholder's daily activities and the information he works with. We chose this combination of starting with an online diary study to get a general understanding of stakeholders' typical activities, information needs and the frequency they occur. In the second part, we used the answers from the diary study to guide the follow-up interviews for gathering more detailed information as well as answering open questions from and clarifying vague responses to the diary study<sup>4</sup>.

**Diary Study.** For each day of the diary study, we asked the participants three sets of questions after asking them about their role. First, they were asked about the top five activities they spent the most time on during their work day, second,

<sup>4</sup>The survey used for the diary study and the questions guiding the semi-structured interview can be found at: <http://www.ifi.uzh.ch/seal/people/mueller/info-needs>

the participants were asked about the information sources, documents and tools they used for each one of the activities mentioned, and finally, we asked participants to state the stakeholders, in particular their roles, they interacted with during their day.

**Interview Study.** The follow-up interview was conducted as a semi-structured interview. This means, we prepared a set of questions for the interview, but did not strictly follow these questions. Instead, we used them as a general guidance to gain further insights into the answers the participants provided during the diary study. In particular, the questions focused on the participants' activities, their information needs and how the participants acquire, manage and share the needed information artifacts for the performed activities. The interview was recorded and notes were taken manually. Directly after each interview, the protocol was transcribed and augmented with additional comments by the interviewer.

#### D. Data Analysis

Over the course of the study, we collected a large amount of data consisting primarily of answers to the diary study and transcripts of the follow-up interviews. Over all participants, we collected 26 diary entries for software testers, 54 for software developers, 12 for requirements engineers, 8 for project managers and 8 for line managers. Despite providing the participants the flexibility to start the diary study when they had time over the period of a month, not all participants were able to fill in the diary study for the whole 6 work days due to availability. From these entries, we gathered detailed descriptions of 403 activities and the corresponding information artifacts used for these activities.

In order to analyze the collected data from the diary and the interview study, we used a grounded theory approach [27]. First, we followed an open coding approach to develop concepts and categories from the interview transcriptions. We then used axial coding to relate these concepts and categories to each other. Finally, we identified important categories that were brought up by most study participants. Finally, by using selective coding, we systematically related portions of our data to the identified categories. Our findings are presented in Section IV and are discussed in Section V.

#### E. Threats

There are several threats to the validity of our study, mainly to the external and conclusion validity.

**External Validity.** Since we gathered data from a single company, the generalizability of our findings might be limited. We tried to mitigate the risk by observing two different software development teams of that company. Also, since the two observed teams exhibit characteristics in terms of size and roles similar to others reported in the research literature, e.g. [24], [25], we believe that the impact of the single company is not a substantial limitation.

Second, we only investigated the typical activities during a period of six work days which, again, might limit the

generalizability of our findings. However, during the follow-up interview, we asked participants and found out that most of the reported activities are typical and representative for their overall activities.

**Conclusion Validity.** We only used one interviewer to conduct the interviews and transcribe the interview protocol. Additionally, we also only used one coder to categorize our field notes. To minimize the risk of misinterpretation and misunderstanding, we audio recorded the interviews and reviewed our field notes, as well as the categorization with a second investigator.

## IV. RESULTS

Based on the analysis of our qualitative and quantitative data, we identified several observations on stakeholders' activities and on the characteristics of stakeholders' information needs. In the following section, we first provide some background on the software evolution activities before we discuss the key observations with respect to our research questions on information needs for artifacts and their dependencies in a real world context.

### A. Software Evolution Activities

Participants in our study perform a variety of activities during software evolution. From the collected data, we identified a total of 403 activities from which we identified 26 unique activities. Figure 1 provides an overview of these activities and illustrates how much time on average each stakeholder per role spent on a given activity per day. The figure demonstrates that multiple stakeholder roles regularly perform the same activity. In total 19 of the 26 unique activities were performed by more than one stakeholder role. Four activities—attending meetings, project management, communication / interaction, and bug triaging—were mentioned by all five stakeholder roles. While software developers did not mention bug triaging as one of the top 5 activities during the diary study, multiple software developers mentioned bug triaging as a typical activity in their follow-up interview. Most stakeholders agree that it is a very important activity, e.g.,

*“Bug triaging is the most important process [...] I wouldn't know what to do without this process.” (P1)*

Figure 1 also shows that, while a lot of time is spent each day on meetings and project management by all stakeholder roles, there are activities specific to a single or a few stakeholder roles that take up a lot of time. Department management, for instance, an activity specific to a line manager's work, consumes a big chunk of his daily activities and consists of subactivities related to running the department, such as team evaluations, vacation planning, training, promotion, resources allocation and HR acquisition. Another example is reverse engineering, in this case referring to high-level architecture recovery, which was solely performed by requirements engineers. Given the role distribution of our study participants in each team with one line manager and up to six software developers per team (see Table I), only an average of 2% of all participants' time per day was spent on bug triaging, an

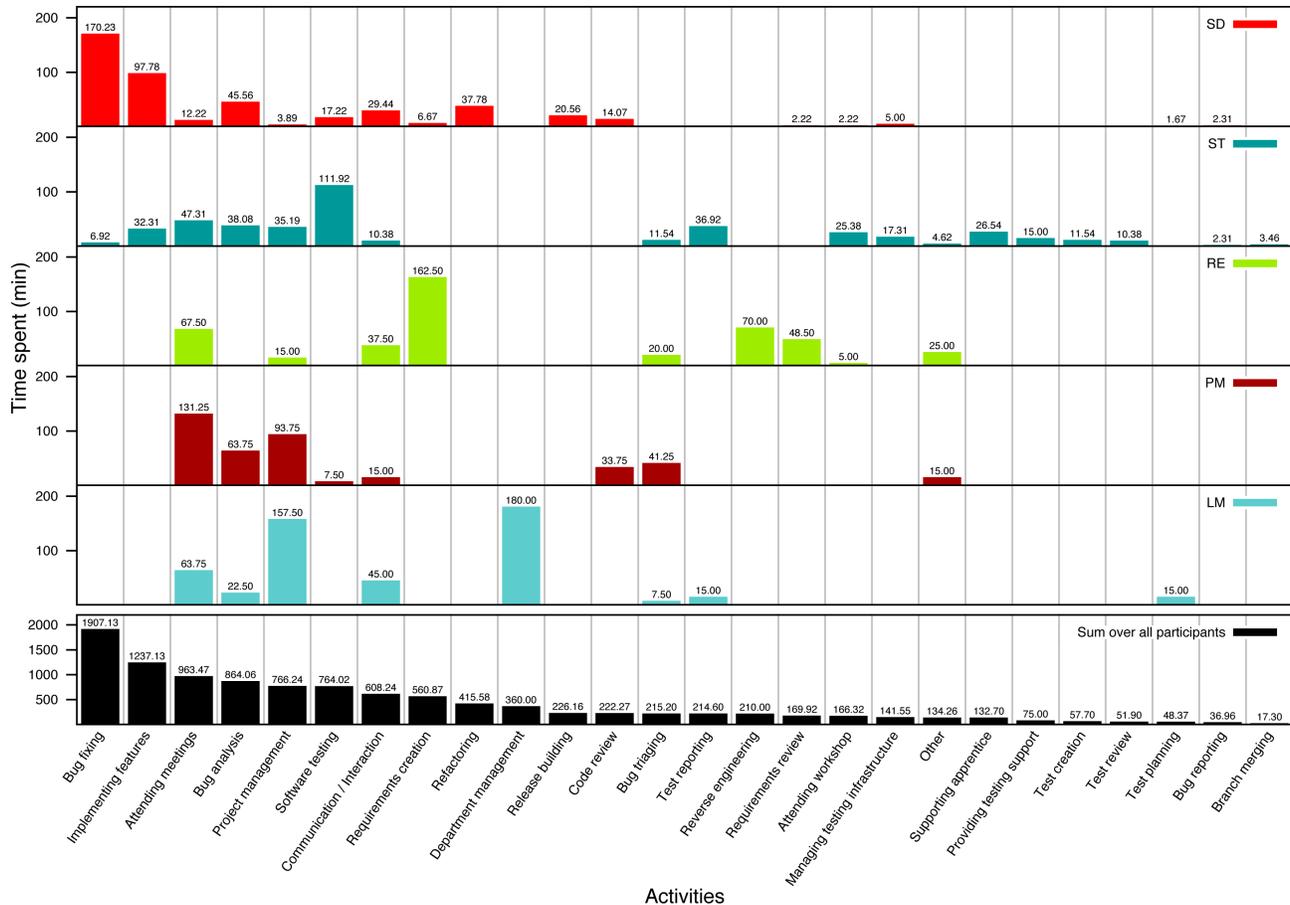


Fig. 1. Overview of the unique activities reported by the various software development stakeholders in our diary study. The height of each bar depicts the average time a stakeholder role spent on an activity per day. The last row depicts the time that all participants of both teams spent in total per day on an activity ( $\Sigma$ ).

activity that is performed by all 5 stakeholder roles, while bug fixing consumed 18.1% and implementing features used up 11.7% of both team's time per day and was only performed by software developers and testers.

The most common activities in a software development team vary with respect to a stakeholder's role. Table II lists the three most common activities per stakeholder role with the average time that was spent on a given activity per day and participant, and the frequency with which the given activity was mentioned in the diary study. The table shows, for example, that bug analysis, an activity that is often related to software developers and testers, is among the top three activities of project managers in our study with them spending more than an hour a day on it. From the interviews, bug analysis was referred to as the activity of assessing a bug report to check if it really denotes a bug and if all important information to fix this bug is available, before it is assigned to a developer to fix it.

TABLE II  
MOST COMMON ACTIVITIES PER STAKEHOLDER ROLE WITH THE AVERAGE TIME SPENT ON IT PER DAY AND PARTICIPANT AND THE TOTAL NUMBER OF TIMES THE ACTIVITY WAS REPORTED.

Stakeholder	Activity	Av. time (min)	Frequency
Software developer	Bug fixing	170.23	48
	Implementing features	97.78	22
	Bug analysis	45.56	18
Software tester	Software testing	111.92	18
	Attending meetings	47.31	12
	Bug analysis	38.08	12
Requirements engineer	Requirements creation	162.50	9
	Reverse engineering	70.00	6
	Attending meetings	67.50	10
Project manager	Attending meetings	131.25	12
	Project management	93.75	7
	Bug analysis	63.75	9
Line manager	Department management	180.00	4
	Project management	157.50	8
	Attending meetings	63.75	5

### B. Information Needs for Software Evolution Activities

To perform their daily activities, stakeholders frequently have to seek, manage and modify large amounts of information

artifacts. These artifacts are acquired by stakeholders using a lot of different tools and accessing various repositories.

**Information Needs Exhibit a Crosscutting Nature Over Information Artifact Kinds.** To successfully perform a single activity, stakeholders require many different kinds of information artifacts. For instance, for bug fixing stakeholders reported to use code artifacts, change sets, planning documents, change requests, code documentation, logs, test cases, code models, configuration files and web sites. During the interviews, one software developer stated:

*“[For bug fixing] I look at the CR and check if the bug is reproducible. Sometimes I use a simulator to test this. [...] Afterwards, using a debugger I can see where the problem is and then I look at the source code to see where I have to change something.” (D3)*

Table III presents the information artifacts that are being used for each of the 15 most common activities illustrated in Figure 1. For all activities, more than one kind of information artifact is being used. On average, 6.9 different information artifacts are being used for each of the top 15 activities shown in Figure 1 and 4.9 information artifacts are being used on average for all 26 activities we have observed during the diary study. In addition to the information artifact kinds shown in Table III, stakeholders also mentioned that they used a lot of different tools and repositories, relied on their personal experience, and also communicated a lot with other stakeholders as they performed activities. In this paper, we focus on the information artifact kinds and fragments, leaving other aspects, such as tools, experience and communication to future work.

The crosscutting information needs vary by stakeholder role for each activity. While, for example, software developers mentioned six different kinds of artifacts that they use for performing bug analysis—change requests, code, requirements, test cases, logs and code models—line managers only used change requests and requirements.

Although less crosscutting, the varying needs of different stakeholder roles can also be seen for bug triaging, an activity that all stakeholder roles perform. While all stakeholder roles reported to use change requests every time they do bug triaging, requirements engineers additionally rely on requirements specifications, code documentation and planning documents, while project managers, for instance, only occasionally use the code base but no documentation or requirements.

Over all activities, the most commonly used artifact kind are change requests. A change request can either be a bug report, a feature request or an improvement. For the top 15 activities, change requests were mentioned in 22.2% of the cases as an important information artifact for performing the given activity. During the interviews, various participants reinforced the importance of change requests by stating how significant a change request is for code-related activities:

*“For each change in the code - even a single line of code - there is a change request.” (D1)*

*“Every implementation work is based on a change request. There is no coding without a change request.” (D4)*

**Information Needs Per Artifact Are Fragmented.** To successfully perform a software evolution activity, not only do stakeholders need information from various kinds of artifacts, the information needs per artifact are fragmented and vary by activity and stakeholder role. In our study we identified this fragmentation of relevant information within information artifacts predominantly for change requests, the most commonly used artifact. Change requests contain various information such as the summary, the description, the creator, the iteration it is planned for or the resolution state. Even though all stakeholder roles use change requests when performing an activity such as bug triaging, there is a difference in the fragments that are considered important in a change request during this activity. For instance, a line manager stated that the creator of a change request is the most important information fragment while bug triaging, while a project manager claimed that the type and the severity are the most important fragments in a change request for the same activity:

*“Especially important is the person who has reported the change request.” (M2)*

*“The type of the change request is important. If it is a defect, the severity is important.” (P1)*

For software testing, a software tester named the resolution as the most important fragment while a software developer stated that the reproduction steps are most important:

*“The resolution is one of the most important piece of information [in a change request]. It is the main mean of communication between developer and tester.” (T4)*

*“Most important are clearly the steps to reproduce a bug. Everything else can be derived from that.” (D3)*

These examples for bug triaging and software testing also illustrate that the relevant information fragments of change requests vary for different activities.

As already pointed out by de Souza *et al.* [19], change requests serve stakeholders as boundary objects [28] to communicate with other team members and to align the coordination needs. A boundary object is generally defined as an artifact that is able to support the communication between different groups by providing a common content that can be accessed and interpreted by all groups [29]. As mentioned above, change requests are the most common and central information kind in all software evolution activities of the two teams we studied. Participants used them to steer the workflow and to pass information to other stakeholder roles. For each stakeholder role though, different fragments of the change request are considered important. For instance, one participant stated that after fixing a bug, the responsible developer enhances the information in the change requests and adds a description of his resolution steps. For testing the bug fix, the software tester then reads these resolution

TABLE III  
INFORMATION ARTIFACT KINDS USED FOR EACH OF THE 15 MOST COMMON ACTIVITIES.

Activity	Code	Requirements	Release notes	Change set	Planning documents	Change request	Code documentation	Test report	Personal notes	Logs	HR-related documents	Test case	Code review	Code model	Configuration file	Website	# of artifacts
Bug fixing	■			■			■			■		■		■			10
Implementing features	■			■			■										11
Attending meetings		■						■									6
Bug analysis	■						■			■		■		■			10
Project management					■						■						8
Software testing	■		■				■					■		■			8
Communication/Interaction											■				■		11
Requirements creation									■							■	7
Refactoring	■						■					■					4
Department management											■						2
Release building	■		■				■							■			6
Code review	■	■		■			■						■				5
Bug triaging	■	■		■			■						■				5
Test reporting		■					■	■		■		■					6
Reverse engineering	■	■					■								■		4

steps to understand what to test and how to adapt test cases if necessary.

**Crosscutting and Fragmented Information Needs Vary by Activity and Role.** The collected data shows that many information needs in software development exhibit a crosscutting nature. They cut across stakeholder roles, activities, artifacts and even fragments of artifacts. For each activity performed during software development, a multitude of different information artifacts is being used. While different stakeholder roles perform some of the same activities on a daily basis, different roles use different information artifacts to perform the same activity. The crosscutting nature of the information needs is even visible for a single artifact, since different roles use different fragments of the same artifact for the same activity. This is particularly evident for change requests, the most commonly used artifact in the observed software development teams, for which certain fields are only relevant to certain stakeholder roles.

### C. Information Artifact Inter-Dependencies

A lot of dependencies exist between artifacts, in particular in the development and evolution of software. There are dependencies between a requirement and its implementation in the code or the test cases that cover the requirement, structural relations between different code artifacts and the link between a change request and the change sets to resolve it, to name just a few. To successfully perform the daily activities, these relations are an important part of stakeholder's information needs to identify and understand the relevant

fragments of artifacts. Research often talks about these interdependencies in terms of traceability links that are useful for instance for requirements validation, impact analysis and program comprehension (e.g., [6], [5]).

**Inter-Dependencies Relevance Varies by Activity and Stakeholder Role.** Participants in our interview study talked a lot about the dependencies between artifacts and the communication efforts to identify them. In particular, the dependencies between requirements or versions of a requirement and code or change requests were mentioned continuously:

*“Developers most frequently ask where to find the specifications for a specific component.” (R1)*

*“If there is no requirements document attached to the change request, it is often necessary to ask a requirements engineer to get the appropriate document.” (D6)*

Overall, interdependencies were mentioned as an important part to successfully perform software evolution activities with a lot of time being spend on recovering them.

Similarly to the crosscutting nature of information needs, the relevance of dependencies between artifacts also changes with stakeholder role and activity. For the same activity, different stakeholders reported different inter-dependencies as most relevant. For instance, for the analysis of a bug reported by a field worker, a line manager (M2) stated that he first examines dependencies between the reported bug and requirements by talking to the requirements engineers and investigating whether it is a bug or a feature. A software tester (T5) said that the reproduction of the bug is most important

and he therefore first examines which component is affected and then talks to the software developer who wrote the code. Finally, a software developer (D3) mentioned that he first analyses the configuration file that was in use when the bug occurred to see if there is a problem in there before examining the code.

In other cases, the same interdependency is required for different activities. Participants in our study repeatedly talked about the links between a code fragment and the responsible software developer for various activities. While a project manager (P1) stated to require this link in the process of bug triaging, a software tester (T5) said he uses the link for bug analysis to find out more information on how to test the bug, and a software developer (D4) stated to use this link when he reviews code using a static analysis tool and wants to talk to the responsible developer about the problems he found.

**Links between Information Artifacts are Often Missing.** While some information artifacts can be explicitly linked, such as test cases or requirements to change requests, we observed that these links are often out of date, unreachable or not available at all:

*“Requirements documents are difficult to find. There are a lot of different places to store a requirements document. It is not always linked to a change request and it is not even always clear if there is a requirements document and if there is, where it can be found.” (D8)*

*“I guess that in only 6% of all cases there is really a link between the test case and the requirements.” (R2)*

In the interviews, 14 participants (70%) explicitly mentioned the lack of links between artifacts in their daily activities and the problems the missing links cause.

**Links are Missing for Many Reasons.** Participants mentioned a variety of reasons for the lack of explicit links. Several stated that there are a lot of different systems and repositories and it is not always clear where to best store or find artifacts and links, *e.g.*,

*“There is a lot of documentation available, but it is not widely known where to find it.” (D9)*

The rapid evolution and high frequency of change in the artifacts were also mentioned to make it difficult for keeping links up to date, *e.g.*,

*“Developers do not know which requirements documents they have to update if they change something in the code, because they can’t find the associated requirements documents.” (P1)*

In addition, a lot of the software systems that participants are working on are legacy systems. When they first started working on these systems, requirements specifications, documentation, as well as other information artifacts were not fully available and the main focus was on the development of the code. Therefore, links between various information artifacts

were often not established. Recovering and establishing these links today is not a priority, in particular since it requires a lot of time and effort while the systems continue to rapidly evolve.

**Missing Links Lead to Additional and Repeated Communication Effort.** The missing links between information artifacts and fragments lead to several problems. Multiple stakeholders have problems to identify the dependent information artifacts or fragments themselves and therefore communicate a lot with other stakeholders. Participants mentioned the time-consuming communication effort in particular for requirements:

*“If there is no requirements document attached to the change request, it is often necessary to ask a requirements engineer to get the appropriate document.” (D6)*

*“4-5 iterations [with a requirements engineer] are necessary until every issue is clarified and until I can start to implement anything.” (D2)*

Since links between artifacts are not captured explicitly, participants also have to repeatedly explain the same interdependencies to other stakeholders:

*“The same clarification requests have to be answered again and again.” (R3)*

*“I have to ask the requirements engineer again and again for clarification [...] Some requirements documents have dependencies on other documents. They overwrite information in other documents. It is very difficult to find the appropriate and complete description.” (D2)*

In our data analysis, we observed that almost all stakeholder roles spend a lot of time on link retrieval. These observations also explain why there is so much communication and interaction reported from various stakeholder roles in Figure 1.

**Wikis are Used to Compensate for the Missing Links.** To overcome the often time-consuming interaction with other stakeholders, participants started to use wikis. In the organization we observed, people started using wikis mostly a few months and up to a year before our interviews took place, even though the projects have been existing for several years. In general, the wiki provides a single-entry point to the participants where a lot of the missing links are kept:

*“The wiki particularly contains links to already existing documents. It provides a single-entry point for a lot of documentation and information. For example I add a link to a software release. Then the testers use this link to get the newest software versions for their simulators.” (D7)*

*“In the wiki links to other important internal documents, *e.g.* requirements specifications, are stored.” (D3)*

Wikis are used to manage a lot of different information artifacts that are used in a lot of different activities. However, these collection of links and pointers to information artifacts and fragments are not very structured. The wiki is largely

a collection of information, such as instructions on how to install or use tools, instructions on how to set up projects and environments, release notes for each software version linking to the list of change requests for each release and the known bugs fixed in the release, as well as outstanding bugs, and links to technical documentation.

During the follow-up interviews, we observed that 11 out of 20 interviewed stakeholders use the wiki on a regular basis. We also observed that not all stakeholders think that the wiki is a good solution for storing the links between various information artifacts, e.g.,

*“We in the requirements engineering team do know that developers use a wiki, but we do not think that this is useful.”* (R2)

## V. DISCUSSION

The variety and crosscutting nature of information needs in software evolution activities in combination with the large amounts of continuously changing software project information, make it challenging for stakeholders to satisfy their information needs in a timely manner. In our study in the workspace of stakeholders, we observed several participants jumping back and forth between a multitude of tools, editors and repositories for a single activity, since each one of them only presented one kind of artifact. In addition, participants reported on spending a lot of time on communicating with other stakeholders to identify dependencies between artifacts as well as discussing the artifacts.

**Need for Aggregating Information Artifacts.** The need for stakeholders to switch between tools and gather a variety of crosscutting information fragments to perform a single activity, creates cognitive burden and requires time and effort. Rather than providing one view per artifact, tool support is needed to aggregate and synthesize the multitude of artifacts that stakeholders work with these days. This was also partially mentioned by participants:

*“There is a need to unify all the internal repositories, documents, processes, etc.”* (M2)

*“The main problem is that there is no search function that covers all the databases.”* (R3)

However, we hypothesize that it is not only important to provide access to multiple kinds of artifacts as asked for by the participants, but that new techniques are provided that aggregate various kinds of artifacts in a single view or presentation.

**Tailoring Views Based on Role and Activity Context.** In our study we found that the information fragments used for a given activity vary depending on the activity and stakeholder role context. By providing support for tailoring the aggregated information to the activity and role, we might be able to cut down large portions of information, such as irrelevant fields in a change request. This in turn would allow us to focus

stakeholders’ attention to artifacts that are really relevant for their current activity.

**Lack of Socio-Technical Congruence.** The high communication load between stakeholders for their information needs in combination with the missing links between artifacts points to a lack of socio-technical congruence. Socio-technical congruence refers to the idea of “fit” between the social and the technical dimension in software development [30], [31]. It has previously been shown that teams are more efficient, when the technical links and the communication structure is congruent [30], [32].

One approach to overcome the problem of missing links is the usage of wikis to manage an unstructured collection of links. Other companies might not use a wiki but other techniques, strategies and tools to try to overcome the missing linkage between information artifacts, such as dashboards [33], or explicit iteration plans.

We also observed participants pointing out that the problems of missing links are increasing with the increasing size and distribution of teams:

*“The requirements engineer no longer hears what we developers are talking about and therefore he can no longer intervene if the developers plan to implement something in a way that is not correct from a requirements perspective.”* (D8)

**Making Links First Class Entities.** Current repositories and tools that maintain information artifacts store links between information artifacts most often as by-products of the information artifact itself. Furthermore, there are often multiple places in these artifact repositories to store these links. This leads to links being neglected, not updated or not available and makes it difficult for stakeholders to identify and find the relevant links.

We suggest to make artifact links first class entities so that it is easier to search for, find and update links between artifacts. We hypothesize that this will result in more explicit links and in turn a higher socio-technical congruence and a reduced communication effort between stakeholders. Since humans are good at recalling associations [34], links as first class entities can also allow stakeholders to more efficiently find relevant information by querying for associations rather than just artifacts, as related research has already shown [35].

## VI. CONCLUSION

In this paper, we presented the results of an empirical study investigating stakeholders’ information needs for software evolution activities. The study consisted of a six day diary study and a follow-up interview and was conducted with 23 stakeholders of two professional software development teams. The focus of our study was on providing evidence for the information needs of multiple stakeholder roles and how they manifest themselves in the context of a real world company.

From the analysis of the collected quantitative and qualitative data, we found that information needs exhibit a cross-

cutting and fragmented nature. Thus, to successfully perform their daily activities, stakeholders require multiple different kinds of artifacts or fragments of these artifacts, and these required information fragments vary by stakeholder role and activity. Furthermore, we observed a lack of socio-technical congruence: dependencies between information artifacts are often not explicitly captured or out of date and require stakeholders to repeatedly put additional effort into communicating with other stakeholders to understand and recover these missing links.

We suggest that approaches to support multiple stakeholder roles with their information needs should provide means to aggregate multiple kinds of artifacts in a single view that can be tailored by stakeholder role and activity and that dependencies between information artifacts should be made first class entities. Future work will look into extending our study to further teams and companies and examine generalizability, as well as we plan on developing concrete techniques to support the crosscutting information needs as suggested.

#### REFERENCES

- [1] D. Perry, N. Staudenmayer, and L. Votta, "People, organizations, and process improvement," *IEEE Software*, vol. 11, pp. 36–45, July 1994.
- [2] R. E. Kraut and L. A. Streeter, "Coordination in software development," *Commun. ACM*, vol. 38, no. 3, pp. 69–81, 1995.
- [3] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.
- [4] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proc. of the 32nd ICSE '10*, pp. 125–134, ACM, 2010.
- [5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.
- [6] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796–810, 2003.
- [7] W. L. Johnson and A. Erdem, "Interactive explanation of software systems," in *Proc. of the 10th KBSE '05*, pp. 155–164, IEEE Computer Society, 1995.
- [8] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st ICSE '09*, pp. 298–308, IEEE Computer Society, 2009.
- [9] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," in *Proceedings of the CSCW '10*, (New York, NY, USA), pp. 301–310, ACM, 2010.
- [10] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.
- [11] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. of the 29th ICSE '07*, pp. 344–353, IEEE Computer Society, 2007.
- [12] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proc. of the 32nd ICSE '10*, pp. 185–194, ACM, 2010.
- [13] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proc. of the 32nd ICSE '10*, pp. 175–184, ACM, 2010.
- [14] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?," in *Proc. of the ICSE '12*, pp. 255–265, IEEE Press, 2012.
- [15] C. Seaman, "The information gathering strategies of software maintainers," in *Proc of the ICSM '02*, pp. 141–149, 2002.
- [16] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proc. of the 28th ICSE '06*, pp. 492–501, ACM, 2006.
- [17] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*, CASCON '10, pp. 174–188, IBM Corporation, 2010.
- [18] A. Schröter, J. Aranda, D. Damian, and I. Kwan, "To talk or not to talk: factors that influence communication around changesets," in *Proc. of the ACM CSCW '12*, pp. 1317–1326, ACM, 2012.
- [19] C. R. B. d. Souza, D. Redmiles, G. Mark, J. Penix, and M. Sierhuis, "Management of interdependencies in collaborative software development," in *Proc. of the ISESE '03*, pp. 294–303, IEEE Computer Society, 2003.
- [20] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?," in *Proc. of the 33rd ICSE '11*, pp. 804–807, ACM, 2011.
- [21] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. of the 25th ICSE '03*, pp. 125–135, IEEE Computer Society, 2003.
- [22] X. Chen and J. Grundy, "Improving automated documentation to code traceability by combining retrieval techniques," in *Proc. of the 26th IEEE/ACM ASE '11*, (Washington, DC, USA), pp. 223–232, IEEE Computer Society, 2011.
- [23] S. Acuna, N. Juristo, and A. Moreno, "Emphasizing human capabilities in software development," *IEEE Software*, vol. 23, no. 2, pp. 94 – 101, 2006.
- [24] M. Yilmaz, R. O'Connor, and P. Clarke, "A systematic approach to the comparison of roles in the software development processes," in *Proc. of the 12th SPICE '12*, pp. 198–209, Springer Berlin Heidelberg, 2012.
- [25] D. Rodríguez, M. A. Sicilia, E. García, and R. Harrison, "Empirical findings on team size and productivity in software development," *Journal of Systems and Software*, vol. 85, no. 3, pp. 562–570, 2012.
- [26] P. C. Pendharkar and J. A. Rodger, "An empirical study of the impact of team size on software development effort," *Information Technology and Management*, vol. 8, no. 4, pp. 253–262, 2007.
- [27] P. Y. Martin and B. A. Turner, "Grounded theory and organizational research," *The Journal of Applied Behavioral Science*, 1986.
- [28] S. L. Star and J. R. Griesemer, "Institutional ecology, 'translations' and boundary objects: amateurs and professionals in berkeley's museum of vertebrate zoology, 1907-39," *Social Studies of Science*, vol. 19, pp. 387–420, 1989.
- [29] C. Kimble, C. Grenier, and K. Goglio-Primard, "Innovation and knowledge sharing across professional boundaries: Political interplay between boundary objects and brokers," *International Journal of Information Management*, vol. 30, no. 5, pp. 437 – 444, 2010.
- [30] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in *Proc. of the 2nd ACM-IEEE ESEM '08*, pp. 2–11, ACM, 2008.
- [31] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proc. of the 20th Anniversary CSCW '06*, pp. 353–362, ACM, 2006.
- [32] F. Bolici, J. Howison, and K. Crowston, "Coordination without discussion? socio-technical congruence and stigmergy in free and open source software projects," in *2nd International Workshop on Socio-Technical Congruence, ICSE '09*, 2009.
- [33] C. Treude and M.-A. Storey, "Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds," in *Proc. of the 32nd ICSE '10*, pp. 365–374, ACM, 2010.
- [34] E. Tulving and D. M. Thomson, "Encoding specificity and retrieval processes in episodic memory," *Psychological Review*, vol. Vol. 80, no. 5, pp. 359–380, 1973.
- [35] D. H. Chau, B. Myers, and A. Faulring, "What to do when search fails: finding information by association," in *Proc. of the SIGCHI '08*, pp. 999–1008, ACM, 2008.

## Improving Feature Location by Enhancing Source Code with Stereotypes

Nouh Alhindawi<sup>1</sup>, Natalia Dragan<sup>1</sup>, Michael L. Collard<sup>2</sup>, Jonathan I. Maletic<sup>1</sup>

<sup>1</sup>Department of Computer Science

Kent State University

Kent, Ohio, USA

{nalhinda, ndragan, jmaletic}@kent.edu

<sup>2</sup>Department of Computer Science

University of Akron

Akron, Ohio, USA

collard@uakron.edu

**Abstract**—A novel approach to improve feature location by enhancing the corpus (i.e., source code) with static information is presented. An information retrieval method, namely Latent Semantic Indexing (LSI), is used for feature location. Adding stereotype information to each method/function enhances the corpus. Stereotypes are terms that describe the abstract role of a method, for example get, set, and predicate are well-known method stereotypes. Each method in the system is automatically stereotyped via a static-analysis approach. Experimental comparisons of using LSI for feature location with, and without, stereotype information are conducted on a set of open-source systems. The results show that the added information improves the recall and precision in the context of feature location. Moreover, the use of stereotype information decreases the total effort that a developer would need to expend to locate relevant methods of the feature.

**Keywords:** software maintenance, information retrieval, feature location, method stereotypes, program comprehension.

### I. INTRODUCTION

When correcting a fault, adding a new feature, or adapting a system to conform to a new platform or API, software engineers must first find the relevant parts of the code that correspond to the particular change. This is termed feature or concept location [1, 2]. Feature location involves searching, exploring, reading, and understanding the source code. These types of comprehension activities make up a major portion of the costs in the evolution of modern software systems [3, 4].

A number of different techniques to support feature location have been suggested and involve everything from simple regular-expression matching, dynamic and static program analysis, and information-retrieval techniques. Regular-expression matching is often used by programmers but returns far too many false positives and has no ability to rank the results. Static and dynamic methods often suffer from the same types of problems [2, 5] (too many false positives) or require very accurate test cases for the feature, which may not be available.

Over the past few years' information-retrieval methods have been used for feature location with encouraging results [2, 6-12]. Information retrieval (IR) methods move far beyond keyword matching and regular expressions and use advanced probability and information theory to derive relationships between documents based on the vocabulary and occurrences of words in each document. This is attractive because queries (to find particular) features can be made in the language of the documents (i.e., programming language terms, identifiers, and natural language of comments). There are also means to rank

the results from a query, much like the manner that web search engines present results.

While the use of IR methods has been successful for feature location, there is room for improvement. In particular, false positives are an issue and the most relevant documents are not always ranked highly. This presents problems for software engineers using tools for feature location. Adoption is a problem because results are not good enough and searching through a long list of possible relevant documents is costly and time consuming.

To address this problem a number of researchers have combined various static and dynamic analysis techniques with the results from IR methods. For example, Formal Concept Analysis (FCA) has been used to help rank the results produced by IR methods [10]. The approach taken here addresses the problem of improving the results of IR methods in a novel and very different manner.

We use IR methods in a standard manner [8, 10, 13] for the problem of feature location. However, before applying the IR method, we enhance the corpus (i.e., source code) with new information. This new information is derived automatically from the source code via static program analysis. Specifically, we re-document the source code (i.e., add new terms) with stereotype [14] information for each method/function in the system. After this has been completed we use the IR method to run queries for feature location. This type of up-front enhancement of a corpus to improve results has not been investigated previously.

Adding these new terms is a form of supervision added on top of an unsupervised method (i.e., LSI). Apriori knowledge is often used to direct and supervise machine-learning and information-retrieval approaches [15]. Here, we derive this information from the corpus itself. Others have used similar approaches based on ontological information [16] and inferred semantics from term distribution [17]. From an information theoretic standpoint the addition of (relevant) information will improve the results of an information-retrieval technique [18, 19]. That is, more information is better, so long as you don't add noise.

Our hypothesis is that the stereotype information is relevant and will improve the results in the context of feature location. Our experimental study supports this hypothesis. The results demonstrate a definite improvement in locating relevant methods pertaining to the feature being queried when stereotype information is included.

We chose to use stereotype information for a number of reasons. Stereotypes describe the abstract behavior and role of a method within a class. We felt that this was relevant

information and our previous work investigating the automatic detection of stereotypes [14, 20], bears evidence that they support program comprehension. Moreover, we found that distributions of method stereotypes can be used to derive class stereotypes. This evidence gave support to enhancing the information within the source code. Lastly, stereotype information is new information that did not previously exist in the source code (i.e., new vocabulary).

The remainder of this paper is organized as follows. First in Section II we present related work on using information retrieval methods for feature location. Section III presents a taxonomy of method stereotypes and a brief description of how they are computed and added to the source code. The approach of using LSI and stereotypes together is described in Section IV. An experiment comparing the results of using LSI with the additional stereotype information is given in Section V. This is followed by discussion of the results in Section VI and threats to validity in Section VII.

## II. RELATED WORK

An overview of existing static feature location approaches is reviewed along with related work on feature location using LSI.

### A. Previous Work on Feature Location

Historically, developers used pattern-matching techniques like *grep* to locate features in the source code. Using pattern-matching techniques is simple; it performs an investigation through pattern matching on character strings. Nevertheless, it requires a lot of experience from the developer. If the technique failed, more advanced tools were required, especially when the system is large [2, 4, 6, 7, 21].

Biggerstaff et al. [1] referred to concept location as the concept location assignment problem. Their work was a preliminary point for many efforts to facilitate and develop the process of concept location. Call graphs, program clustering graphs, etc. are used in their approach.

Chen and Rajlich [22] presented an approach based on using an Abstract System Dependencies Graph (ASDG). The ASDG can lead, guide, and assist the user in the process of searching for a particular feature.

Wilde [23] developed the software-reconnaissance method, which utilizes dynamic information to locate features in existing systems. Wong et al. [24] analyzed the execution slices of test cases to the same end. Eisenbarth et al. [5] used dynamic information gathered from scenarios of invoking features in a system to locate the features in source code. Tools that deal with feature location are either static or dynamic. Overlap between features cannot be distinguished using dynamic analysis, while static analyses do not often identify units contributing to a particular execution scenario [2, 4].

Revelle and Poshyvanyk [25] presented an investigative study of ten feature-location techniques that use different combinations of textual, dynamic, and static analyses. A survey of feature location techniques is presented in [2].

### B. Previous Work on Feature Location using IR

Recently, IR methods have been used successfully and effectively for feature location [2, 6-12]. For more details, we refer the readers to the survey by Binkley and Lawrie [4] on information-retrieval applications in software maintenance and evolution.

Marcus and Maletic [11] were the first researchers to use LSI for application to software engineering. They obtained similarity measures between source-code components in order to cluster and classify these components. They also defined a number of comprehension metrics. These metrics use the profile produced by the application of LSI to a matrix of source code. In [6], Marcus et al. linked LSI to concept location, where they used LSI to map concepts expressed in change requests described using natural language to the relevant source-code components.

Many efforts have been made to improve the use of LSI in feature location by adding or integrating meaningful information to the whole process of feature location [8, 10]. For example, in [8], the authors combined LSI with user-execution scenarios to improve the accuracy of feature location.

Poshyvanyk et al. [13] proposed a Visual Studio plugin called IRiSS which, based on the existing “find” feature, uses LSI to search projects using natural-language queries. In [9], Poshyvanyk et al. combined static and dynamic techniques they had previously developed. Their goal was to use both certain and uncertain knowledge extracted with both static and dynamic analyses, filter it by probabilistic and information-retrieval techniques, and in this way to identify features in source code. Moreover, Poshyvanyk et al. [7], in order to improve the accuracy of feature location process, proposed a technique that combines information from an execution trace and from the comments and identifiers in the source code. M. Revelle et al. [12] applied advanced web mining algorithms (*Hyperlinked-Induced Topic Search (HITS)* and *PageRank*) to analyze execution information during feature location. Their approach improved the effectiveness of existing approaches by as much as 62%. The ability of LSI in providing a straightforward language-independent method that recognizes relationships between documents is shown in SNIAFL [26].

The dimensions of singular value decomposition when using LSI have been studied. The range of 200 to 300 dimensions has been proposed as a “golden standard” [6]. In [9], Poshyvanyk et al. looked at varying the number of dimensions when using LSI. Their findings concluded that any larger factor could improve the results but would generate too large a search space.

Generally, the current approaches either use IR methods alone or in combination with other techniques, such as [7, 9]. There is a need for improvement in recall and precision of feature location. None of these approaches augment the source code with new information. In our approach, the source code is augmented with method stereotypes, which is described next.

## III. METHOD STEREOTYPES

Stereotypes are a concise abstraction of a method’s role and responsibility in a class and system [14]. They are widely used

to informally describe methods. Stereotypes for classes are also used in the same manner to describe their role and responsibility within a system’s design. UML provides mechanisms for documenting class stereotypes. Manually documenting method stereotypes is relatively easy for a small number of classes and methods however it is quite costly to do so for an entire system.

A taxonomy of method stereotypes (see Table 1) and technique to automatically reverse engineer stereotypes for existing methods was presented by Dragan et al. in [14]. This work was further extended to support the automatic identification of class stereotypes in [20]. That work describes an approach to automatically identify method stereotypes that we use in this research. We refer the readers to those works for complete details on computing method stereotypes; however we present the main points here.

TABLE 1 TAXONOMY OF METHOD STEREOTYPES AS GIVEN IN [20]. THE TAXONOMY IS MAINLY FOCUSED ON THE C++ PROGRAMMING LANGUAGE. METHODS MAY BE LABELED WITH ONE OR MORE STEREOTYPES.

Stereotype Category	Stereotype	Description
<b>Structural Accessor</b>	get	Returns a data member.
	predicate	Returns Boolean value which is not a data member.
	property	Returns information about data members.
	void-accessor	Returns information through a parameter.
<b>Structural Mutator</b>	set	Sets a data member.
	command	Performs a complex change to the object’s state ( <i>this</i> ).
	non-void-command	
<b>Creational</b>	constructor, copy-const, destructor, factory	Creates and/or destroys objects.
<b>Collaborational</b>	collaborator	Works with objects (parameter, local variable and return object).
	controller	Changes an external object’s state ( <i>not this</i> ).
<b>Degenerate</b>	incidental	Does not read/change the object’s state.
	empty	Has no statements.

The taxonomy of method stereotypes given in Table 1 unifies and extends previous literature on stereotypes and addresses a number of gaps and deficiencies that were present. The taxonomy was developed primarily for C++ but many aspects of it can be applied to other programming languages. Based on this taxonomy, static program analysis is used to determine the stereotype for each method in an existing system. The taxonomy is organized by the main role of a method while emphasizing the creational, structural, and collaborational aspects with respect to a class’s design. Structural methods provide and support the structure of the class. For example, accessors read an object’s state, while mutators change it. Creational methods create or destroy objects of the class.

Collaborational methods characterize the communication between objects and how objects are controlled in the system. Degenerate are methods where the structural or collaborational stereotypes are limited. The naming is based on the mathematical term for a case for which a stereotype cannot be any simpler. Also, a method may have more than one stereotype. A tool [14], *StereoCode*, was developed that analyzes and re-documents C++ source code with the stereotype information for each method. Re-documenting the source code is based on srcML (Source Code Markup Language) [27], an XML representation of source code that supports easy static analysis of the code.

In order to provide the method-stereotype identification, we translate the source code into srcML, and then, *StereoCode* takes over by leveraging XPath, an XML standard for addressing locations in XML. For details about the rules for identifying each method stereotype, we refer the readers to [14]. Adding the comments (annotations) to source code is quite efficient in the context of srcML. The XPath query gives us a location of the method and we can then do a simple transformation within the srcML document to add the necessary comments. This process is fully automated and very efficient/scalable. Running *StereoCode* on two systems used in the evaluation takes less than a minute each.

```

class DataSource :public Observable
{
...
public:
  /** @stereotype get */
  const string& getName() const;

  /** @stereotype predicate */
  bool isValidLabel(const string& label) const;

  /** @stereotype command */
  virtual void reserve(int count );
...
};

```

Figure 1. A code snippet of the HippoDraw C++ Class DataSource after re-documenting with the method stereotypes.

Methods can be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have secondary stereotypes from the collaborational and degenerate categories. For example, a two-stereotype method *get-collaborator* returns a data member that is an object or uses an object as a parameter or a local variable.

Figure 1 presents an example of stereotype labeling for part of the class DataSource from the HippoDraw open-source application (one of the systems used in the experiment). The class DataSource supplies one or more arrays of data.

The evaluation of the taxonomy and approach demonstrated two things. First, the method-stereotype taxonomy covered a very large percentage of the methods studied. That is, almost all methods can be labeled by the classification scheme. Second, the tool re-documents systems according to the taxonomy with very high accuracy in comparison to human evaluation.

#### IV. LSI+STEREOTYPES FOR FEATURE LOCATION

We now describe the approach taken for feature location. The same approach as taken in [6] is used here. The IR method, Latent Semantic Indexing (LSI) [4, 28] is the basis of the approach. We term our approach LSI+S (LSI plus stereotypes) to differentiate it using with LSI without stereotypes.

We start with the source code for a software system. As described in the previous section, our *StereoCode* tool is applied to automatically determine the stereotype of each method and re-document it with a comment stating its stereotype. Next preprocessing is done to the resultant re-documented source code to convert it into input for LSI. This is termed a corpus (we will describe how the corpus is generated below in section B).

At this point LSI is applied to the corpus. A co-occurrence matrix of vocabulary  $\times$  documents is computed and Singular Valued Decomposition (SVD) [29] is applied to reduce the dimensionality of this matrix by exploiting the co-occurrence of related terms. The result is a subspace that can be queried against to locate documents most similar to the query phrase. Ranked documents will be retrieved based on their similarities to the query. The user then inspects the results. More details about these steps are covered separately on the following subsections. We now give a brief overview of LSI and describe the details of how we set up the feature location process.

##### A. Latent Semantic Indexing

LSI is a corpus-based statistical technique which is used for inducing and representing characteristics of the meanings of words and passages (of natural language) reflective in their usage [6, 28].

Among code-based feature-location techniques, LSI is considered one of the better techniques capable of recognizing terms in source code that are relevant to a user query [4]. Moreover, LSI is language independent and using it to preprocess and query the source code is more efficient than using a pattern-matching technique, especially with its capability in dealing with synonymy and polysemy. It is also simpler than using graph-based techniques [4, 6].

The initial step of the IR process is to build the corpus for the software system. The corpus consists of a set of documents. In this work (in most all feature location works), documents in the corpus are methods or functions. These documents include the text of each method including all the identifier names, comments, etc.

##### B. Corpus Creation

Constructing the corpus is an important step for feature location using LSI. Five actions are taken to create the corpus: 1) Extraction of identifiers, and comments; 2) Extraction of method stereotypes; 3) Identifier (term) separations; 4) Removing stop words; 5) Division into documents (method level).

A well-built corpus helps in locating the relevant methods (effectiveness measure). As mentioned in [12], not all feature-location techniques can locate all feature-relevant methods, and one of the reasons behind that is the preprocessing steps taken

by each technique when building or creating the corpus. We developed an efficient corpus builder in C++ to extract these important elements from source code using srcML [27]. It takes less than 30 seconds to build both the corpus (corpora for the two systems) with stereotypes and the corpus without stereotypes.

Names such as identifiers, function name, etc. are split according to the standard separators [25]. An underscore, ‘\_’, is used as a separator to split identifiers that contain more than one word, e.g., *feature\_location* after splitting becomes *feature\_location*, and *feature\_location*. Camel casing is also used as a separator, e.g., *FeatureLocation* is split into *Feature*, *Location*, and *FeatureLocation*, and *FEATURELocation* is split into *FEATURE*, *Location*, and *FEATURELocation*.

The final step of preprocessing is partitioning the code into documents. Each function is considered to be a separate document (i.e., level of granularity). Typically, a document in the corpus can be a file of source code or a program entity such as a class, function, interface, etc. When the preprocessing is completed the software system is represented by a set of documents,  $S = \{d_1, d_2, \dots, d_n\}$ , where  $d_i$  is any contiguous set of lines of source code and/or text. Each document  $d_i$  contains the function name, identifiers that the function uses, internal comments, string literals, and the stereotype annotation (if it has been re-documented) for each the function. After these steps, the corpus is constructed.

##### C. Indexing

The next step is to index the corpus using LSI. After creating the LSI space (using SVD), each document  $d_i$  in system  $S$  will have a corresponding vector  $v_i$ . Reduction of dimensionality is done in this step and reflects the most important latent aspects of the corpus. The dimension of the vector is a parameter of the algorithm. It is normally between 100 and 300 [6]. The typical manner to choose this value is to run experiments with different values (e.g., 100, 200, 300) and select the one that gives the best results with respect to evaluation measures as shown later [6].

Measuring the similarities between any two documents  $sim(d_i, d_j)$ , can be done by measuring the similarities between their correspondents vectors. Here cosine similarities are used. By studying and analyzing these similarities, we can identify the semantic information regarding source-code fragments, and the relations connecting them.

##### D. Formulating and Ranking Queries

The user formulates a query by using natural language to describe a change request in the same manner as [8]. A user query ( $q$ ) is converted into a document of LSI space ( $d_q$ ) and vector ( $v_q$ ) for it is constructed. Based on the similarity measure between  $v_q$  and all documents in the corpus, the most relevant documents to  $v_q$  are retrieved as a ranked list  $\{P_1, P_2, \dots, P_n\}$ .

Once LSI retrieves the relevant documents ranked by their similarities to the user query, then the user has the task of inspecting and investigating these documents to decide which of them are actually relevant to the query. The first ranked document ( $P_1$ ) will be investigated first and then ( $P_2$ ) and so on.

The user decides when to stop investigating. If the user discovers a part of the feature, then the intended feature is located successfully. Otherwise, the user can reformulate the query taking into account these results.

## V. EXPERIMENTAL STUDY

A feature-identification study, over two open-source software systems is conducted to evaluate and compare the results of LSI and LSI+S. The study is designed based on recommendations from [30]. Both techniques, LSI and LSI+S, are applied independently and then the results compared. The only difference between the techniques is the inclusion of the stereotype information in LSI+S. Otherwise, the parameters used and the construction of the corpus is exactly the same. We chose one large and one medium-size open-source system to demonstrate the scalability/practicality of the proposed approach.

### A. Design and Objectives of the Experimental Study

The first system is HippoDraw, an open-source application written in C++ that provides a data-analysis environment. It includes data-analysis processing and visualization with an application GUI interface, and can be used as a stand-alone application or as a python extension module. HippoDraw source code is well written and follows a pretty consistent object-oriented style. Its library consists of approximately 50K LOC and over 300 classes. HippoDraw 1.21.3 release is used in our study since it's well documented.

The second system used is the open source cross-platform application and UI framework Qt. It has extensive international support, as developers from Nokia, Digia, and other companies are involved in Qt's development. Qt is mainly written in C++ but has some language extensions with a special code generator (called the Meta Object Compiler) and special macros. It is cross platform for Windows, Linux, or Mac, and all of its editions support a wide range of compilers (e.g., gnu gcc, and MS Visual Studio). The Qt 4.4.3 release is used in our study. The major purpose of this particular release is to supply bug fixes and performance developments based on both internal testing and client feedback.

TABLE 2. DETAILS OF THE CORPUS USED AS INPUT TO LSI FOR EACH OF THE TWO SYSTEMS USED IN THE EXPERIMENTAL STUDY.

	HippoDraw 1.21.3	Qt 4.4.3
Vocabulary Size	6,803	91,187
Number of Parsed Documents/Methods	3,706	70,871
Dimensionality Used	200	300

Table 2 describes the characteristics of HippoDraw and Qt in the context of their use for LSI. It is clear that Qt is a much larger system in all aspects. We apply both LSI and LSI+S separately to each system. This allows us to compare the results and assess their quality relative to each other for the context of the added stereotype information. The method level of granularity is chosen in both studies. We followed the same

methodology described in section IV in ranking the relevant parts of source code with respect to user query, with different dimensionality-reduction factors chosen for each study.

### B. Evaluation Measures

To evaluate the results of feature location, a number of studies [7, 9, 12], use the position of first relevant method as an effort measure. Other studies [31] use recall and precision measures. In addition to these we compute the total effort measurement and use the position of the last relevant method. All of these measures as well as p-value are used to evaluate the results of the LSI and LSI+S approaches.

First we use the standard IR measurements [4] *recall* and *precision*. Recall of 100% means that all the relevant documents are recovered, though there could be recovered documents that are not relevant. Precision of 100% means that all the recovered documents are relevant, though there could be relevant documents that were not recovered. Typically there is a tradeoff between precision and recall. If recall is high, then precision normally is low. If precision is high, then recall normally is low. In computing recall and precision we only include the first 100 (cut point) ranked items produced by the query as was shown and justified in [32]. This is a standard approach to computing these values as anything more than 100 is beyond what a developer would normally investigate. Recall and precision are defined as follows:

- Recall =  $|\text{relevant} \cap \text{retrieved}| \div |\text{relevant}|$
- Precision =  $|\text{relevant} \cap \text{retrieved}| \div |\text{retrieved}|$

The main goal of all feature-location techniques is to reduce the effort of the developers in the location process. Therefore, in this evaluation, as has been done by previous researchers [4], we measure the effort that the developers need (maintenance-effort measurements) as the number of methods from the retrieved ranked list that they have to investigate until finding the first *relevant* method (PFR), the last *relevant* method (PLR), and all *relevant* methods ( $\Sigma$  EM). Typically, with respect to maintenance-effort measurements, lower values are preferred. These measures are defined as follows:

- $\Sigma$  EM: Total Effort Measurement (number of methods the developer needs to investigate to find *all* relevant documents).
- PFR: Position of first relevant document.
- PLR: Position of last relevant document.

We use the Wilcoxon signed-rank test to examine whether the difference in terms of effectiveness ( $\Sigma$  EM - Total Effort Measurement) for two approaches is statistically significant by computing the p-value.

### C. Feature Selection & Determining Relevant Methods

For both systems in the study we selected 11 features for each (see Table 3 and Table 5). The features were selected based on bug reports in the online system documentation for both HippoDraw and Qt. A concatenation of the title and the description of the bug were used. Additionally, we conducted a

separate experiment for 14 more features found by inspecting eight bug reports in Qt.

Both systems have extensive and very complete documentation. Developers maintain very detailed bug reports and descriptions of the modification to fix each. We determined manually the set of relevant methods for each feature using this documentation as described below. For each feature we examined the related bug reports and descriptions of the fixes. We included all the methods that were modified in response to the bug fix and conducted a manual inspection of the code to determine all other methods relevant to that feature. Two graduate students did this inspection. They individually used the systems websites, bug-tracking reports<sup>1</sup>, source code, etc. This collected data was then examined and any differences were resolved by additional inspection. This process took approximately 20 person/hours for HippoDraw and approximately 60 person/hours for Qt.

TABLE 3. HIPPODRAW FEATURE DESCRIPTION, APPLIED QUERY, AND THE NUMBER OF RELEVANT METHODS FOR EACH FEATURE.

Feature	Query	Number Relevant Methods
1. change font size	<i>change font size weight set</i>	10
2. change font style	<i>change font style italic</i>	18
3. update zoom mode	<i>update zoom mode zoomin zoomout</i>	9
4. reset printer settings	<i>reset change printer settings</i>	8
5. add item	<i>insert add item canvas</i>	7
6. remove item	<i>Delete remove item canvas</i>	7
7. change mouse property	<i>Option change mouse property</i>	9
8. change cut color	<i>change cut color set</i>	7
9. change representation color	<i>change representation color set</i>	7
10. make new display	<i>make new display add make</i>	12
11. update axis modeling	<i>update axis modeling reset</i>	8

#### D. Locating Features in HippoDraw

For version 1.21.3 of HippoDraw we ran our experiment on the 11 features and queries described in Table 3. For the corpus that was re-documented, we examined the stereotypes of relevant methods. It was found that all of the relevant methods for all features were labeled with at least one stereotype. That is, no relevant method was unclassified, which is a possible result from the re-documentation process. For overall distributions and details of the specific stereotyping of the HippoDraw system we refer the reader to [14].

In order to find the best user query that describes the intended feature accurately and completely, other researchers

have used the process of formulating four different user queries and then choosing the best one among them [8]. The same procedure is followed here. For each feature in Table 3, the given query gives the best results of the four queries that were investigated. That is, the taken query gave the best ranking of the relevant documents for LSI. Table 3 also presents the number of relevant documents (methods) for each feature. With respect to dimensionality reduction for LSI and LSI+S, we determined 200 as the best value using the previously described method.

Table 4 summarizes the results obtained in identifying the features in the HippoDraw study. The first column indicates the feature number (from Table 3), the 2<sup>nd</sup> indicates the total effort measure, and the 3<sup>rd</sup> and the 4<sup>th</sup> columns indicate the positions of first and last relevant documents in the corpus respectively. As we can see in Table 4, using stereotypes (LSI+S) improved all three measures comparing with the result of using no stereotypes (LSI). The first relevant method (PFR) for LSI+S is equal or better to LSI. The precision and recall results are shown in Figure 2 and Figure 3, respectively. These figures show that LSI+S improves both recall and precision compared to LSI alone for most features. Specifically, the recall and precision improved for 9 features using LSI+S, while for 2 features the recall and precision are the same for both approaches.

TABLE 4. RESULT OF HIPPODRAW FOR THREE MEASUREMENTS; TOTAL EFFORT MEASUREMENT ( $\Sigma$  EM), POSITION OF FIRST RELEVANT DOCUMENT (PFR), AND POSITION OF LAST RELEVANT DOCUMENT (PLR).

Feature	Total Effort Measurement ( $\Sigma$ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	208	103	8	1	109	32
2	466	362	9	3	70	54
3	172	98	6	1	36	22
4	328	231	3	2	210	100
5	455	339	1	1	216	183
6	648	484	12	10	238	138
7	834	544	4	1	121	67
8	1595	764	2	2	1290	534
9	602	471	1	1	250	174
10	503	387	2	1	125	94
11	1721	843	3	1	1200	388

#### E. Locating Features in Qt

For version 4.4.3 of Qt we ran our experiment on the 11 features and queries described in Table 5. The same steps taken on the first system were also done here. Again, we chose four different queries, and then chose the best one among them. Experiments with different dimensionality reduction values showed that 300 gave the best results. Table 5 presents the summarization for all investigated features and the best queries used to locate these features. Table 6 summarizes the results obtained in identifying the features in the Qt study. As we can see LSI+S results in better values for all three measures compared with LSI alone. For this study, the precision and

<sup>1</sup> See <https://bugreports.qt-project.org>

recall results are also shown in Figure 4 and Figure 5 respectively. Again, LSI+S improves recall and precision.

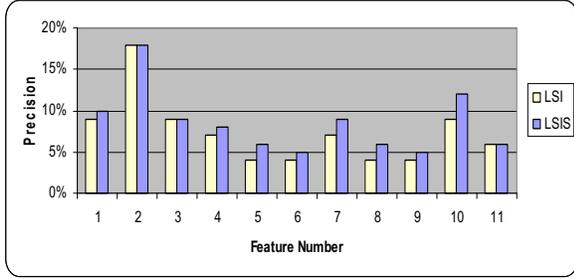


Figure 2. Precision results for the HippoDraw.

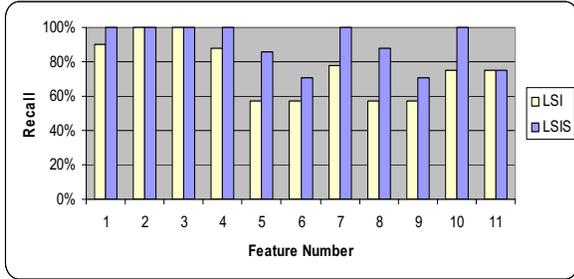


Figure 3. Recall results for the HippoDraw .

TABLE 5. QT FEATURE DESCRIPTIONS; FEATURE NAME, APPLIED QUERY, AND NUMBER OF RELEVANT METHODS TO EACH FEATURE.

Feature	Query	Number Relevant Methods
1. update font settings	font update options settings reset	21
2. create new font	create new font	24
3. change font size	size font change	23
4. set password	set password change	12
5. set RGB	update RGB color RGBA RGBF	7
6. add menu	add create new menu insert menubar	15
7. remove menu	menu remove delete	7
8. add action	insert action add new	11
9. remove action	action delete remove	9
10. search	index search searching searcher indexing find	12
11. draw polygon	points polygon draw lines polyline	7

In addition to the previous ones, we then examined an additional 14 features that were derived by investigating, eight new bug reports in Qt. These bug reports are given in Table 7. These 14 features were chosen because they were the most frequently changed. LSI+S improved or preserved the position

of the most relevant method in each case. For instance, the bug 24685 affected versions 4.7.4 and 4.8.0, and was fixed in version 4.8.3. Based on the bug description, it occurs when the method `QPainter::drawText()` is called from a thread. A memory leak occurs if the text contains Russian characters (i.e., "Время"). For this bug to be fixed the three functions `painter()`, `setFont()`, and `drawText()` all need to be modified. For the query we used the bug title "memory leak in `drawText()`". Using LSI these three methods were ranked 47, 65, and 11 respectively, while using LSI+S they were ranked 28, 31, and 1. An explanation for this result is that the function `drawText()` is overloaded 18 times, 9 of them have only one line of code in the body of the function, and were labeled with predicate or void-accessor. The others have different and more complex behavior, and were labeled as command-collaborator or void-accessor. In the context of our query the most relevant `drawText()` function is labeled with command-collaborator like the other two relevant methods `painter()` and `setFont()`. This function is ranked in the first position using LSI+S, while it is ranked in the 11th position using LSI alone.

Another example is the bug 11204, which impacts version 4.6.2, and is fixed version 4.7.1. Based on the description of this bug, this bug involves two features "direction of text" and "alignment of text". Table 8 gives the relevant methods for this bug, and how they were ranked using both techniques. In this experiment we used the bug title "direction change no longer implies alignment change" as a query. The total effort measure for those new 14 features is examined, LSI+S has better values for all features with 38% average improvement. Moreover, the position of the most relevant method is improved using LSI+S for 10 out of 14 features, where for the remaining 4 features, LSI+S gives the same ranks as shown in Table 7.

TABLE 6. RESULT OF QT FOR THREE MEASUREMENTS; TOTAL EFFORT MEASUREMENT (EM), POSITION OF FIRST RELEVANT DOCUMENT (PFR), AND POSITION OF LAST RELEVANT DOCUMENT (PLR).

Feature	Total Effort Measurement ( $\Sigma$ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	2208	1846	2	1	1054	332
2	1900	928	1	1	520	467
3	1668	1192	4	1	684	443
4	1760	996	4	1	710	359
5	112	100	19	8	59	40
6	2792	1667	2	1	830	451
7	251	149	1	1	101	94
8	1239	701	3	1	815	456
9	359	185	1	1	153	100
10	1078	599	2	1	184	150
11	1641	566	1	1	1321	450

## VI. DISCUSSION

Our hypothesis was that adding stereotype information to the corpus (source code) would improve the results of LSI in the context of the feature-location problem. It is quite clear

from the data that the addition of the stereotype information does improve the results of feature location using LSI for the presented queries in the context of these two systems. In all cases, and for all measures, LSI+S has equal or better values.

Examining the results of the studies, given in Table 4 and Table 6, we see that the position of the first relevant method improved with LSI+S in approximately 75% of the queries. The remaining 25% produced the same value. Moreover, the position of the first relevant method for LSI+S is in the first position in 7 of the 11 features for HippoDraw and 10 of the 11 features in Qt. Using LSI alone produced first positions of 2 of 11 for HippoDraw and 4 of 11 for Qt. This is a particularly nice improvement in the context of usability for the developer, as they need not look far into the list for a result.

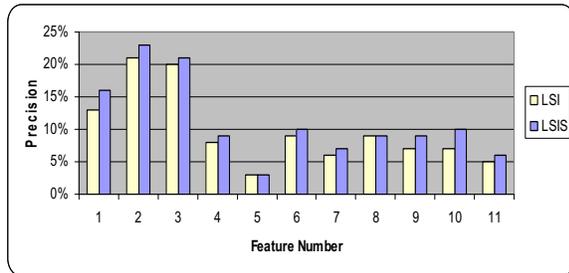


Figure 4. Precision for the 11 features from Qt.

Furthermore, the position of the last relevant method has been improved for all studied features in all cases with LSI+S. The improvement in this measure is much more evident (approximately one half on average). In

Table 9 we summarize the difference between the first and last relevant method positions for the two approaches for HippoDraw and Qt respectively. We see that there is an average improvement for these 11 features of 43% for HippoDraw and 36% for Qt in the distance from the first relevant method to the last relevant method.

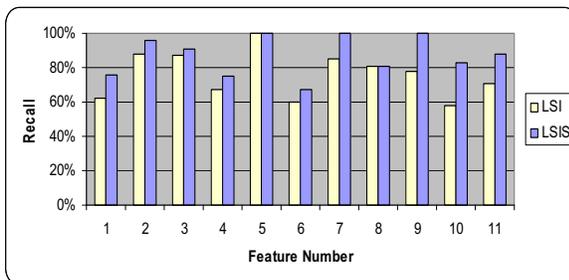


Figure 5. Recall for the 11 features from Qt .

The total effort measure is examined in Table 4 and Table 6. LSI+S again has better values for all queries. The average improvement is 46% with a range of 11% to 66% for both HippoDraw and Qt. From a usability standpoint this means that a developer would need to wade through far fewer methods on average to find all relevant methods.

TABLE 7. DESCRIPTION OF EIGHT BUGS (14 FEATURES) FROM QT BUG REPORTS..

Bug Number	Component	Number Relevant Methods	Rank of Most Relevant	
			LSI	LSI+S
24685 (1)	GUI: Font handling	3	11	1
15754 (3)	GUI: Font handling	7	6	3
11204 (2)	GUI: Text handling	4	3	1
5002 (2)	GUI: OpenGL	10	5	5
4210 (2)	GUI: Painting	9	7	4
2276 (1)	Widgets: Itemviews	13	11	9
1868 (2)	GUI: Text handling	8	1	1
935 (1)	GUI: Workspace	7	25	14

TABLE 8. RESULTS FOR LOCATING THE RELEVANT METHODS FOR BUG 11204.

Rank LSI+S	Relevant Methods	Rank LSI
1	direction()	43
262	setTextDirection()	285
5	setAlignment()	5
17	fixedAlignment()	21

TABLE 9. THE DIFFERENCE OF FIRST RELEVANT AND LAST RELEVANT METHOD FOR EACH QUERY IN HIPPODRAW AND QT. THE PERCENTAGE COLUMN IS THE IMPROVEMENT USING LSI+S.

Feature	HippoDraw			Qt		
	LSI	LSI+S	%	LSI	LSI+S	%
1	101	31	69%	1052	331	69%
2	61	51	16%	519	466	10%
3	30	21	30%	680	442	35%
4	207	98	53%	706	358	49%
5	215	182	15%	40	32	20%
6	226	128	43%	828	450	46%
7	117	66	44%	100	93	7%
8	1288	532	59%	812	455	44%
9	249	173	31%	152	99	35%
10	123	93	34%	182	149	18%
11	1197	387	78%	1320	449	66%
<b>Average Improvement</b>			<b>43%</b>			<b>36%</b>

The results for recall and precision for both studies are shown in Figure 2, Figure 3, Figure 4, and Figure 5. For both systems LSI+S has equal or better precision and recall values. Other studies that have used LSI alone [6] or combined with other analysis [2, 5] approaches produce comparable precision and recall values. This improvement appears to be on the same order as what has previously been observed.

The Wilcoxon signed-rank test was performed to investigate whether the difference in terms of effectiveness for the two approaches is statistically significant. We computed it for the ( $\Sigma EM$ ) dependent variable. The null hypothesis is that there is no statistical significant difference in terms of effectiveness between LSI and LSI+S. The alternative hypothesis is that LSI+S has statistically significant higher

effectiveness than LSI. Our results for the two systems were found to be statistically significant. The p-value is lower than  $\alpha = 0.05$  for the two systems, and was actually less than 0.0001. This allows us to reject the null hypothesis.

All the data from the three experimental studies supports the hypothesis that the addition of relevant information (in this case the stereotype) improves the results of querying in the context of feature location. This lays the foundation to generalize the result further, however we need to explain why this particular type of information helps. Beyond the abstract information-theoretic explanation (i.e., more information will give you better results) it would be prudent to understand some of the specific reasons that we see improvements.

It has been found that when using LSI, methods with small bodies and small numbers of identifiers are not ranked correctly [9] because there is not enough terms to properly build an accurate vector representation. However, the addition of stereotypes seems to mitigate this problem to some degree. That is, small methods appear to be ranked more correctly with the extra stereotype information. For example, in HippoDraw feature 3 “update zoom” using LSI resulted in the first relevant function `getZoomMode()` being ranked in the 6<sup>th</sup> position, while using LSI+S it is ranked first. We investigated this further and made some interesting observations. LSI ranked the function `hasZoomY()` in the first position, which is not relevant to the feature. However, `hasZoomY()` is small with only a couple lines of code. When re-documented, it is labeled with the *predicate* stereotype. This additional information changed the similarity between it and the query. We observed this same type of situation happening elsewhere. That is, small methods being ranked high by LSI but after being labeled with stereotypes receiving a much lower ranking.

We believe that using the stereotype information acts as a type of filtering mechanism when building the LSI subspace. That is, simple methods such as `get/set`, are superficially related to a feature, as they rarely impact the actual behavior and often play little part in the actual maintenance task. However, this belief is speculative in part and further investigation is needed to substantiate or generalize this hypothesis.

Stereotypes, by nature, increase the similarities between any two methods that have the same category. Since stereotypes are an abstract summary of a method’s role and behavior, therefore, this implies that methods with similar roles will be made more similar (within the LSI subspace). Table 10 presents an overview of how the relevant methods were stereotyped. This is for both systems across all the 36 features. There were 311 relevant methods. We see that the vast majority (almost 90%) are labeled with the *command* and/or *collaborator* stereotypes. Approximately 6% are predicates and the remaining is a variety with no single stereotype category making up more than 5.4%. In short, the most relevant methods, in these three studies, are almost always some type of *command* or *collaborator* method. We observed this distribution after running the studies while attempting to better understand the results.

*Command* and *collaborator* methods do the majority of the logic within a class. They model the behavior of a class and

hence provide most of behavior of observable system features. Thus, it makes sense that the most relevant methods for any system feature would most likely be of the *command* stereotype.

TABLE 10. DISTRIBUTION OF STEREOTYPES FOR THE RELEVANT METHODS OVER BOTH STUDIES. THE OTHER 17 WERE A VARIETY OF DIFFERENT STEREOTYPES WITH NO ONE CATEGORY MAKING UP MORE THAN 2%.

Stereotype	Number of Methods	Ratio (%)
<i>Command-Collaborator</i>	221	71%
<i>Command</i>	53	17%
<i>Predicate</i>	20	6%
Others	17	6%
<b>Total</b>	<b>311</b>	<b>100%</b>

## VII. THREATS TO VALIDITY

A number of issues could affect the results of the study we conducted and so may limit the generalizability of the results. The authors attempted to minimize factors so to decrease their effect. Feature selection is an issue. We picked features that were commonly modified in the systems based on the documentation. We also needed features for which all relevant methods could be identified. As such they were selected with no preconceived notion of how well either LSI or LSI+S would perform on them.

The number of queries we used could also be too few for a rigorous comparison. Compared to other studies on feature location [8] the number we used, 30 queries over 36 features, represents a bit larger set (i.e., previous studies have used as few as three queries). However, other studies [33] have used more but they depend on bug reports titles or descriptions directly as a query without filtering or preprocessing. They also only include items that were changed due to the bug report. This may not include all relevant items, but only relevant items that were changed. Another issue is if the features used in this study are representative to those used in practice. Taking features directly from active open-source systems minimizes this to a degree. Also, these features were involved in actual maintenance tasks. We also minimized this threat by selecting two different systems from two different domains. Expanding the study to other systems could further minimize this issue. Another issue is that query selection depends on the knowledge of the user. We attempted to minimize this by selecting the best query for LSI from the set of four queries.

Lastly, we may not have found all relevant methods or may have labeled methods as relevant that actually were not. This was addressed by a careful manual inspection of the systems and associated documentation.

## VIII. CONCLUSIONS AND FUTURE WORK

A novel technique to improve the results of using Latent Semantic Indexing on the problem of feature location is introduced. The technique involves adding new information to the source code before applying LSI. In this case, the new information added is method stereotypes, which were derived via static program analysis from the source code.

We compared the results of using LSI on the original code base with that of a version re-documented with stereotype

information. This experimental study on two open-source systems demonstrated that the added stereotype information improved the query results for the feature-location process. We saw substantial average improvements in the results for all measures. For each individual query we saw equal or better results in all cases when using the stereotype information. The results were compared using recall, precision, position of first and last relevant document, and a total effort measure.

The implications of these results are important for a number of reasons. The results confirm that adding information to a corpus (here source code) will improve the results for extracting and querying that corpus. The results provide evidence that the addition of other information than stereotypes, gained via static or dynamic analysis of the code, could also improve the results. The results also imply that stereotype information is relevant for feature location (and comprehension), which supports our previous studies on stereotypes. This last issue could give rise to a new means for evaluating techniques to support comprehension. If we claim that adding or deriving particular information from source code supports comprehension, then it should in theory also improve the results of IR methods such as LSI.

Future work on this topic includes a better understanding of why stereotypes improve the results. Additionally, we are investigating what other types of information added to source code improves the results of IR methods.

#### REFERENCES

- [1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *CACM*, vol. 37, pp. 72-82, 1994.
- [2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *JSME*, vol. 25, pp. 53-95, 2013.
- [3] R. J. Turver and M. Munro, "An early impact analysis technique for software maintenance," *JSME*, vol. 6, pp. 35-52, 1994.
- [4] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution," in *Ency. of SE*, 2010.
- [5] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *TSE*, vol. 29, pp. 210-224, 2003.
- [6] A. Marcus, A. Sergeev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *WCRE*, 2004, pp. 214-223.
- [7] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *TSE*, vol. 33, pp. 420-432, 2007.
- [8] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE*, 2007, pp. 234-243.
- [9] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," in *ICPC*, 2006, pp. 137-148.
- [10] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *ICPC*, 2007, pp. 37-48.
- [11] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *ICSE*, 2001, pp. 103-112.
- [12] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in *ICPC*, 2010, pp. 14-23.
- [13] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeev, "IRISS - A Source Code Exploration Tool," in *ICSM*, 2005, pp. 69-72.
- [14] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in *ICSM*, 2006, pp. 24-34.
- [15] A. Perotte, N. Bartlett, N. Elhadad, and F. Wood, "Hierarchically Supervised Latent Dirichlet Allocation," in *Neural Information Processing Systems*, ed, 2011.
- [16] H.-M. Müller, E. E. Kenny, and P. W. Sternberg, "Textpresso: An Ontology-Based Information Retrieval and Extraction System for Biological Literature," in *PLoS Biol.*, 2, e309, ed, 2004.
- [17] J. Teevan, "Improving Information Retrieval with Textual Analysis: Bayesian Models and Beyond," Master's thesis, Massachusetts Institute of Technology, 2001.
- [18] D. C. Blair, "Information retrieval and the philosophy of language," *Review of Info Science and Tech*, vol. 37, pp. 3-50, 2003.
- [19] T. W. C. Huibers, M. Lalmas, and C. J. v. Rijsbergen, "Information retrieval and situation theory," *SIGIR Forum*, vol. 30, pp. 11-25, 1996.
- [20] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *ICSM*, 2010, pp. 1-10.
- [21] D. Poshyvanyk, "Using information retrieval to support software maintenance tasks," in *ICSM*, 2009, pp. 453-456.
- [22] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," in *IWPC*, 2000, pp. 241-249.
- [23] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *JSME*, vol. 7, pp. 49-62, 1995.
- [24] E. Wong, S. Gokhale, and J. Horgan, "Quantifying the closeness between program components and features," *JSS*, vol. 54, pp. 87-98, 2000.
- [25] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *ICPC*, 2009, pp. 218-222.
- [26] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNI AFL: Towards a Static Non-Interactive Approach to Feature Location," in *ICSE*, 2004, pp. 293-303.
- [27] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *SCAM*, 2011, pp. 173-184.
- [28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *J. Am. Soc. of Info Science*, vol. 41, pp. 391-407, 1990.
- [29] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [30] R. K. Yin, *Case Study Research: Design and Methods (4th Edition)*. Thousand Oaks, CA: Sage, 2009.
- [31] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *ICSE*, 2011, pp. 111-120.
- [32] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimothy, "Using information retrieval based coupling measures for impact analysis," *EMSE*, vol. 14, pp. 5-32, 2009.
- [33] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *TOSEM*, vol. 21, pp. 1-34, 2013.

## Will Fault Localization Work For These Failures ?

### An Automated Approach to Predict Effectiveness of Fault Localization Tools

Tien-Duy B. Le and David Lo  
 School of Information Systems,  
 Singapore Management University, Singapore  
 {btdle.2012,davidlo}@smu.edu.sg

**Abstract**—Debugging is a crucial yet expensive activity to improve the reliability of software systems. To reduce debugging cost, various fault localization tools have been proposed. A spectrum-based fault localization tool often outputs an ordered list of program elements sorted based on their likelihood to be the root cause of a set of failures (i.e., their suspiciousness scores). Despite the many studies on fault localization, unfortunately, however, for many bugs, the root causes are often low in the ordered list. This potentially causes developers to distrust fault localization tools. Recently, Parnin and Orso highlight in their user study that many debuggers do not find fault localization useful if they do not find the root cause early in the list.

To alleviate the above issue, we build an oracle that could predict whether the output of a fault localization tool can be trusted or not. If the output is not likely to be trusted, developers do not need to spend time going through the list of most suspicious program elements one by one. Rather, other conventional means of debugging could be performed. To construct the oracle, we extract the values of a number of features that are potentially related to the effectiveness of fault localization. Building upon advances in machine learning, we process these feature values to learn a discriminative model that is able to predict the effectiveness of a fault localization tool output. In this preliminary work, we consider an output of a fault localization tool to be effective if the root cause appears in the top 10 most suspicious program elements. We have experimented our proposed oracle on 200 faulty programs from Space, NanoXML, XML-Security, and the 7 programs in Siemens test suite. Our experiments demonstrate that we could predict the effectiveness of fault localization tool with a precision, recall, and F-measure (harmonic mean of precision and recall) of 54.36%, 95.29%, and 69.23%. The numbers indicate that *many* ineffective fault localization instances are identified *correctly*, while only *very few* effective ones are identified *wrongly*.

#### I. INTRODUCTION

Despite the advancement in software tools and processes, bugs are prevalent in many systems. In 2002, it was reported that software bugs cost US economy more than 50 billion dollars annually [34]. Software testing and debugging cost itself is estimated to account for 30-90% of the total labor spent on a project [4]. Thus there is a need to develop automated means to help reduce software debugging cost. One important challenge in debugging is to localize the root cause of program failures. When a program fails, it

is often hard to locate the faulty program elements that are responsible for the failure. The root cause could be located far from the location where the failure is exhibited, e.g., the location where a program crashes or produces a wrong output.

In order to address the high cost of debugging in general, and help in localizing root causes of failures in particular, many spectrum-based fault localization tools have been proposed in the literature, e.g., [19], [1], [24]. These tools typically take in a set of normal execution traces and another set of faulty execution traces. Based on these set of program execution traces, these tools assign suspiciousness scores to various program elements. Next, program elements could be sorted based on their suspiciousness scores in descending order. The resultant list of suspicious program elements can then be presented to a human debugger to aid him/her in finding the root cause of a set of failures.

An *effective* fault localization tool would return a root cause at the top of a list of suspicious program elements. Although past studies have shown that fault localization tools could be effective for a number of cases, unfortunately, for many other cases, fault localization tools are not effective enough. Root causes are often listed low in the list of most suspicious program elements. Parnin and Orso pointed out in their user study that many developers do not find fault localization useful if they do not find the root cause early in the list [26]. This *unreliability* of fault localization tools potentially cause many developers to distrust fault localization tools.

In this work, we plan to increase the usability of fault localization tools by building an oracle to predict if a particular output of a fault localization tool is likely to be effective or not. We define an output of a fault localization tool to be effective if the faulty program element or root cause is listed among the top-10 most suspicious program elements. With our tool, the debuggers could be better informed whether he can trust or distrust the output of a fault localization tool run on a set of program execution traces. The following scenarios illustrate the benefits of predicting the effectiveness of a fault localization output:

*Scenario 1 - Without Oracle:* Tien-Duy had 10 bugs to fix.

He ran a fault localization tool for the 10 bugs. He followed the tool recommendations, however he only found 2 of the 10 recommendations to be effective. He wasted much time following 8 bad recommendations given by the tool.

*Scenario 2 - With Oracle:* Tien-Duy had 10 bugs to fix. He ran a fault localization tool for the 10 bugs and he had an oracle that can predict which fault localization outputs are likely to be effective. The oracle predicted that 3 outputs are likely to be effective. For 2 out of the 3 outputs, the fault localization outputs are indeed effective and saved Tien-Duy much time. Tien-Duy only wasted time following 1 bad recommendation.

To build the oracle, we extract values of important features from the execution traces and outputs of fault localization tools. These feature values extracted from a training data are then used to build a discriminative model leveraging a machine learning solution. The resultant discriminative model serves as an oracle and could be used to predict the effectiveness of a fault localization tool on other inputs.

We have experimented our approach on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. We investigate a well known spectrum-based fault localization tool namely Tarantula [19] which was also studied by Parnin and Orso [26]. Our experiments show that we can predict whether a fault localization tool is effective or not by a precision, recall, and F-measure (i.e., harmonic mean of precision and recall) of 54.36%, 95.29%, and 69.23%. We also investigate if our tool is effective to help two other fault localization tools, i.e., Ochiai [1], and Information Gain [24], with promising results.

In this work, our contributions are as follows:

- 1) We define a new research problem namely predicting the effectiveness of a fault localization tool given a set of execution traces. Solving this problem would help developers to better trust the output of a fault localization tool.
- 2) We present a machine learning framework to tackle the research problem. We propose a novel set of features that are relevant for predicting the effectiveness of a fault localization tool. We build upon and extend a state-of-the-art machine learning solution for the prediction problem by addressing the issue of imbalanced data. The issue of imbalanced data occurs since many outputs of Tarantula are ineffective.
- 3) We have evaluated our approach on 200 faulty programs from NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite. We show that we could achieve a precision, recall, and F-measure of 54.36%, 95.29%, and 69.23%. This shows that *many* ineffective and *almost all* effective outputs of Tarantula are detected correctly.

The structure of this paper is as follows. In Section II,

Table I  
SPECTRA NOTATIONS

Symbol	Definition
$n$	Total number of test cases in the test suite
$n^e$	Number of test cases that executes a program element $e$
$n_s$	Number of test cases that pass
$n_f$	Number of test cases that fail
$n_s^e$	Number of test cases that execute $e$ and pass
$n_f^e$	Number of test cases that execute $e$ and fail

we describe preliminary materials on spectrum-based fault localization and an intuition how effectiveness prediction could be solved. In Section III, we present a birds-eye-view of our proposed framework. Section IV outlines what features are extracted from the execution traces and output of the fault localization tool. Section V elaborates our approach to learn a discriminative model using a classification algorithm and how we address the problem of imbalanced data. We present our experiment settings, datasets, and results which answer a number of research questions in Section VI. We discuss related studies in Section VII. We finally conclude and mention future work in Section VIII.

## II. PRELIMINARIES & PROBLEM DEFN.

In this section, we first introduce fault localization. We then define the problem of effectiveness prediction and give some intuitions on how this could be solved.

### A. Fault Localization

Fault localization takes as input a faulty program, along with a set of test cases, and a test oracle. The faulty program is instrumented such that when a test case is run over it, a program spectra is generated. A program spectra records certain characteristics of a particular program run and thus it becomes a behavioral signature of the run [28]. This program spectra could constitute a set of counters which record how many times different program elements (e.g., statement, basic block, etc) are executed in a particular program run [14]. Alternatively, the counter could record a boolean flag that indicates whether a program element is executed or not. The test oracle is used to decide if a particular program run is correct or faulty. Faulty runs or executions are also referred to as failures. Fault localization task is to analyze program spectra of correct and faulty runs with the goal of finding program elements that are the root causes of the failures (i.e., the faults or errors).

Various spectra have been proposed in past studies [14]. In this study, we use *block-hit spectra*; we instrument every block of a program and collect information on which blocks are executed in a run. Block-hit spectra is suitable as all statements in a basic block have the same execution profile. It has also been shown in the literature that the cost of collecting block-hit spectra is relatively low and the resultant spectra could be used for fault localization [1], [14].

Figure 1 shows an example code with several program spectra. The identifiers of the basic blocks are shown in the first column. The statements located in the basic blocks are

Blk ID	Program Elements	T1	T2	T3	T4
1	int count, n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO)/*maxprio=3*/	•	•	•	•
2	{return;}	•			
3	src_queue = prio_queue[prio]; dest_queue = prio_queue[prio+1]; count = src_queue->mem_count; if (count > 1)/* Bug!/* supposed : count >= 1*/{		•	•	•
4	n = (int)(count*ratio + 1); proc = find_nth(src_queue, n); if (proc){		•	•	
5	src_queue = del_ele(src_queue, proc); proc->priority = prio; dest_queue = append_ele(dest_queue, proc);}}		•	•	
Status of Test Case Execution :		P	P	P	F

Figure 1. Four Block-Hit Program Spectra

shown in the second column. There is a bug in the example code at basic block three; the condition of the if statement should be “count >= 1” instead of “count > 1”. Columns 3 to 6 show the program spectra that are produced when four test cases are run. Three of the test cases do not expose the bug, i.e., running them result in correct executions. The fourth test case exposes the bug, i.e., running it result in a faulty execution. A cell marked by a • indicates that a particular basic block is executed when a particular test case is run. An empty cell indicates that a particular basic block is not executed when a particular test case is run.

To identify the faulty program elements (e.g., basic block 3 in Figure 1), we compute the suspiciousness scores of each of the program elements. There are various ways to define suspiciousness. In this work, we primarily consider a well-known suspiciousness score defined by Jones and Harrold, named Tarantula [19]. Considering several notations in Table I, Tarantula’s suspiciousness score can be defined as follows:

$$Tarantula(e) = \frac{\frac{n_f^e}{n_f}}{\frac{n_s^e}{n_s} + \frac{n_f^e}{n_f}}$$

Tarantula considers an element more suspicious if it occurs more frequently in failed executions than in correct executions. Considering the example shown in Figure 1, the suspiciousness score of block 1 is:  $\frac{1}{(1+1)} = 0.5$ . The suspiciousness scores of block 2, 4, and 5 are zeros since the numerator of Tarantula (i.e.,  $\frac{n_f^e}{n_f}$ ) is zero. The suspiciousness score of block 3 is:  $\frac{1}{(\frac{2}{3}+1)} = 0.6$ . Thus using Tarantula, the most suspicious block is block 3, followed by block 1, followed by blocks 2, 4, and 5. We could sort the basic blocks based on their suspiciousness scores and the debugger could check the blocks one-by-one from the most to the least suspicious block. Following Tarantula’s recommendation, the fault could be found after one basic block inspection.

## B. Effectiveness Prediction

The goal of our work is to predict if a particular fault localization tool is effective for a particular set of execution traces. We refer to the process where a fault localization tool is used to process a set of execution traces and output a list of suspicious program element as a *fault localization instance*. We define a fault localization instance to be effective if the root cause is located among the top-10 most suspicious program elements. Ties are randomly broken; this means that for example, if the top-20 program elements have the same suspiciousness scores, we randomly select 10 out of the 20 to be the top-10. Also, in case the root cause spans more than one program element (i.e., basic block) as long as one of the program elements is in the top-10, we consider the fault localization instance to be an effective one.

Various information could be leveraged to predict if a fault localization tool is effective given a set of program execution traces. We could investigate the execution traces. If there are very few failing execution traces, then it is likely to be harder for a spectrum based fault localization tool to differentiate faulty from correct program elements. In the extreme case, when there are no test cases that expose the fault (no failing execution traces), then the output of a fault localization tool cannot be effective. We could also investigate the output of the fault localization tool. In the special case where all program elements are given the same suspiciousness score, there is a very low likelihood that the fault localization tool will be effective for those execution traces.

## III. OVERALL FRAMEWORK

The goal of our framework is to build an oracle that is able to predict if a fault localization instance is effective or not. To realize this, our framework, illustrated in Figure 2, works on two phases: training and deployment. The training phase would output a model that is able to differentiate effective and ineffective fault localization instances. The deployment phase would apply this model to a number of unknown fault localization instances and output if the cases are likely to be effective or not. Let us describe these two phases in more detail.

In the training phase, we take in a set of fault localization instances. Some of these cases are effective and some others are ineffective. Each of these cases is represented by the following:

- 1) Program *spectra* corresponding to correct and faulty execution traces.
- 2) A list of *suspiciousness scores* that are assigned by the fault localization tools to the program elements.
- 3) An *effectiveness label*: effective (if the root cause is in the top-10) or ineffective (otherwise).

The training phase consists of two processes: feature extraction, and model learning. During feature extraction, based on a training data, we extract some feature values

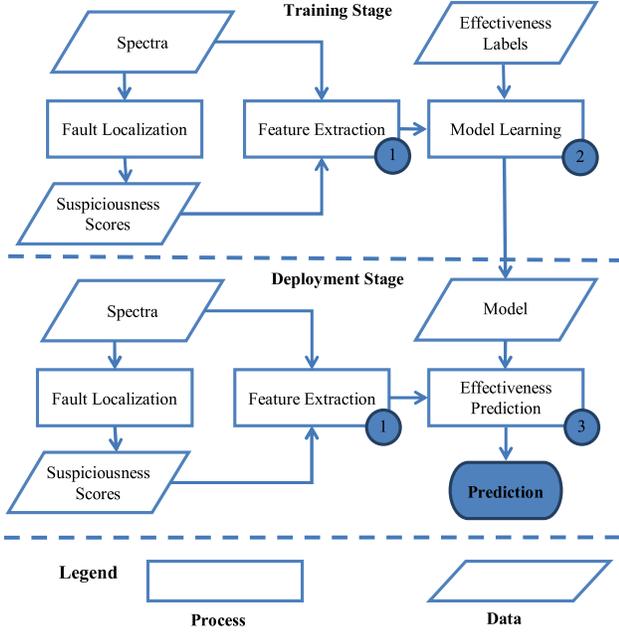


Figure 2. Proposed Framework

that shed light into some important characteristics that potentially differentiate effective from ineffective instances. In the model learning step, the feature values of each of the training instances along with the effectiveness labels are then used to build a discriminative model which is able to predict whether an unknown fault localization instance is effective or not. This discriminative model is output to the deployment stage.

The deployment stage consists of two blocks: feature extraction, and effectiveness prediction. We extract feature values from unknown instances whose labels, effective or ineffective, are to be predicted. These values are then fed to the discriminative model learned in the training phase. The model would then output a prediction.

We elaborate the feature extraction block in Section IV. The model learning and effectiveness prediction blocks are elaborated in Section V.

#### IV. FEATURE EXTRACTION

We extract values of a number of features from input execution traces and from the outputs of a fault localization tool. Table II shows these features. We have in total 50 features. Fifteen of the features are extracted from input execution traces and the remaining thirty five features are extracted from the suspiciousness scores output by the tool.

The first fifteen input features capture information about program execution traces and program elements covered by these execution traces. Features  $T_1$  to  $T_5$  capture information on the number of execution traces available for fault localization. Too few number of traces might cause poor fault localization performance especially if there are too

few failing traces. In the worst case where the number of failing traces is zero, the fault localization tool reduces to random guess. Features  $PE_1$  to  $PE_4$  capture the information on program elements that are covered by the execution traces. The more the number of program elements, the more difficulty a fault localization tool is likely to have as it needs to compare and differentiates more elements. With more program elements, the more likely a faulty program element to be assigned the same or lower suspiciousness scores as other program elements. Feature  $PE_5$  captures cases where some program elements only appear in faulty but not correct executions. Intuitively, the chance for such cases to be effective is likely to be high. Feature  $PE_6$  captures the opposite which might indicate omission errors: some program elements that should be executed are not executed. Features  $PE_7$  to  $PE_{10}$  capture the two highest proportions of failures that passed by one program element. Intuitively, the higher the proportion of failures that passes a program element, the more likely it is the root cause.

The next thirty five output features capture the suspiciousness scores that are output by the fault localization tool. Features  $R_1$  to  $R_{10}$  capture the top-10 suspiciousness scores. If the suspiciousness scores are too low, intuitively it is less likely for a fault localization instance to be effective. Features  $SS_1$  to  $SS_6$  compute some simple statistics of the top-10 suspiciousness scores. They serve as statistical summary of the scores. Features  $G_1$  to  $G_{11}$  and  $C_1$  to  $C_8$  are aimed to capture a “break” or gap in the top-10 suspiciousness scores. This “break” shows that the localization tool is able to differentiate some program elements to be significantly more suspicious than the others. That might indicate that some of the top-10 program elements are probably to be the root cause. If the fault localization tool is unable to differentiate program elements, it is less likely to be effective. In the worst case, if it is unable to distinguish all program elements, fault localization again turns into random guess.

#### V. MODEL LEARNING & EFFECTIVENESS PREDICTION

We first describe our model learning process. Next, we describe how we apply the model to effectiveness prediction.

##### A. Model Learning

As inputs to this process, we have a set of training instances with their effectiveness labels. Each of the instance is represented as 50 feature values (aka. a feature vector) produced by the feature extraction process described in Section IV. The goal of the model learning process is to convert these set of feature vectors into a discriminative model that could predict the effectiveness label of a fault localization instance whose effectiveness is unknown.

We build upon and extend a state-of-the-art classification algorithm namely Support Vector Machine (SVM) [13]. SVM has been used in many past software engineering research studies, e.g., [2], [33], [25], [35], [36]. We first

Table II  
LIST OF FEATURES (50 FEATURES)

ID	Description
<b>Input: Traces (5 Features)</b>	
$T_1$	Number of traces
$T_2$	Number of failing traces
$T_3$	Number of passing traces
$T_4$	$T_3 - T_2$
$T_5$	$\frac{T_2}{T_3}$
<b>Input: Program Elements (10 Features)</b>	
$PE_1$	Number of program elements covered in the failing execution traces
$PE_2$	Number of program elements covered in the correct execution traces
$PE_3$	$PE_2 - PE_1$
$PE_4$	$\frac{PE_1}{PE_2}$
$PE_5$	Number of program elements that appear only in failing execution traces
$PE_6$	Number of program elements that appear only in correct execution traces
$PE_7$	Highest proportion of failing execution traces that pass by one program element
$PE_8$	Second highest proportion of failing execution traces that pass by one program element
$PE_9$	$PE_7 - PE_8$
$PE_{10}$	$\frac{PE_8}{PE_7}$
<b>Output: Raw Scores (10 Features)</b>	
$R_1$	Highest suspiciousness score
$R_2$	Second highest suspiciousness score
$R_i$	$i^{th}$ highest suspiciousness score, where $3 \leq i \leq 10$
<b>Output: Simple Statistics (6 Features)</b>	
$SS_1$	Number of distinct suspiciousness scores in $\{R_1, \dots, R_{10}\}$
$SS_2$	Mean of $\{R_1, \dots, R_{10}\}$
$SS_3$	Median of $\{R_1, \dots, R_{10}\}$
$SS_4$	Mode of $\{R_1, \dots, R_{10}\}$
$SS_5$	Variance of $\{R_1, \dots, R_{10}\}$
$SS_6$	Standard deviation of $\{R_1, \dots, R_{10}\}$
<b>Output: Gaps (11 Features)</b>	
$G_1$	$R_1 - R_2$
$G_2$	$R_2 - R_3$
$G_i$	$R_i - R_{(i+1)}$ , where $3 \leq i \leq 9$
$G_{10}$	$Max_{1 < i < 9} (G_i)$
$G_{11}$	$Min_{1 < i < 9} (G_i)$
<b>Output: Relative Differences (8 Features)</b>	
$C_1$	$\frac{(R_2 - R_{10})}{(R_1 - R_{10})}$
$C_i$	$\frac{(R_{(i+1)} - R_{10})}{(R_1 - R_{10})}$ , where $2 \leq i \leq 8$

describe standard off-the-shelf SVM. We then describe our extended SVM that handles the issue of imbalanced data caused since there are more ineffective fault localization instances than effective ones.

1) *Off-the-Shelf SVM*: SVM solves the classification problem by looking for a linear optimal separating hyperplane, which separates data instances of one class from another [37]. The chosen hyperplane is called *maximum marginal hyperplane* (MMH) in which the separation between two classes are maximized. For example, consider a training dataset in form of  $(\vec{x}_k, y_k)$ , where  $\vec{x}_k$  is the feature vector of the  $k^{th}$  training data instance. Each  $y_k$  represents class label of data instance ( $y_k \in \{+1, -1\}$ ). The problem of searching for a separating hyperplane with maximal margin could be reduced to finding the minimal value of  $\frac{1}{2} \|\vec{w}\| = \frac{1}{2} \sqrt{w_1^2 + \dots + w_n^2}$  which satisfies the

constraints:  $y_k(\vec{w} \cdot \vec{x}_k + b) \geq 1 \forall k$ , where  $\vec{w}$  is perpendicular to the separating hyperplane,  $n$  is the number of attributes, and  $b$  is a constant number indicates position of the hyperplane in multi-dimensional space. In this study, we use SVM<sup>light</sup> version 6.02<sup>1</sup> with linear kernel.

2) *SVM<sup>Ext</sup>*: Imbalanced training data is one of the issues that we encounter during the course of our study. There are more ineffective than effective fault localization instances. Thus we build upon standard off-the-shelf SVM to address this imbalanced data problem. We call our solution SVM<sup>Ext</sup>.

The pseudo-code of our proposed SVM<sup>Ext</sup> is shown in Figure 3. The algorithm takes as input a set of effective and ineffective fault localization instances - *EI* and *II*. We first check if there are more ineffective than effective localization instances (Line 1). If there are, we perform a data balancing step (Lines 2-8). We would like to duplicate effective instances that appear close to the hyperplane – these are effective instances that are close to one of the ineffective instances. In order to find these effective instances, we compute the similarity between each effective instance with each of the ineffective instances (Line 2). Each fault localization instance could be viewed as a 50-dimensional vector; each dimension is a feature and a localization instance is represented by the values of the 50 features described in Section IV. To measure the similarity between two instances we compute the Cosine similarity [29] of their representative vectors. Consider two vectors  $(a_1, \dots, a_{50})$  and  $(b_1, \dots, b_{50})$ . The Cosine similarity of these two vectors is defined as:

$$\frac{\sum_{i=1}^{50} (a_i \times b_i)}{\sqrt{\sum_{i=1}^{50} (a_i)^2} \times \sqrt{\sum_{i=1}^{50} (b_i)^2}}$$

Next, for each effective instance, we calculate its highest similarity with an ineffective instance (Line 3). We sort the effective instances based on their highest similarities with ineffective instances (Line 4). We then insert these instances from the most similar to the least similar to the collection of effective instances *EI* until the number of effective instances matches that of ineffective ones (Lines 5-8). We then proceed to learn a model using off-the-shelf SVM and output the resultant model (Lines 9-10).

### B. Effectiveness Prediction

The discriminative model learned in the model learning phase would be able to predict if an unknown instance (i.e., a fault localization instance whose effectiveness is unknown) is effective or not. The unknown instance needs to be transformed to a set of feature values using the feature extraction process described in Section IV. These feature values (aka. a feature vector) are then compared with the model and a prediction would be output. The feature vector is compared with the hyperplane that separates effective

<sup>1</sup><http://svmlight.joachims.org/>

**Procedure SVM<sup>Ext</sup>****Inputs:***EI*: Effective fault localization instances*II*: Ineffective fault localization instances**Output:** Discriminative model**Method:**

- 1: If ( $|EI| < |II|$ )
- 2: Let  $S_i^j$  = Similarity between  $EI[i]$  (i.e., the  $i^{th}$  effective instance) with  $II[j]$  (i.e., the  $j^{th}$  ineffective instance)
- 3: Let  $M_i = \text{Max}_{j \in \{0, \dots, |II|-1\}} S_i^j$
- 4: Let  $MOSTSIM$  = Sorted  $EI$  (sorted in descending order of  $M_i$ )
- 5: Let  $idx = 0$
- 6: While( $|EI| < |II|$ )
- 7: Add  $MOSTSIM[idx \% |MOSTSIM|]$  to  $EI$
- 8:  $idx++$
- 9: Let  $Model$  = Model learned with off-the-shelf SVM with  $EI$  and  $II$  as training data
- 10: **Output**  $Model$

Figure 3. SVM<sup>Ext</sup>

and ineffective training instances. Based on which side of the hyperplane the feature vector lies, the corresponding unknown instance is assigned either effective or ineffective prediction label.

## VI. EXPERIMENTS

In this section we first describe our dataset, followed by our evaluation metrics, research questions, and results.

## A. Dataset

We analyze 10 different programs. These include NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [17]. These programs have been widely used in past studies on fault localization and thus could collectively be used as a benchmark [19], [27], [22], [1]. Table III provides the details on the programs.

NanoXML is an XML parsing utility written in Java. We download NanoXML from Software Infrastructure Repository (SIR) [8]. SIR contains 5 variants of NanoXML: NanoXML\_v1, NanoXML\_v2, NanoXML\_v3, NanoXML\_v4, and NanoXML\_v5. Each of the variants contains faulty versions except NanoXML\_v4. We downloaded all 32 faulty versions of these variants. We exclude two of the faulty versions as there are no failure-inducing test cases that expose the bugs. Thus, for NanoXML, in total, we analyze 30 faulty versions. XML-Security is a digital signature and encryption library written in Java. There are 3 variants of XML-Security in SIR: XMLSec\_v1, XMLSec\_v2, and XMLSec\_v3. For each variant, several faulty versions are provided. In total, we downloaded 52 faulty versions from these variants; we analyze 16 of them, as there are no failure-inducing test cases that expose the other bugs. Space was used in European Space Agency and is an interpreter for Array Definition Language (ADL) written in C. All 35 faulty versions of Space downloaded from SIR are used for our experiments. For these 3 programs, in total we analyze, 81 faulty versions.

Table III

DATASET DESCRIPTIONS: NAME, LINES OF CODE, PROG. LANGUAGE, NUMBER OF FAULTY VERSIONS, AND NUMBER OF TEST CASES.

Dataset	LOC	Language	# Faulty	# Tests
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6,218	C	35	13,585
NanoXML v1	3,497	Java	6	214
NanoXML v2	4,007	Java	7	214
NanoXML v3	4,608	Java	9	216
NanoXML v5	4,782	Java	8	216
XML security v1	21,613	Java	6	92
XML security v2	22,318	Java	6	94
XML security v3	19,895	Java	4	84

Siemens programs are originally created for a study on test coverage adequacy performed by researchers from Siemens Corporation Research [17]. Each of the seven programs has many faulty versions derived “by seeding realistic faults” [17]. Each faulty version contains one bug that may span more than one program element (i.e., basic block). It comes with test cases and bug free versions. Siemens programs have been used in many fault localization studies including [19], [27], [22], [1]. The Siemens test suite<sup>2</sup> include the following programs: print\_tokens, print\_tokens2, replace, schedule, schedule2, tcas, and tot\_info. There are a total of 132 versions in the test suite. We instrumented each blocks in the versions. We exclude versions that are seeded by bugs residing in variable declarations as our instrumentation cannot reach these declarations. Thus, we exclude the following versions: version 12 of replace, versions 13, 14, 15, 36, 38 of tcas, and versions 6, 10, 19, 21 of tot\_info. Versions 4 and 6 of print\_token are also excluded because they are identical with the bug free version. We exclude version 9 of schedule2 as running all test cases only produces correct executions – no test case is a failure-inducing one. In total, we include 119 faulty versions from Siemens test suite for our experiment. Adding the 81 faulty versions from the 3 other programs, we have in total 200 faulty versions.

## B. Evaluation Metrics &amp; Experiment Settings

We evaluate the accuracy of our solution in terms of precision, recall, and F-measure. These metrics have been frequently used to evaluate various prediction engines [13]. We first define the concepts of true positives, false positives, true negatives, and false negatives:

True Positives (TP): Number of effective fault localization instances that are predicted correctly

False Positives (FP): Number of ineffective fault localization instances that are predicted wrongly

<sup>2</sup>We use the variant at: [www.cc.gatech.edu/aristotle/Tools/subjects](http://www.cc.gatech.edu/aristotle/Tools/subjects)

True Negatives (TN): Number of ineffective fault localization instances that are predicted correctly

False Negatives (FN): Number of effective fault localization instances that are predicted wrongly

Based on the above concepts, we can define precision, recall, and F-measure as follows:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

$$F-Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (3)$$

There is often a trade-off between precision and recall. Higher precision often results in lower recall (and vice versa). To capture whether an increase in precision (or recall) outweighs a reduction in recall (or precision), F-measure is often used. F-measure is the harmonic mean of precision and recall and it combines the two measures together into a single summary measure.

We perform ten-fold cross validation to evaluate the effectiveness of our proposed approach. Ten-fold cross validation is a standard approach in data mining to estimate the accuracy of a prediction engine [13]. Its goal is to assess how the result of a prediction engine generalizes to an independent test data. In ten-fold cross validation, we divide the dataset into ten groups. We use nine of the groups for training and one of the groups for testing. We repeat the process 10 times using different groups as the test group. We aggregate all the results and compute the final precision, recall, and F-measure.

### C. Research Questions

We would like to answer the following research questions. The research questions capture different aspects that measure how good our proposed approach is.

**RQ1.** How effective is our approach in predicting the effectiveness of a state-of-the-art spectrum-based fault localization tool ?

We evaluate the accuracy of our tool in predicting the effectiveness of Tarantula which has been demonstrated to be one of the most accurate fault localization tools.

**RQ2.** How effective is our extended Support Vector Machine (SVM<sup>Ext</sup>) compared with off-the-shelf Support Vector Machine (SVM) ?

To learn a discriminative model, we extend SVM to address the data imbalance issue. We would like to investigate if this extension is necessary to make our framework work.

**RQ3.** What are some important features that help in discriminating if a fault localization tool would be effective given a set of input traces ?

We investigate which of the 50 features that we use are more dominant and thus more effective to help us achieve higher prediction accuracy. In the machine learning community, Fisher score is often used to measure how dominant or discriminative a feature is [9], [11]. We compute the Fisher score of every feature as follows:

$$FS(j) = \frac{\sum_{class=1}^{\#class} (\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class} \left( \frac{1}{n_{class}-1} \sum_{i=1}^{n_{class}} (x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2 \right)}$$

In the equation,  $FS(j)$  denotes the Fisher score of the  $j^{th}$  feature.  $n_{class}$  is the numbers of data points (i.e., fault localization instances) with label  $class$  (i.e., effective or ineffective).  $\bar{x}_j$  denotes the average value of the  $j^{th}$  feature of all data points.  $\bar{x}_j^{(class)}$  is the average value of the  $j^{th}$  feature of  $class$ -labeled data points.  $x_{i,j}^{(class)}$  denotes the value of the  $j^{th}$  feature of the  $i^{th}$   $class$ -labeled data point. Fisher score ranges from 0 to 1. A Fisher score of 0 indicates that a feature is not discriminative, while a Fisher score of 1 indicates that a feature is very discriminative.

**RQ4.** Could our approach be used to predict the effectiveness of different types of spectrum-based fault localization tool ?

There are different spectrum-based fault localization tools proposed in the literature. We would like to investigate if our approach also works for different spectrum-based fault localization tools. We consider two other well known spectrum-based fault localization tools: Ochiai [1], and Information Gain [24].

**RQ5.** How sensitive is our approach to the amount of training data ?

We use ten-fold cross validation to evaluate our approach. In ten-fold cross validation, we use 90% of the data for training, and the remaining 10% for testing. In this research question, we investigate the impact of reducing the number of training data on the accuracy of the proposed approach.

**RQ6.** Could data from one software program be used to train a discriminative model used to predict effectiveness of a fault localization tool on failures from other software programs ?

To answer this research question, we use data from N-1 (i.e., 9) software programs to build a model. This model is then used to predict the effectiveness of a fault localization tool on the remaining one software program. We refer to this process as N-fold cross-program validation.

### D. Results

In this section, we answer our research questions one at a time by performing a set of experiments. For all research questions except RQ2, we use the default setting of our proposed framework presented in previous sections.

1) *RQ1: Overall Accuracy*: To answer our first research questions, we simply run Tarantula on the 200 faulty versions. We then predict if Tarantula is effective or not for each of the 200 faulty versions using SVM<sup>Ext</sup>. We perform ten-fold cross validation and aggregate the result for the final precision, recall, and F-measure. For Tarantula, 85 of the localization instances are effective and 115 of the instances are ineffective. Thus, the data is imbalanced.

The result of our experiment is shown in Table IV. The result shows that we can achieve a precision of 54.36%. This means that we can correctly identify *many* ineffective fault localization instances (i.e., 47 out of the 115 ineffective instances). We can also achieve a recall of 95.29%. This means that we correctly identify *almost all* effective instances (i.e., 81 out of the 85 effective instances). F-measure, the harmonic mean of precision and recall, is often used to gauge on how effective a prediction engine is. Our F-measure is 69.23%. Comparing with many other studies performing other prediction tasks in software engineering research literature, e.g., [31], [32], our F-measure is comparable or higher.

Table IV  
PRECISION, RECALL, AND F-MEASURE OF OUR PROPOSED APPROACH

<b>Precision</b>	54.36%
<b>Recall</b>	95.29%
<b>F-Measure</b>	69.23%

2) *RQ2: SVM<sup>Ext</sup> vs. SVM*: Next, we compare our extended SVM (SVM<sup>Ext</sup>) with standard off-the-shelf SVM. The precision, recall, and F-measure of using SVM<sup>Ext</sup> and SVM is shown in Table V. SVM<sup>Ext</sup> clearly outperforms SVM with respect to precision, recall, and F-measure. We also compute the relative improvement of SVM<sup>Ext</sup> over SVM by the following formula:

$$\text{Relative Improvement} = \frac{(SVM^{Ext} \text{ Result} - SVM \text{ Result})}{SVM \text{ Result}}$$

We find that SVM<sup>Ext</sup> outperforms SVM in terms of precision, recall, and F-Measure by 6.50%, 65.29%, and 27.87% respectively. SVM is not able to handle imbalanced data. The imbalanced data causes SVM to predict more unknown instances with the majority label that it sees in the training data (i.e., ineffective). This reduces the number of true positives and increases the number of false negatives, which causes a significant reduction in recall.

Table V  
PRECISION, RECALL, AND F-MEASURE OF SVM<sup>Ext</sup> AND SVM

	SVM <sup>Ext</sup>	SVM	Relative Improve.
<b>Precision</b>	54.36%	51.04%	6.50%
<b>Recall</b>	95.29%	57.65%	65.29%
<b>F-Measure</b>	69.23%	54.14%	27.87%

3) *RQ3: Important Features*: Next, we investigate which features are important. We use Fisher score to rank the features. Table VI shows the list of top-10 most important

features. Interestingly, we find that the top-10 features include input and output features. Both input execution traces and suspiciousness scores generated by a fault localization tool are important to predict the effectiveness of a fault localization instance.

Relative-difference features, i.e., C7, C8, C6, C5, and C1, are the most discriminative (5 out of the top-10 features). These features can capture a “break” or gap in the top-10 discriminative scores. This “break” signifies that the fault localization tool is able to differentiate some program elements to be significantly more suspicious than the others. Three of the top-10 features are related to program elements, i.e., PE1, PE2, and PE4. They capture the number of program elements covered in execution traces. The more program elements are covered, the harder it is to get effective fault localization as the fault localization tool needs to differentiate more program elements to find the root cause. The other two of the top-10 features are the highest suspiciousness score (R1) and the number of distinct suspiciousness scores in the top-10 scores (SS1). These are intuitively related to fault localization effectiveness: the higher a suspiciousness score is, the more likely a program element is the root cause; the more the number of distinct suspiciousness scores, the more that a fault localization tool differentiates program elements.

Table VI  
TOP-10 MOST DISCRIMINATIVE FEATURES<sup>2</sup>

Rank	Feature	Rank	Feature
1	C7	6	SS1
2	C8	7	C5
3	C6	8	C1
4	PE1	9	PE4
5	PE2	10	R1

4) *RQ4: Different Fault Localization Tools*: We also investigate if our approach could be generalized to other spectrum-based fault localization tools aside from Tarantula. We use the same set of 200 faulty versions and perform the same ten-fold cross validation using SVM<sup>Ext</sup> to evaluate two other spectrum-based fault localization tools: Ochiai [1], and Information Gain [24]. Table VII shows the precision, recall, and F-measure when we predict the effectiveness of Tarantula, Ochiai, and Information Gain.

We note that a similar precision, recall, and F-measure can be achieved for predicting the effectiveness of Ochiai and Information Gain. Our framework can achieve an F-measure of more than 75% for Ochiai and Information Gain. This is higher than the accuracy of our framework for Tarantula.

Table VII  
PRECISION, RECALL, AND F-MEASURE FOR VARIOUS FAULT LOCALIZATION TOOLS

Tool	Precision	Recall	F-Measure
Tarantula	54.36%	95.29%	69.23%
Ochiai	63.23%	97.03%	76.56%
Information Gain	64.47%	93.33%	76.26%

<sup>2</sup>Please refer to Table II for the description of the features.

5) *RQ5: Different Amount of Training Data:* In ten-fold cross validation, we use 90% of the data for training on only 10% for testing. To answer this research question, we vary the amount of training data from 10% to 90% and show the resultant precision, recall, and F-measure. We randomly pick the data that we use for training. We show the result in Table VIII. Note that as we randomly resample the 90% data, the result is different with that of RQ1. We find that the performance of our framework does not degrade too much (F-measure > 60%) if there is sufficient data for training (30-90%), the performance degrades significantly if there is too little training data (10-20%).

Table VIII  
PRECISION, RECALL, AND F-MEASURE FOR VARIOUS AMOUNT OF TRAINING DATA

Amount of Data	Precision	Recall	F-Measure
90%	61.54%	100.00%	76.19%
80%	51.52%	100.00%	68.00%
70%	58.14%	100.00%	73.53%
60%	50.77%	97.06%	66.67%
50%	53.33%	95.24%	68.38%
40%	51.02%	98.04%	67.11%
30%	46.77%	98.31%	63.39%
20%	55.56%	36.76%	44.25%
10%	48.78%	26.32%	34.19%

6) *RQ6: Cross-Program Setting:* We perform N-fold cross-program validation to answer this research question. The result is shown in Table IX. The result shows that our approach could be used in cross-program setting with an F-measure of 63%, which is lower than our result for RQ1 (i.e., 69.23%). This is as expected as the programs are diverse and each program might have its own characteristics. It is thus harder to predict fault localization effectiveness for one program using training data from other programs.

Table IX  
PRECISION, RECALL, AND F-MEASURE IN CROSS-PROGRAM SETTING

<b>Precision</b>	46.4%
<b>Recall</b>	100.00%
<b>F-Measure</b>	63.43%

### E. Threats to Validity

We consider three kinds of threats to validity: internal, external, and constructing validity. Threats to internal validity corresponds to experimenter bias. In our experiments, we use the programs that are manually instrumented by Lucia et al. [24]. Due to the manual instrumentation process, there might be some basic blocks that are missed (i.e., no instrumentation code is added for them). Threats to external validity corresponds to the generalizability of our findings. In this study, we have analyzed 10 different programs. These programs are widely studied in past fault localization studies and thus collectively they can be used as a benchmark. We have also analyzed programs written in two programming languages: C and Java. Still, more programs can be analyzed to reduce the threat further. We plan to do this in a future

work. Threats to construct validity corresponds to the suitability of our metrics. We use standard metrics of precision, recall, and F-measure. These are well known metrics in data mining, machine learning, and information retrieval and have been used in many past studies in software engineering, e.g., [16], [25], [2]. Thus with respect to these metrics, we believe there is little threat to construct validity. Another threat to construct validity is our definition of effective fault localization instance. In this preliminary study, we consider an instance is effective if at least one of the root cause is in the top-10 most suspicious program elements. Other definitions of effective fault localization could be considered, e.g., the root cause must be in the top-1 most suspicious program elements for an instance to be effective, etc. We leave the consideration of other definitions of effective fault localization for future work.

## VII. RELATED WORK

In this section, we highlight a number of studies in *spectrum-based* fault localization which analyze program traces or their abstractions which capture the runtime behaviors of program.

Many spectrum-based fault localizations studies analyze two sets of program spectra: one set corresponding to correct executions, and another set corresponding to faulty executions [19], [1], [40], [21], [22], [30], [6], [23], [10], [20], [24], [3]. Based on these inputs, these studies would typically compute likelihood of different program elements to be the root cause of the faulty executions (aka. failures). Jones and Harrold propose Tarantula that computes the suspiciousness scores of various program elements by following this intuition: program elements that are executed more frequently by faulty executions rather than correct executions are deemed to be more suspicious [19]. Abreu et al. propose a different formula to compute suspiciousness scores [1]. They show that their proposed formula named Ochiai is able to outperform Tarantula. Zeller proposes Delta Debugging which compares a faulty execution and a correct execution and find the minimum state differences [40]. Liblit et al. compute predicates whose true evaluation correlates with failures [21]. This work is extended by Chao et al. which propose a work, named SOBER, that considers the repeated outcomes of predicate evaluations in a program run [22]. Santelices et al. use multiple program spectra to localize faults [30]. Cheng et al. propose an approach to mine a graph-based signatures, referred to as bug signatures, that differentiates correct from faulty executions [6]. Lo et al. extend the work of Cheng et al. by minimizing signatures and fusing minimized signatures to capture the context of program errors better [23]. Gong et al. after that propose a test case prioritization technique to reduce the number of test cases with known oracles for fault localization [10]. Gong et al. propose interactive fault localization where a fault localization tool iteratively updates its recommendation

as it receives feedback from end users [20]. Lucia et al. investigate many association measures and adapt them for fault localization [24]. They find that information gain performs the best. Wang et al. employ search-based algorithms to combine various association measures and existing fault localization algorithms [38]. Artzi et al. use test generation for fault localization [3].

Other spectrum-based fault localizations analyze only one set of program spectra, i.e., faulty executions [41], [12], [18]. These techniques typically modify program runtime states systematically to localize faulty program elements. In this work, we focus on fault localization tools that compare correct and faulty executions.

## VIII. CONCLUSION AND FUTURE WORK

In this study, to address the unreliability of fault localization tool, we build an oracle that can predict the effectiveness of a fault localization tool on a set of execution traces. We propose 50 features that can capture interesting dimensions that potentially differentiate effective from ineffective fault localization instances. Values of these features from a training set of faulty localization instances can be used to build a discriminative model using machine learning. This model is then used to predict if unknown instances are effective or not. We have evaluated our solution on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. Our solution can achieve a precision, recall, and F-measure of 54.36%, 95.29%, and 69.23%, respectively. We have also tested different aspects of our solution including its ability to handle cross-program setting and the results are promising.

As future work, we plan to improve the precision and F-measure of our proposed approach further. We plan to perform an in-depth analysis of cases where our proposed approach is less effective and design appropriate extension to the approach. We would also like to extend our approach to predict the effectiveness of other fault localization techniques, e.g., [6], [38], [30], [7]. We also plan to investigate the effectiveness of and incorporate some findings from recent studies on learning from imbalanced data performed in the data mining community [15] to further improve our  $SVM^{Ext}$ . It is also interesting to leverage other information aside from execution traces; some failures come with textual descriptions [42], and it would be interesting to employ advanced text mining solutions [5], [39] to identify whether fault localization tools would be effective on such failures.

## REFERENCES

- [1] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.
- [2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE*, 2006, pp. 361–370.
- [3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *ISSTA*, 2010.
- [4] B. Beizer, *Software Testing Techniques*, 2nd ed. Boston: International Thomson Computer Press, 1990.
- [5] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet allocation," *J. Machine Learning Research*, vol. 3, pp. 993–1022, 2003.
- [6] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009.
- [7] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.
- [8] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.
- [9] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley-Interscience Publication, 2001.
- [10] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *ASE*, 2012, pp. 30–39.
- [11] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," in *UAI*, 2011, pp. 266–273.
- [12] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE*, 2005, pp. 263–272.
- [13] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.
- [14] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, 2000.
- [15] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, 2009.
- [16] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," in *ASE*, 2011.
- [17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of ICSE*, 1994, pp. 191–200.
- [18] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*, 2008.
- [19] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.
- [20] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang, "Interactive fault localization leveraging simple user feedback," in *ICSM*, 2012, pp. 67–76.
- [21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003, pp. 141–154.
- [22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005.
- [23] D. Lo, H. Cheng, and X. Wang, "Bug signature minimization and fusion," in *HASE*, 2011, pp. 340–347.
- [24] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
- [25] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *ASE*, 2012.
- [26] C. Parmin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011, pp. 199–209.
- [27] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 141–154.
- [28] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ESEC/FSE*, 1997.
- [29] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [30] R. Santelices, J. Jones, Y. Yu, and M. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE*, 2009.
- [31] H. Seo and S. Kim, "Predicting recurring crash stacks," in *ASE*, 2012, pp. 180–189.
- [32] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, 2012.
- [33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE (1)*, 2010.
- [34] G. Tasse, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.
- [35] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCSE*, 2012.
- [36] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *CSMR*, 2012, pp. 385–390.
- [37] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed. Springer-Verlag, 2000.
- [38] S. Wang, D. Lo, and L. Jiang, "Search-based fault localization," in *ASE*, 2011.
- [39] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009.
- [40] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.
- [41] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006.
- [42] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012.

# Enhancing Software Traceability By Automatically Expanding Corpora With Relevant Documentation

Tathagata Dasgupta, Mark Grechanik  
 University of Illinois at Chicago  
 Chicago, IL 60607-7053  
 {tdasgu2,drmark}@uic.edu

Evan Moritz, Bogdan Dit, Denys Poshyvanyk  
 College of William and Mary  
 Williamsburg, VA 23185  
 {eamoritz,bdit,denys}@cs.wm.edu

**Abstract**—Software traceability is the ability to describe and follow the life of a requirement in both a forward and backward direction by defining relationships to related development artifacts. A plethora of different traceability recovery approaches use information retrieval techniques, which depend on the quality of the textual information in requirements and software artifacts. Not only is it important that stakeholders use meaningful names in these artifacts, but also it is crucial that the same names are used to specify the same concepts in different artifacts. Unfortunately, the latter is difficult to enforce and as a result, software traceability approaches are not as efficient and effective as they could be – to the point where it is questionable whether the anticipated economic and quality benefits were indeed achieved.

We propose a novel and automatic approach for expanding corpora with relevant documentation that is obtained using external function call documentation and sets of relevant words, which we implemented in TraceLab. We experimented with three Java applications and we show that using our approach the precision of recovering traceability links was increased by up to 31% in the best case and by approximately 9% on average.

**Keywords**—software traceability; machine learning; API call;

## I. INTRODUCTION

Software traceability is the ability to describe and follow the life of a requirement in both a forward and backward direction by defining relationships to related development artifacts [29]. Recovering *traceability links (TLs)* (or *traces*) between requirements and software artifacts automatically and with high precision has a significant economic impact, since TLs improve the quality of different software maintenance tasks by enabling stakeholders to reveal errors and ambiguities in requirements and to ensure that all requirements are implemented and tested. Software traceability is especially useful when it connects concepts from the problem domain (where requirements are expressed using vague objectives or wish lists) to the solution domain (i.e., the domain in which engineers use their ingenuity to solve problems [33, pages 87,109]), e.g., the source code is a solution domain. When creating TLs, stakeholders map high-level intent reflected in the problem domain to low-level implementation details in the solution domain thereby solving an instance of the concept assignment problem [6]. It is highly desirable that programmers who implement specific requirements work with other stakeholders to record traces between requirements and their implementations using traceability matrices. While this is an ideal method, which likely leads to higher precision, it contributes to the high cost of software [25]. In addition,

TLs are often recorded after software is implemented to avoid interruption of software development, which makes the task of recovering TLs approximate and error-prone.

Automating software traceability is a big and important problem. Delegating the task of computing traces between different artifacts to an automated tool saves cost and improves various software maintenance efforts. Even with manual traceability, some traces are erroneous, and validating these traces also requires a large investment. It is reported that less than 47% of Fortune 500 companies made their business requirements traceable and well integrated into testing, making software traceability as one of the biggest problems of software engineering [15].

A plethora of different *information retrieval (IR)* approaches for recovering TLs automatically depend on the quality of textual information in requirements and software artifacts. Not only is it important that stakeholders use meaningful names in these requirements and artifacts, but it is also equally important that the same names are used to specify the same concepts in different requirements documents and artifacts. Unfortunately, the latter is difficult to enforce, and as a result, software traceability approaches are not as efficient and effective as they could be – to the point where it was questionable whether the anticipated economic and quality benefits were indeed achieved from traceability approaches.

A fundamental problem of using IR approaches for software traceability is the mismatch in words that are used in requirements and software artifacts to describe high-level concepts. Words are fundamental blocks for existing traceability approaches to compute similarities between artifacts, and subsequently TLs, by matching words in these artifacts (e.g., words in requirements documents, comments in the source code, or the names of program variables and types). If no match is found, then potentially correct TLs are never recovered. This situation is aggravated by the fact that many applications are developed by large teams where different artifacts are created and maintained by different stakeholders; matches between words that designate the same concepts are not guaranteed.

Moreover, programmers routinely use *Application Programming Interface (API)* calls from third-party vendors to encode specific implementations of high-level concepts. For example, encountering an instance of the class `DESKeySpec` in the source code will undoubtedly lead a stakeholder to infer that a requirement for encrypting sensitive data is implemented

in the fragment of the code that uses this instance, even though the class name does not match any words in the requirements documentation. Of course, the stakeholder uses the information that comes from the descriptions of the API calls to obtain a high-level concept and match it to requirements. This is where our intuition lies: *using relevant external documentation to expand the corpora of different artifacts may lead to uncovering more TLs with a higher degree of precision.*

We observed that in industry many computer professionals intuitively attempted to implement this approach in ad-hoc ways. Some tried to merge corpora that they obtained from different applications that belong to the same domain; others mixed different related corpora to increase the probability of retrieving correct TLs using different IR approaches [5]. Even though stories that describe successful results are disseminated about the effectiveness of these approaches, it is not clear how scientifically valid these results are. In addition, expanding corpora manually is an unsystematic and laborious effort whose precision depends on how similar merged corpora are perceived by stakeholders.

We propose an automatic approach for *ENhancing TRAcability usiNg API Calls and rElevant woRds (ENTRANCER)*. The input to ENTRANCER is program source code, requirements documents, and other artifacts, which we collectively call the *base corpora*, and the external information sources from which we enhance the corpora. These sources consist of documentation for third-party API calls (e.g., the Java Development Kit (JDK)<sup>1</sup>) and external lexical databases with sets of cognitive synonyms (e.g., Wordnet<sup>2</sup> synsets), which we call the *enhancing corpora*. The core idea of ENTRANCER is to automatically expand words in the base corpora with semantically related words from the enhancing corpora to recover TLs or traces among requirements documents and other artifacts with a high degree of precision. To the best of our knowledge, this is the first comprehensive study of an automated approach that expands the base multilingual corpora, i.e., Italian and English.

We evaluated ENTRANCER on three open-source Java applications using the TraceLab experimental framework and obtained results that suggest our approach is effective. We showed that ENTRANCER can increase the precision of the recovering TLs by up to 31% in the best case. All data files and subject applications are available from the project website <http://www.cs.uic.edu/~drmark/entrancer.htm>.

## II. OUR APPROACH

In this section, we explain the theory of relevance behind our approach, formulate our hypotheses, and describe the experimental framework TraceLab.

### A. Software Traceability Is A Similarity Measure

Suppose that a user is given two photographs and asked to establish traces between different features on these images. Naturally, a user looks for similarities among different features, and depending on the resolution of the image, some features may be blurry or collapsed into just a few pixels. By increasing

the resolution and adding more pixels to features, the user can find similarities among these features much faster.

Finding similar features on two images is analogous to finding TLs by measuring similarities between components in the source code and requirements. Many automated traceability approaches imitate stakeholders by measuring similarities between elements in the problem and solution domains (e.g., how closely words in a requirement paragraph match words in comments and identifier names of the source code). When similarity between artifacts is computed using an IR approach, and this similarity is above a user-defined threshold, a TL is recorded between these artifacts. A straightforward approach for measuring similarities between requirements and software artifacts is to count matches among words from requirements to source code identifiers (e.g., names of variables, types, etc.). Computing traces with high precision depends on the quality of information in artifacts, specifically, on programmers choosing meaningful names that reflect correctly the concepts or abstractions that they implement, but this compliance is generally difficult to enforce [2][39].

We define software traceability as the similarity between artifacts by using Mizzaro's well-established conceptual framework for relevance [40, 41]. In Mizzaro's framework, similar artifacts are relevant to one another if they share some common concepts. Once these concepts are known, a corpus of artifacts can be clustered by how they are relevant to these concepts. Subsequently all artifacts in each cluster will be more relevant to one another when compared to artifacts that belong to different clusters. This is the essence of the cluster hypothesis that specifies that artifacts that cluster together tend to be relevant to the same concept [48][39], and consequently traceability links are established between these artifacts.

### B. Our Hypotheses

A requirement is relevant to some software artifact if this artifact implements the same abstraction that is specified in this requirement [39]. For example, if a requirement specifies that cryptographic services should be used to protect information, and a module in an application uses encryption, then these requirement and software artifacts are relevant to a certain degree. Currently, most IR approaches use artifacts as "bags of words" with no semantics, and the relevancy or similarity of these artifacts to one another can be determined by matches between these words. This is the essence of *syntagmatic associations*, where artifacts are considered similar (i.e., traced to each other) when terms (i.e., words) in these documents occur together [45]. The problem with this approach is that computed traceability links are relatively imprecise when compared with ENTRANCER (as we show in Section IV).

Syntagmatic associations are used in a variety of techniques for computing TLs, such as *Vector Space Model (VSM)*, where artifacts are represented as vectors of words and a similarity measure is computed as the cosine between these vectors [46]. One main problem with VSM is that different programmers can use the same words to describe different requirements (i.e., the *synonymy* problem) and they can use different words to describe the same requirements (i.e., the *polysemy* problem). This problem is a variation of the vocabulary problem, which states that "no single word can be chosen to describe a

<sup>1</sup><http://docs.oracle.com/javase/6/docs/api/index-files/index-1.html>

<sup>2</sup><http://wordnet.princeton.edu>

programming concept in the best way” [26]. This problem is general to IR, but somewhat mitigated by the fact that different programmers who participate in the projects use coherent vocabularies to write code and documentation, thus increasing the chance that two words in different applications may describe the same requirement.

Our **first hypothesis** is that *it is possible to increase the precision of IR approaches that are based on syntagmatic associations by expanding the vocabulary of artifacts using related words*. Consider the requirement that information should be protected from unauthorized use. Suppose that a software artifact contains the variable “encrypt”, which is syntactically different from the word “protect”, which makes it difficult to establish the match. However, since these words co-occur in many documents, we can enhance the vocabularies of both requirements document and software artifacts by appending both words to the corpora obtained from these requirements documents and software artifacts. As a result, a match will be found and a TL will be established.

In Mizzaro’s framework, a key characteristic of relevance is how information is represented in artifacts. We concentrate on *semantic anchors*, which are elements of artifacts that precisely define the artifacts’ semantic characteristics [39]. Semantic anchors may take many forms (e.g., they can refer to elements of semantic ontologies that are precisely defined and agreed upon by different stakeholders). This is the essence of *paradigmatic associations* where artifacts are considered similar if they contain terms with high semantic similarities [45]. While paradigmatic associations are considered more reliable than syntagmatic ones, both can be useful when computing TLs between requirements and software artifacts.

Our **second hypothesis** is that *by using semantic anchors it is possible to compute similarities between requirements and software artifacts with a higher degree of accuracy*. Our idea is based on the observation that in the solution domain engineers go into implementation details to realize requirements (i.e., engineers look for reusable abstractions that are often implemented using third-party API calls). Since programs contain API calls with precisely defined semantics, these API calls can serve as semantic anchors to compute the degree of similarity between requirements and artifacts by matching the semantics of these applications that are expressed with these API calls. Programmers routinely use third-party API calls (e.g., the *Java Development Kit (JDK)*) to implement various requirements [10, 21, 30, 31, 47]. API calls from well-known and widely used libraries have precisely defined semantics—unlike names of program variables, types, and words that programmers use in comments. Since these documentations are written by different people who have different vocabularies, appending words from these documentations to the corpora makes a richer vocabulary. In this paper, we use the JDK API calls as semantic anchors to compute similarities among applications by expanding these API calls with words from their documentations, thereby partially addressing the vocabulary problem.

Finally, our **third hypothesis** is that *by using the hybrid of syntagmatic and paradigmatic vocabulary expansion we can increase the precision of computing TLs*. Our rationale for this hypothesis is that combining both approaches expands the vocabulary and increases the precision of IR approaches for traceability.

### C. Experimental Testbed – TraceLab

TraceLab [12, 35, 14] is a software infrastructure designed to address the issue related to the reproducibility of experiments (i.e., lack of implementation, implementation details, datasets, etc.) in software engineering (SE) research.

TraceLab provides (i) a set of predefined components (i.e., tools that are commonly used in SE techniques and approaches, such as data importers, preprocessors, IR approaches, state of the art TL recovery techniques, etc.), as well as (ii) a development kit which includes the guidelines and support for creating custom components. These components can be assembled to create experiments, which can be executed alongside other SE techniques, on the same datasets, and their results can be compared to determine which technique produces the best results using standard metrics (e.g., precision, recall, etc.), as well as statistical tests. In addition, the newly created experiments, which are fully reproducible, are shared with the community in order to facilitate the creation of new techniques (based on the existing one) and the comparison of new techniques against existing ones.

TraceLab was funded by the National Science Foundation and was developed at DePaul University in collaboration with Kent State University, University of Kentucky, and the College of William and Mary. Since its introduction, TraceLab has already been successfully used in several projects [43, 23, 24, 16].

## III. EXPERIMENTAL DESIGN

In this section, we describe the experimental design for evaluating ENTRANCER.

### A. Research Questions

In this paper, we claim that we can improve the precision of computing software traceability using similarity-based methods by expanding the corpus obtained from application’s artifacts with relevant documentation. We seek to answer the following Research Questions (RQs).

- $RQ_1$  Does using expansion of the corpus with documentation from JDK API calls result in a higher precision of recovering TLs?
- $RQ_2$  Does using expansion of the corpus with a combined documentation from JDK API calls and Wordnet result in a higher precision of recovering TLs?
- $RQ_3$  Does including words from comments result in a higher precision of recovering TLs when expanding the corpus with a combined documentation from the JDK API calls and Wordnet?
- $RQ_4$  How does the size of the corpus affect the precision of recovering TLs?
- $RQ_5$  Is ENTRANCER equally effective using different IR approaches for recovering software TLs?

With these RQs, we decompose our experimental results to evaluate the effectiveness of ENTRANCER for different components of software traceability. In all cases, we start with the initial corpus that contains words from requirements documents on one side and words from the source code of a subject application on the other side.

With  $RQ_1$ , we investigate a claim that replacing JDK method occurrences in initial source code corpus with words that appear in the method description in Java API Specification, increases correct TL recovery. Our rationale is the following: API calls allow programmers to express abstraction of high-level concepts from requirement documentation thus reducing the chance of IR methods matching relevant words in the source code corpus. Hence, if every method invocation is replaced with its description, it introduces more relevant words in the corpus which in turn should lead to more recovered correct TLs. The rationale for  $RQ_2$  is similar. We use the Italian WordNet called MultiWordnet<sup>3</sup>, to include more relevant words by including synsets of dictionary words existing in the corpus, since the subject applications are written by Italian programmers who wrote comments and identifier names in Italian.

Unlike rest of the source code, comments are written in natural language by the programmers and as such should contain relevant dictionary words and phrases expressing high-level intents. So in  $RQ_3$ , we inspect how significant is the impact of source code comments on traceability.

To answer  $RQ_4$ , we will consider a correlation between the size of the corpus and the precision of computing TLs when applying different IR methods that we describe in Section III-D1. Since having more words in corpus increases the probability that more correct matches may be obtained between these words and words in requirements documents, the rationale for  $RQ_4$  is to establish if the size of the corpus alone may be indicative of the future quality of obtained TLs.

Finally, our claim is that we designed and implemented a methodology as part of ENTRANCER for achieving a higher precision for software traceability by expanding the corpora of software applications with relevant documentation. The rationale for  $RQ_5$  is to evaluate results to determine if ENTRANCER is effective when compared across different IR-based traceability approaches.

### B. Subject Software Applications

The subject Java applications that we used to evaluate different traceability approaches are publicly available for researchers from the TraceLab web site<sup>4</sup>. These applications have been used in other studies, making it easy to reproduce our results and compare them with other approaches.

**Albergate**, the first evaluated application, is a software system designed to implement all the operations required to administrate and manage a small/medium size hotel (room reservation, bill calculation, etc.). It was developed from scratch by a team of final year students at the University of Verona (Italy) on the basis of 16 functional requirements expressed (as well as all the other system documentation) in the Italian language. Albergate exploits a relational database and consists of 13 requirements documents, 95 classes and about 20 KLOC [4]. Albergate has been used in different traceability studies [38]. **eTour** is an electronic tourist guide developed by students. It has 58 use cases and a total of 174 Java classes with 366 recorded traceability links. Finally, **SMOS** is an

application that is used to monitor high school students (e.g., absence, grades). It has 67 use cases and a total of 100 Java classes with 1,044 recorded TLs.

### C. Preprocessing Source Code Using Identifier Splitting

High quality identifier splitting is very important for achieving good precision for software traceability approaches. Low quality identifier splitting results in words that are not in a dictionary or they incorrectly represent the semantic meaning that they should otherwise convey.

The source code of the three subject applications, was written by Italian developers, and they contain a combination of both Italian and English words in identifier construction. This prevents us from using any identifier splitting method that relies primarily on heuristics derived from mining source code repository. Accordingly, we built a fast, regular expression based identifier splitting that uses camel case and Java identifier naming convention to split each identifier into their separate English/Italian words.

### D. Methodology

Our hypotheses are partially based on our idea that it is better to compute similarity-based traceability by utilizing relevant words from Wordnet and API calls as semantic anchors that come from the JDK and that programmers use to implement various requirements. To evaluate ENTRANCER, we carry out experiments to explore its effectiveness and to answer RQs.

*1) Independent Variables:* In our experimental design, we consider two types of independent variables: IR approaches and corpus treatment methods.

**IR Approaches.** TraceLab already implements a set of IR approaches, which will be described briefly as independent variables for our experimental design. In the context of ENTRANCER, these techniques take as input a corpus of documents (i.e., the target artifacts) and a set of queries (i.e., the source artifacts) and compute the textual similarities between the source and target artifacts.

One of earliest techniques is the **Vector Space Model (VSM)** [46], which works as follows. First, it represents the set of documents (i.e., the corpus) as a term-by-document (TD) matrix, where each element of the matrix represents the number of occurrences of the term in the document. Second, the TD matrix is normalized and weighted using a traditional weighting scheme, such as the term frequency-inverse document frequency (tf-idf), which gives more weight to terms that are relevant to the document, and less weight to common terms that frequently appear in the documents. Third, the similarities between the source artifacts, which are also represented as a vector of terms, and the target artifacts are computed using the cosine similarity between these two vectors. Although VSM produces good results when the vocabulary between the source and target artifacts matches, it does not handle the polysemy and synonymy problem.

To overcome this issue, a more advanced IR technique called **Latent Semantic Indexing (LSI)** [19] was introduced. LSI also represents the corpus as a TD matrix, but it uses Singular Value Decomposition (SVD) to decompose the weighted

<sup>3</sup><http://multiwordnet.fbk.eu>

<sup>4</sup><http://www.coest.org/index.php/for-researches1>

TD matrix into three different matrices using a dimensionality reduction factor  $k$ , specified by the user. The reduced space of the decomposed matrices is an approximation of the original TD matrix and captures the most important concepts present in the structure of the original matrix, and at the same time ignoring any minor differences in terminology, thus addressing the polysemy and synonymy problem. Similarly to VSM, LSI also uses the cosine similarity to determine the similarities between source and target artifacts.

**Jensen-Shannon (JS)** [1] is a recent IR technique that represents each artifact of the corpus as a probability distribution of the terms occurring in the artifact. The probability distribution is based on the weight assigned to each term for that particular artifact. The similarities between two software artifacts (i.e., two probability distributions), are measured using an entropy-based metric, called the Jensen-Shannon Divergence. Similarly to VSM, JS does not take into account the relation between terms, thus it encounters the same problems, namely polysemy and synonymy.

**Latent Dirichlet Allocation (LDA)** [8] is a generative probabilistic technique, which models each artifact as a mixture of topics. In other words, each artifact is represented as a probability distribution over a set of topics, and each topic is represented as a probability distribution over the set of terms in the corpus. In order to generate a model, the user must specify the number of topics, and other parameters that affect the “smoothness” of the distribution of the topics in the documents, as well as the “smoothness” of the distribution of the words in the topic. The textual similarity between two software artifacts represented as topic distributions is computed using the Hellinger distance [7].

**The Relational Topic Model (RTM)** [9] is a hierarchical probabilistic model that generalizes LDA, by also considering the links between the modeled artifacts. RTM takes as input the corpus of documents (same as LDA) and a set of predefined links between software artifacts, if they exist. Regardless of the predefined links, the output produced by RTM includes (i) the topic distribution for each artifact (same as LDA), as well as (ii) a set of links between the artifacts based on the similarities of their topic distribution (i.e., artifacts with similar topics will be connected via a link).

Given a set of observations produced by different IR methods, Principal Component Analysis (PCA) can be used to determine various orthogonal dimensions (called principal components) present in the data. These principal components also quantify the variability found in the data. For example, when PCA was applied on IR techniques in the context of traceability link recovery (TLR), Gethers et al. [27] identified that VSM and JS produced equivalent results, and RTM produced orthogonal (complementary) results. Therefore, in order to capture more variability in the data, these orthogonal techniques were combined, resulting in the hybrid techniques VSM-RTM and JS-RTM [27]. These hybrid techniques were generated using an affine transformation [34] between the similarities produced by these orthogonal techniques, where the weight of the IR technique is (1) given equal weight (denoted as  $IR_1 + IR_2$ ), and (2) assigned a weight proportional to the variance obtained during the PCA analysis, denoted as  $IR_1 + IR_2(PCA)$ . The study showed that combining orthogonal IR methods produced improved results over standalone

IR methods for recovering TLs [27].

**Corpus Treatment Methods.** One of our goals is to investigate how different corpus treatment methods affect the precision of IR approaches for computing TLs. We translated the JDK API documentation into Italian using Google Translate, since the applications and requirements documents were written by Italian programmers. We select five different corpus treatment methods.

- The method **Strawman (S)** is a baseline method for our experiments where the complete source code corpus is treated as bag of words.
- Next, for the method **JDK API call expansion (J)**, we replaced JDK API calls with their corresponding description in the JDK documentation. This is followed by identifier splitting. Comments from the source code are discarded, so that we can evaluate how expanding JDK API calls with words from the relevant API call documents affects the precision of recovery of TLs thus reducing the influence of other confounding variables.
- The method **J+W** is a one-step extension of the method **J**, where **Wordnet** synsets of dictionary words found in the corpus are injected.
- **J+S** is the extension of the methods **J** and **S**, where the JDK API calls are expanded along with source comments present in the corpus.
- Finally, the method **J+S+W** is ENTRANCER where all expansion techniques are combined in a single approach.

Our goal is to address RQs by running experiments with all combinations of IR and corpus treatment methods.

*2) Dependent Variables:* Dependent variables for ENTRANCER are precision ( $P$ ) and recall ( $R$ ). To evaluate the accuracy of each IR method, the number of correct links and false positives were collected for each recovery activity performed by a tool. The tool takes as an input the ranked list of candidate links and classifies each link as correct link or false positive until all correct links are recovered. Such a classification is automatically performed by the tool exploiting the original traceability matrix as an oracle.

The values of  $P$  and  $R$  are computed as follows:  $R = \frac{|cor \cap ret|}{|ret|}$  and  $P = \frac{|cor \cap ret|}{|cor|}$ , where  $cor$  and  $ret$  represent the sets of correct links and links retrieved by the tool, respectively. Other than recall and precision, we also use average precision [19], which returns a single value for each ranked lists of candidate links provided.

In this paper we report the values of average precision for  $R = 100\%$ , i.e., when we apply IR methods and we change the similarity threshold value to ensure that all TLs from the oracle are in the set of recovered TLs. However, making this threshold too low leads to the decreased precision, since many incorrect TLs are added to the set of accepted TLs. Determining the range of acceptable similarity threshold values is beyond the scope of this paper and is a subject of future research.

### E. Threats to Validity

In this section, we discuss threats to the validity of this experimental design and how we address and minimize these threats.

1) *Internal Validity: Expansion techniques.* Since evaluating hypotheses is based on the data collected from external sources such as the JDK documentation and Wordnet, we identify three threats to internal validity: richness of the vocabularies and their relation to the domains to which the subject applications belong, and the uniformity of word distribution among these sources.

Even though the JDK documentation offers a richer vocabulary, a threat to validity is that this vocabulary is generic, since it does not relate to specific domains for which subject applications are built. Therefore, it is likely that many words that were added to the corpus from the JDK documentation and Wordnet did not result in computing correct TLs. Moreover, there is a possibility that by adding some generic words to the corpus, it is possible to compute TLs that are incorrect. In addition, vocabularies can be much richer if domain-specific dictionaries or SDKs are used to expand the corpora. Also, an inequality in distributions of words among different topics may result in computing more correct TLs for some modules and fewer correct TLs for some other modules for which there are fewer API calls. We address this threat to validity by recording TLs for different components and using statistical information about distributions of TLs to make conclusions with respect to the RQs.

**Traceability methods.** Choosing ineffective IR-based traceability methods pose a big threat to validity. If methods are too general or trivial (e.g., exact matches among some words), then every possible TL that has some similar words in its source code and requirements will be retrieved, thus inundating stakeholders with TLs that are hard to evaluate. On the other hand, if an IR-approach is specific to a subject application (e.g., ontology-based techniques), high precision will be obtained, thus creating a bias towards this specific application and IR-approach. To avoid this threat, we implemented our experimental design in TraceLab (described in Section II-C), which offers a diversified set of widely used IR-method that are application independent. While this diversification of tasks does not completely eliminate this threat to validity, it reduces it significantly.

2) *External Validity:* To make results of this experiment generalizable, we must address threats to external validity, which refer to the generalizability of a casual relationship beyond the circumstances of our experiment. The fact that supports the validity of this experimental design is that the subject applications and traceability methods are representative of methods that are used in industry and research. A threat to external validity concerns the usage of software traceability tools in the industrial settings, where applications may not use third-party API call libraries. However, it is highly unlikely that modern large-scale software projects can be effectively developed, maintained, and evolved without this reuse.

## IV. RESULTS

In this section, we describe the results of our experiments and using these results we give answers to our RQs.

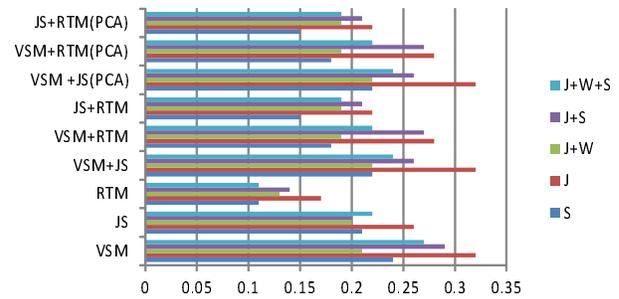


Fig. 1. Precisions for recovering TLs using different IR methods and corpus treatment methods for subject application Albergate.

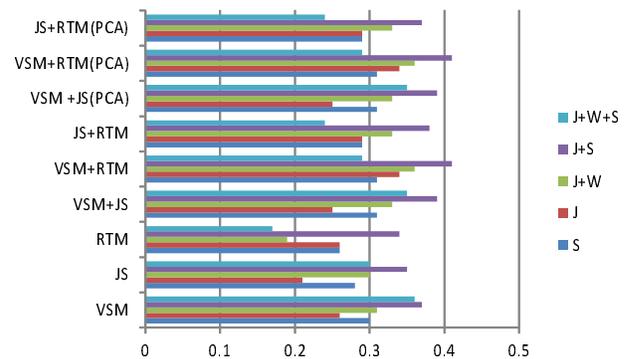


Fig. 2. Precisions for recovering TLs using different IR methods and corpus treatment methods for subject application eTour.

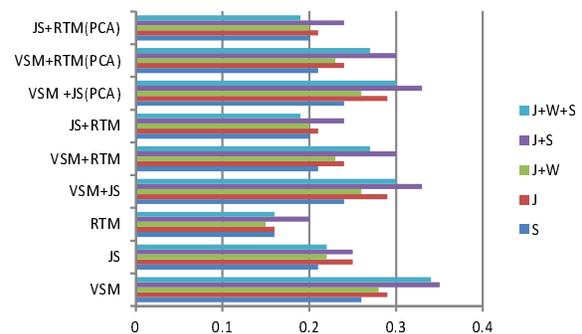


Fig. 3. Precisions for recovering TLs using different IR methods and corpus treatment methods for subject application SMOS.

The results of experiments with subject applications for different values of the dependent variable Precision,  $P$  are shown in Table I and they are visualized in Figure 1, Figure 2, and Figure 3 for the corresponding subject applications. ENTRANCER leads to the increase in  $P$  in the majority of cases (75.92%), thus positively answering  $RQ_1$ - $RQ_3$ . Closer examination of the data reveals that for the method JS for the subject application Albergate the precision increases for the

TABLE I. RESULTS OF EXPERIMENTS WITH SUBJECT APPLICATIONS FOR DIFFERENT IR TECHNIQUES. COLUMNS 2-6 DESCRIBE THE SIZE OF THE DATASETS IN TERMS OF NUMBER OF REQUIREMENTS DOCUMENTS, JAVA SOURCE CODE FILES, NUMBER OF POSSIBLE LINKS, NUMBER OF ACTUAL LINKS IN THE GOLD SET, AND THE RATIO OF ACTUAL TO POSSIBLE LINKS. COLUMNS 7-9 SHOW THE SOURCE CODE AND REQUIREMENTS DOCUMENT SIZES IN KB AND THE RATIO OF THESE SIZES. THE NEXT COLUMN, “MAX EXP RATIO,” SHOWS THE RATIO OF THE NUMBER OF THE UNIQUE WORDS IN THE EXPANDED CORPUS TO THE NUMBER OF UNIQUE WORDS IN THE ORIGINAL CORPUS. FINALLY, THE COLUMN LABELED “METHOD TRACEABILITY” AND THE FOLLOWING FIVE COLUMNS LIST THE MEAN VALUES OF THE PRECISION FOR CORPUS TREATMENT METHODS FROM SECTION III-D1 FOR THE VALUES OF RECALL,  $R = 100\%$ .

Subject App	Doc Files	Code Files	Posibl Links	Actual Links	PL/AL Ratio	Code Size,KB	Req Corp Size,KB	D/C Ratio	Max Exp Ratio	Method Traceability	$P$ for corpus treatment methods				
											S	J	J+W	J+S	J+W+S
Albergate	16	55	880	54	0.06	516	72	0.13	362.08%	VSM	0.24	0.32	0.21	0.29	0.27
										JS	0.21	0.26	0.2	0.2	0.22
										RTM	0.11	0.17	0.13	0.14	0.11
										VSM+JS	0.22	0.32	0.22	0.26	0.24
										VSM+RTM	0.18	0.28	0.19	0.27	0.22
										JS+RTM	0.15	0.22	0.19	0.21	0.19
										VSM +JS(PCA)	0.22	0.32	0.22	0.26	0.24
										VSM+RTM(PCA)	0.18	0.28	0.19	0.27	0.22
										JS+RTM(PCA)	0.15	0.22	0.19	0.21	0.19
										eTourITA	67	100	6700	1044	0.16
JS	0.28	0.21	0.3	0.35	0.3										
RTM	0.26	0.26	0.19	0.34	0.17										
VSM+JS	0.31	0.25	0.33	0.39	0.35										
VSM+RTM	0.31	0.34	0.36	0.41	0.29										
JS+RTM	0.29	0.29	0.33	0.38	0.24										
VSM +JS(PCA)	0.31	0.25	0.33	0.39	0.35										
VSM+RTM(PCA)	0.31	0.34	0.36	0.41	0.29										
JS+RTM(PCA)	0.29	0.29	0.33	0.37	0.24										
SMOS	67	100	6700	1044	0.16	715	181	0.25	242.17%						
										JS	0.21	0.25	0.22	0.25	0.22
										RTM	0.16	0.16	0.15	0.2	0.16
										VSM+JS	0.24	0.29	0.26	0.33	0.3
										VSM+RTM	0.21	0.24	0.23	0.3	0.27
										JS+RTM	0.2	0.21	0.2	0.24	0.19
										VSM +JS(PCA)	0.24	0.29	0.26	0.33	0.30
										VSM+RTM(PCA)	0.21	0.24	0.23	0.3	0.27
										JS+RTM(PCA)	0.2	0.21	0.2	0.24	0.19

corpus treatment method J when compared to S, but it drops when applying the corpus treatment methods J+W, J+S, and J+W+S. In fact, when comparing the results for corpus treatment methods J and S, precision increases for all IR methods for the application Albergate, but it drops for the application eTour when applying the IR methods VSM, JS, VSM+JS, and VSM+JS(PCA). A possible explanation is that by removing comments from the source code for this application when applying the corpus treatment method J, the expansion rate of the corpus is not sufficient to increase precision for methods where word matches are important for computing TLs. A conclusion from this observation is that expanding the corpus with the documentation JDK API calls only is often not enough to get higher precision of traceability links when applying word match similarity methods thus addressing  $RQ_1$ .

We make similar observations about the precision of different approaches when combining expansion of the corpus with documentation from the JDK API calls and Wordnet. When comparing the results for corpus treatment methods J+W and J, precision increases for all IR methods for the application eTour, but it drops for the applications Albergate and SMOS for all IR methods. The resulting precision is still higher in most cases when compared to the baseline corpus treatment method S, however, it points out to an interesting phenomenon. Clearly, expanding the corpus with relevant words works only if the words in the original corpus can serve as good semantic anchors. In our case, using Google translator in the presence

of mixed language vocabulary reduces the semantic quality of information. Identifier splitting also has limited precision, leading to situations where split words may not have semantic relevancy. As a result, expanding semantically irrelevant corpus with more irrelevant words leads to reduced precision. A possible reason that the application eTour avoids the reduction in precision is because the rate of its corpus expansion is the smallest when compared to the two other subject applications, thus having fewer erroneous TLs in the end. *We conclude that expansion of the corpus with a combined documentation from the JDK API calls and Wordnet does not always result in a higher precision of traceability links thus addressing  $RQ_2$ .*

Adding comments from the source code to the corpus when expanding the JDK API calls with their documentation leads across the board increase in  $P$  with exception of the IR method RTM, which remains the worst method in our experiment to see the effect of expanding the corpus with relevant documentation. Part of this is addressed in the previous work, noting that in some studies RTM was found to provide orthogonal results, which could be used to complement other IR techniques [27].

*We conclude that including words from comments results in a higher precision of traceability links when expanding the corpus with a combined documentation from the JDK API calls and Wordnet.*

We investigate the question of correlation between the

size of the corpus and the precision of traceability links. This research question can be answered with more rigorous experimentation in the future. To understand how each of our corpus modification methods effects the source code corpus, we measure the change in the total word count and the number unique words for each source code file from the subject applications. Table II shows the average percentage difference in the length of source code files when measured relative to the unmodified original source code files. eTour source code files are extensively commented with 43,211 words which is 19.21 and 2.21 times greater than Albergate and SMOS respectively. As a result eTour’s corpus size actually shrinks and leads to a decrease in the precision values for the corpus treatment method **J** (which excludes comments). On the other hand, when using corpus treatment methods **J+S** and **J+W+S**, which include the comments, we observe an improvement in precision of recovering TLs across all IR methods.

TABLE II. AVERAGE PERCENTAGE DIFFERENCE IN THE LENGTH OF SOURCE CODE FILES WHEN MEASURED RELATIVE TO THE UNMODIFIED ORIGINAL SOURCE CODE FILES.

		Type	J	J+W	J+S	J+W+S
<b>Albergate</b>	total		202.05	476.80	200.02	624.87
	unique		96.71	261.06	136.81	362.08
<b>eTour</b>	total		-18.50	238.44	100.79	357.75
	unique		-60.41	32.20	16.50	108.58
<b>SMOS</b>	total		69.08	592.92	210.43	734.36
	unique		-18.34	161.68	62.36	242.17

Based on our experimental results, we conclude that there is a correlation between the size of the corpus and higher precision of recovered TLs thus addressing  $RQ_4$ .

ENTRANCER is not equally effective for different IR approaches for computing software traceability – VSM remains the biggest winner and RTM is the biggest loser among the used IR approaches. Adding VSM to a combination of other IR methods improves the overall precision. Best ratios of precision improvements for ENTRANCER are shown in Table III, where the maximum obtained precision increase is 31% for SMOS using the IR method VSM and the average precision gain is nine percent. The lowest precision is obtained with the IR method RTM. These best ratios are shown in Figure 4 for precision improvements when compared to the baseline S for different IR methods applied to subject applications with ENTRANCER that is shown in Table III. We conclude that using VSM results in a higher precision of traceability links when expanding the corpus with a combined documentation from the JDK API calls and Wordnet thus addressing  $RQ_5$ .

## V. RELATED WORK

The first part of this section discusses in detail the most relevant papers for ENTRANCER. The second part of the section briefly enumerates a subset of the traditional TLR techniques and tools.

Cleland-Huang et al. [13] proposed a technique to recover TLs between regulatory codes and requirements by enhancing the original query with similar terms that would help address the polysemy and synonymy problem. More specifically, their technique uses the original query as an input to web search engines which retrieve a set of documents related to the query.

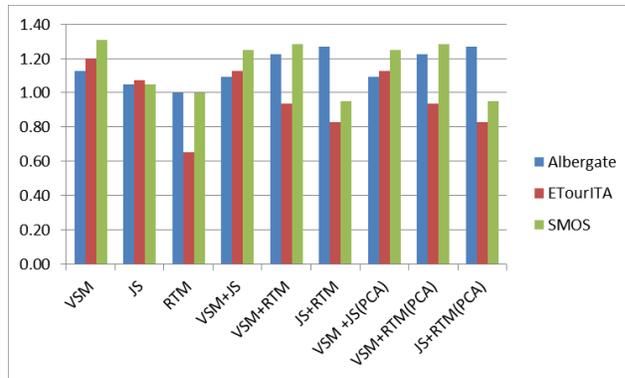


Fig. 4. Best ratios for precision improvements when compared to the baseline S for different IR methods applied to subject applications with ENTRANCER that is shown in Table III.

Among the words of these documents, the nouns and noun-phrases are extracted, and term specific metrics are computed (e.g., domain term frequency, domain specificity, concept generality). The nouns with the highest metric values are appended to the original query, and the enhanced query is used for retrieving TLs. A comparison with a basic IR TLR technique revealed that, in some cases, their proposed technique is more accurate in retrieving TLs [13]. Gibiec et al. [28] improved the traceability technique proposed by Cleland-Huang et al. [13], by automatically identifying the set of domain specific terms (from the retrieved web-mining results), which will be used to enhance the original query. Our approach is similar to Cleland-Huang et al.’s [13] and Gibiec et al.’s [28] approaches in terms of mining additional data to enhance the existing information. However, the main difference is that ENTRANCER uses API documentation to enhance the existing API calls found in the corpus, as opposed to expanding only the query.

Dekel and Herbsleb [21][22] introduced an approach based on the concept of knowledge push, which directs the attention of developers to certain API calls that might need extra consideration. First, the approach extracts some directives (e.g., restrictions, limitations, performance issues, alternatives, etc.) in the form of sentences that are embedded in the API documentation. Second, this knowledge, which could be easily skipped by a developer reading the API documentation, is “pushed” or presented to the developer by highlighting in the Eclipse IDE the method invocations that have associated these directives [21][22].

The identifier mismatch problem was investigated in the literature, and a different number of solutions have been suggested. Deissenboeck and Pizka [20] proposed a tool that enforces unique mappings between identifier names and concepts, in order to reduce the mismatch problem. The tool constantly updates the mappings while the system evolves. Lawrie and Binkley [36] proposed a solution for automatically expanding the splitted identifiers to their unabbreviated form. However, the impact of these tools and techniques [20][36][32] has not yet been evaluated for TLR.

Cleland-Huang et al. [11] advocated for writing requirements in a more concise and clear way, and to use a consistent domain-specific vocabulary. De Lucia et al. [17] proposed COCONUT, an IR-based TLR system that helps developers

TABLE III. BEST RATIOS OF PRECISION IMPROVEMENTS FOR ENTRANCER.

	VSM	JS	RTM	VSM+JS	VSM+RTM	JS+RTM	VSM +JS(PCA)	VSM+RTM(PCA)	JS+RTM(PCA)
Albergate	1.13	1.05	1.00	1.09	1.22	1.27	1.09	1.22	1.27
ETourITA	1.20	1.07	0.65	1.13	0.94	0.83	1.13	0.94	0.83
SMOS	1.31	1.05	1.00	1.25	1.29	0.95	1.25	1.29	0.95

to select the most meaningful identifier names, which are consistent with the domain terms found in the high-level artifacts (e.g., requirements).

Antoniol et al. [3] proposed an IR-based technique to recover TLs between documentation and source code using the vector space model (VSM). Their work was extended by Marcus et al. [37] who used latent semantic indexing (LSI) to recover TLs, and showed that this technique produces better results. Oliveto et al. [42] compared the performances of three IR-based techniques (i.e., VSM, LSI and Jensen-Shannon (JS)) and one topic-model technique, LDA (Latent Dirichlet Allocation). Their results show the IR-techniques produce equivalent results, whereas LDA produced complementary results, which are orthogonal to the ones produced by the IR techniques. The orthogonality of results was leveraged by Gethers et al. [27] who combined the information produced by the IR techniques VSM and JS with the information produced by the topic model technique RTM (Relational Topic Model) to produce superior results than using standalone techniques.

Among the numerous tools for supporting TLR that have been introduced we mention a few representative ones. TOOR [44] is a traceability tool that supports manual tracing between various software artifacts, as well as management of existing TLs. ADAMS [18] is a tool that automatically generates TLs between different software artifacts, using LSI.

## VI. CONCLUSION

We created a novel and automatic approach for expanding corpora with relevant documentation that is obtained using external function call documentation and sets of relevant words. We experimented with three Java applications using the TraceLab framework and we evaluate four methods of systematically expanding source code corpus using relevant words and semantic anchors. These methods improve precision the of TL recovery in more than 75% of the cases, with 31% in the best case and approximately 9% on average.

ENTRANCER is most effective using the IR method *Vector Space Model (VSM)*, where artifacts are represented as vectors of words and a similarity measure is computed as the cosine between these vectors. Since the main idea of ENTRANCER is to expand the corpora with words that are relevant to words in the documents, applying ENTRANCER increases the probability of establishing matches between words in requirements and other software artifacts, thereby leading to sizeable increase in the precision of recovering TLs. However, other IR methods like RTM benefit less from ENTRANCER for a variety of different reasons, one of them being that the expanded corpora contains words that reduce the effectiveness of computing probabilistic distributions of these words across different topics. Thus, we conclude that ENTRANCER produces the best results when applied with VSM.

## ACKNOWLEDGMENTS

This work is supported by NSF CCF-1217928, CCF-1017633, NSF CCF-0916139, NSF CCF-1218129, NSF CNS-0959924, and NSF CCF-1016868. We warmly thank our ICSM reviewers whose comments helped us improve the quality of this paper.

## REFERENCES

- [1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *Proceedings of the 16th IEEE ICPC*, ICPC '08, pages 103–112, Washington, DC, USA, 2008. IEEE Computer Society.
- [2] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.
- [3] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE TSE*, 28(10):970–983, 2002.
- [4] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. Information retrieval models for recovering traceability links between code and documentation. In *IEEE ICSM'00*, ICSM '00, pages 40–, Washington, DC, USA, 2000. IEEE Computer Society.
- [5] G. Bavota, A. D. Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora. The role of artefact corpus in lsi-based traceability recovery. In *7th TEFSE'13*, page to appear, San Francisco, California, May 19, 2013.
- [6] T. J. Biggerstaff, B. G. Mitbender, and D. E. Webster. The concept assignment problem in program understanding. In *ICSE*, pages 482–498, 1993.
- [7] D. Blei and J. Lafferty. *Text mining: Theory and Applications, chapter Topic Models*, chapter 5, pages 365–452. Taylor and Francis, London, 2009.
- [8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- [9] J. Chang and D. M. Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, 4(1):124–150, 2010.
- [10] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine using free-form queries. In *FASE*, pages 385–400, 2009.
- [11] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settini, and E. Romanova. Best practices for automated traceability. *IEEE Computer*, 40(6):27–35, 2007.
- [12] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, G. O., J. Huffman Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder. Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community. In *6th TEFSE2011*, Honolulu, HI, USA, May 23, 2011.
- [13] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Eme-necker. A machine learning approach for tracing regu-

- latory codes to product specific requirements. In *32nd ACM/IEEE ICSE'10*, pages 155–164, 2010.
- [14] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, M. Gethers, D. Poshyvanyk, J. H. Hayes, and W. Li. Toward actionable, broadly accessible contests in software engineering. In *34th IEEE/ACM ICSE'12 NIER Track*, pages 1329–1332, Zurich, Switzerland, June 2-9, 2012.
- [15] I. Consulting. Iag consulting business analysis benchmark. In *IAG*, 2009.
- [16] A. Czauderna, M. Gibiec, G. Leach, Y. Li, Y. Shin, E. Keenan, and J. Cleland-Huang. Traceability challenge 2011: Using tracelab to evaluate the impact of local versus global idf on trace retrieval. In *6th TEFSE'11*, volume 6, Honolulu, HI, USA, 2011.
- [17] A. De Lucia, M. Di Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE TSE*, page (to appear), 2010.
- [18] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM TOSEM*, 16(4), 2007.
- [19] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.
- [20] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.
- [21] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, pages 320–330, 2009.
- [22] U. Dekel and J. D. Herbsleb. Reading the documentation of invoked api functions in program comprehension. In *17th IEEE ICPC'09*, pages 168–177. IEEE, 2009.
- [23] B. Dit, E. Moritz, and D. Poshyvanyk. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In *20th IEEE ICPC'12*, pages 203–208, Passau, Germany, June 11-13, 2012.
- [24] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. Configuring topic models for software engineering tasks in tracelab. In *7th TEFSE'13*, page to appear, San Francisco, California, May 19, 2013.
- [25] R. Dömges and K. Pohl. Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62, Dec. 1998.
- [26] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.
- [27] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia. On integrating orthogonal information retrieval methods to improve traceability link recovery. In *27th IEEE ICSM'11*, pages 133–142, 2011.
- [28] M. Gibiec, A. Czauderna, and J. Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *25th IEEE/ACM ASE'10*, pages 245–254, Antwerp, Belgium, 2010. ACM.
- [29] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *RE*, pages 94–101, 1994.
- [30] M. Grechanik, K. M. Conroy, and K. Probst. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.
- [31] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. M. Cumby. A search engine for finding highly relevant applications. In *ICSE (1)*, pages 475–484, 2010.
- [32] M. Grechanik, K. S. McKinley, and D. E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *ESEC/SIGSOFT FSE*, pages 95–104, 2007.
- [33] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. SpringerVerlag, 2004.
- [34] R. Jacobs. Methods for combining experts' probability assessments. *Neural Computation*, 7(5):867–888, 1995.
- [35] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hussein, and D. Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *34th IEEE/ACM ICSE'12*, pages 1375–1378, Zurich, Switzerland, June 2-9, 2012.
- [36] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *27th IEEE ICSM'11*, pages 113–122, Williamsburg, Virginia, USA, September 25-30, 2011. IEEE Computer Society.
- [37] A. Marcus, J. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.
- [38] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137, 2003.
- [39] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *ICSE*, pages 364–374, 2012.
- [40] S. Mizzaro. Relevance: The whole history. *JASIS*, 48(9):810–832, 1997.
- [41] S. Mizzaro. How many relevances in information retrieval? *Interacting with Computers*, 10(3):303–320, 1998.
- [42] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *18th IEEE ICPC'10*, pages 68–71, 2010.
- [43] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. When and how using structural information to improve ir-based traceability recovery. In *17th CSMR'13*, pages 199–208, Genova, Italy, March 5-8, 2013.
- [44] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.
- [45] R. Rapp. The computation of word associations: comparing syntagmatic and paradigmatic approaches. In *19th ICCL*, pages 1–7, Morristown, NJ, USA, 2002.
- [46] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.
- [47] J. Stylos and B. A. Myers. A web-search tool for finding API components and examples. In *IEEE Symposium on VL and HCC*, pages 195–202, 2006.
- [48] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

# Supporting and Accelerating Reproducible Research in Software Maintenance using TraceLab Component Library

Bogdan Dit, Evan Moritz, Mario Linares-Vásquez, and Denys Poshyvanyk

Computer Science Department  
The College of William and Mary  
Williamsburg, VA, USA  
{bdit, eamoritz, mlinarev, denys}@cs.wm.edu

**Abstract**—Research studies in software maintenance are notoriously hard to reproduce due to lack of datasets, tools, implementation details (*e.g.*, parameter values, environmental settings) and other factors. The progress in the field is hindered by the challenge of comparing new techniques against existing ones, as researchers have to devote a lot of their resources to the tedious and error-prone process of reproducing previously introduced approaches. In this paper, we address the problem of experiment reproducibility in software maintenance and provide a long term solution towards ensuring that future experiments will be reproducible and extensible. We conducted a mapping study of a number of representative maintenance techniques and approaches and implemented them as a library of experiments and components that we make publicly available with TraceLab, called the *Component Library*. The goal of these experiments and components is to create a body of actionable knowledge that would (i) facilitate future research and would (ii) allow the research community to contribute to it as well. In addition, to illustrate the process of using and adapting these techniques, we present an example of creating new techniques based on existing ones, which produce improved results.

**Keywords**—software maintenance, reproducible, experiments, case studies, TraceLab

## I. INTRODUCTION

Research in software maintenance (SM) is primarily driven by empirical studies. Thus, advancing this field requires researchers not only to come up with new, more efficient and effective approaches that address SM problems, but most importantly, to compare their new approaches against existing ones in order to demonstrate that they are complementary or superior and under which scenarios.

However, comparing an approach against existing ones is time consuming and error-prone. For example, the existing approaches may be hard to reproduce because the datasets used in their evaluation, the tools and implementation, or the implementation details (*e.g.*, specific parameter values, environmental factors) are not available [1, 2, 3, 4, 5, 6].

For example, a survey on feature location (FL) techniques by Dit *et al.* [1] revealed that only 5% of the papers surveyed (three out of 60 papers) used in their evaluation the same dataset that was used in evaluating other techniques, and that only 38% of the papers surveyed (23 out of 60 papers) compared their proposed feature location technique against a small number of previously introduced feature location techniques. In addition, these findings are consistent with the

ones from the study by Robles [2], which determined that among the 154 research papers analyzed, only two made their datasets and implementation available, and the vast majority of the papers describe evaluations that cannot be reproduced, due to lack of data, details, and tools. Furthermore, a study by González-Barahona and Robles [6] identified the factors affecting the reproducibility of results in empirical software engineering research and proposed a methodology for determining the reproducibility of a study. In another study, Mytkowicz *et al.* [3] investigated the influence of the omitted-variable bias (*i.e.*, a bias in the results of an experiment caused by omitting important causal factors from the design) in compiler optimization evaluation. Their study showed that factors such as the environment size and the link order, which are often not reported and are not explained properly in the research papers, are very common, unpredictable and can influence the results significantly. Moreover, D'Ambros *et al.* [4] argued that many approaches in bug prediction have not been evaluated properly (*i.e.*, they were either evaluated by themselves, or they were compared against a limited set of other approaches), and highlight the difficultness of comparing results.

This issue of the reproducibility of experiments and approaches has been discussed and investigated in different areas of software maintenance research [1, 2, 3, 4, 5, 6], and some initial steps have been taken towards solving this problem. For example, efforts for establishing datasets or benchmarks that can be used uniformly in evaluations have resulted in online benchmark repositories such as PROMISE [7, 8], Eclipse Bug Data [9], SEMERU feature location dataset [1], Bug Prediction Dataset [4], SIR [10], and others. In addition, different infrastructures for running experiments were introduced, such as TraceLab [11, 12, 13], RapidMiner [14], Simulink [15], Kepler [16], and others. However, among these, the most suitable framework for facilitating and advancing research in software engineering and maintenance is TraceLab (see Section III.B for an in-depth comparison and discussion of TraceLab's features with other tools). TraceLab is a plug-and-play framework that was specifically designed for facilitating creating, evaluating, comparing, and sharing experiments in software engineering and maintenance. These characteristics ensure that TraceLab makes experiments *reproducible*.

The goal of this paper is to ensure that a large portion of existing and future experiments in software maintenance research that are designed and implemented with TraceLab will be *reproducible*. We analyzed the approaches presented in 27

research papers and we implemented them as TraceLab experiments. In order to implement these SM approaches, we identified their common building blocks and we implemented them as components in a well organized (structured), documented and comprehensive *Component Library* for TraceLab. In addition, we used the *Component Library* to assemble and replicate a subset of existing SM techniques, and to exemplify how these components and experiments can be used as starting points for creating new and reproducible experiments.

In summary, the contributions of our paper are as follows:

- a mapping study of techniques and approaches in SM (Section IV) to identify the set of techniques that we reproduced as TraceLab experiments;
- a TraceLab *Component Library (CL)*, which contains a comprehensive and representative set of TraceLab components designed to help instantiate the set of SM experiments, and a *Component Development Kit (CDK)*, which serves as a base for extending this component base in order to facilitate the creation of new techniques and experiments;
- an example of reproducing a feature location technique using the proposed *CL*, as well as using the existing technique as a starting point to design and evaluate new ideas;
- an online appendix that makes publicly available all the resources presented in this paper: [www.cs.wm.edu/semeru/TraceLab\\_CDK](http://www.cs.wm.edu/semeru/TraceLab_CDK)

The paper is organized as follows. Section II presents a motivating example that shows variability in results of applying a simple SM technique and challenges of reproducing those results without complete details. Section III introduces background details about TraceLab and presents a comparison with other tools. Section IV presents the mapping study performed, which we used to implement the Component Library and Development Kit (Section V). Section VI shows an example of reproducing an existing FL technique and presents details on improving it. Finally, Section VII discusses some potential limitations and Section VIII concludes the paper and introduces some ideas for future work.

## II. MOTIVATING EXAMPLE

When new approaches are introduced, in general, authors rightfully focus more on describing the important details of the new techniques, and due to various reasons (*e.g.*, space limitations) they may present only in passing the details of applying well-known and popular techniques (*e.g.*, VSM), as they rely on the conventional wisdom and knowledge (or references to other papers for more details) about applying these techniques [1, 2].

However, for a researcher who tries to reproduce the results exactly, it might be difficult to infer all the assumptions the original authors took for granted and did not explicitly state in the paper. Therefore, the reproducer's interpretation of applying the approach could have significant impact on the results.

To illustrate this point on a concrete example, we applied the popular IR technique Vector Space Model (VSM) [17] on the EasyClinic system from TEFSE 2009<sup>1</sup> challenge to recover

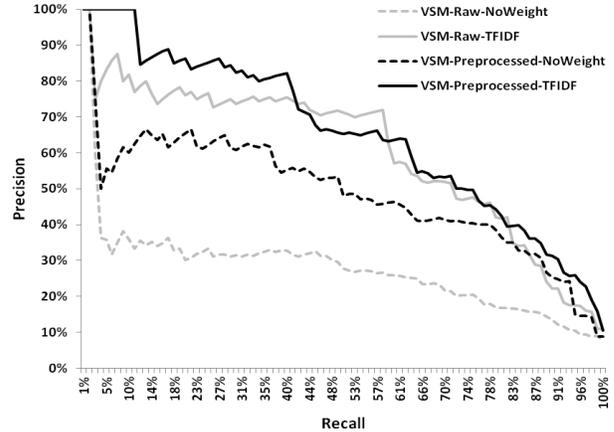


Figure 1 Precision-Recall curves for EasyClinic for recovering traceability links between use cases and classes using a VSM-based traceability technique and different preprocessing techniques (*raw* – gray color, *preprocessed* – black color) and weighting schemes (*no weight* – dash line, *tf-idf* – solid line)

traceability links between use cases and class diagrams. We configured the VSM technique using four treatments consisting of all the possible combinations of two corpus preprocessing techniques and two VSM weighting schemes. The preprocessing techniques were *raw preprocessing* (*i.e.*, only the special characters were removed) and *basic preprocessing* (*i.e.*, remove special characters, split identifiers and stem). The weighting schemes used were *no weighting* and *term frequency-inverse document frequency (tf-idf)* weighting. Figure 1 indicates the raw and basic preprocessing steps with gray and black color respectively, and the no weighting and tf-idf weighting with dashed line and solid line respectively. The results in Figure 1 show a high variety in the precision and recall values, based on the type of preprocessing and weighting schemes used. Assuming these details are not clearly specified in the paper, any of these configurations or variations of these configurations can be chosen while reproducing an experiment, potentially yielding completely unexpected and drastically different results. It is worth emphasizing that in our example we picked a small subset of the large number of weighting schemes and preprocessing techniques that can be found in the literature, and these options were deliberately picked to illustrate an example, as opposed to conducting a rigorous experiment to identify the configuration of factors that could produce the best results.

The main point of this example is that even in this simple scenario of using VSM for a typical traceability task, there are many options on how we can instantiate and use this technique, which leads to completely different results. However, all these problems could be eliminated if all these details are encoded in the experiment description, such as one designed in TraceLab.

## III. BACKGROUND AND RELATED WORK

This section provides the background details about TraceLab as an environment for SM research and compares and contrasts TraceLab to other research tools specific to other domains.

<sup>1</sup> <http://web.soccerlab.polymtl.ca/tefse09/Challenge.htm>

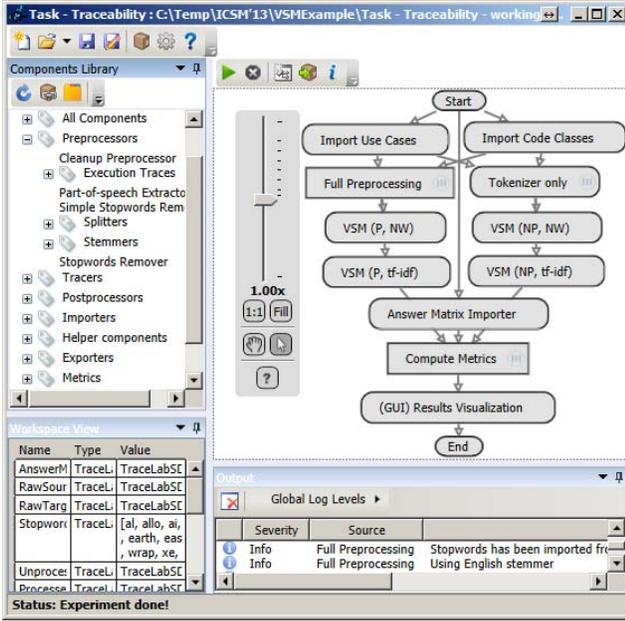


Figure 2 The four "quadrants" of TraceLab in clockwise order from top-right are (i) the sample TraceLab experiment that implements our motivating example in Section 2; (ii) an output window for reporting execution status of an experiment; (iii) the Workspace containing the data and the values of the experiment; and (iv) the *Component Library*

### A. TraceLab

TraceLab [11, 12, 13] is a framework designed to support the reproducibility of experiments in software engineering and software maintenance (see Figure 2). More specifically, it allows researchers to create, evaluate, compare, and most importantly share experiments in SM research. TraceLab was developed at DePaul University in collaboration with researchers at Kent State University, University of Kentucky, and the College of William and Mary.

The heart of a TraceLab experiment lies in its workflow of components and tools (see Figure 2 upper-right). An experiment is a collection of nodes (or components) connected in the form of a precedence graph. Each component communicates with the preceding and following nodes by storing and loading information to and from a common data-sharing interface called the *Workspace* (see Figure 2 lower-left). The status of an experiment is reported in the Output view (see Figure 2 lower-right). Individual components are engineered to implement a specific task and components that implement related tasks can be combined to form composite components, such as the node with rectangular edges labeled *Queries preprocessing* in Figure 4, which implements various tasks such as identifier splitting, stemming, and stopwords removal. In addition, TraceLab provides basic control flow within the experiment via decision nodes and *while loops* (see Figure 4).

A major contribution of this paper is a *Component Library*, designed to implement a wide range of SM techniques that can be easily accessed from TraceLab (see Figure 2 upper-left). The *Component Library* will be included in the distribution of next official TraceLab release.

Table I Comparison of TraceLab with other related tools (columns). The features (rows) are as follows: 1) data-flow oriented GUI [Yes / No]; 2) Type of application [Desktop / Web / API]; 3) License type [Commercial / Open source / Free online access]; 4) Tool allows saving and loading experiments [Yes / No]; 5) Tool allows creating composite components [Yes / No / Programmatically]; 6) Tool has a component "market" where developers can contribute with their own components [Yes / No]; 7) Programming language that can be used to build new components; 8) The platforms were the tool could be used [Software As A Service, Windows, Linux, Mac]

Tool	Yahoo pipes	Weka/R. Miner	Simulink	Gate	Kepler	TraceLab
<b>GUI</b>	Y	Y	Y	N	Y	Y
<b>Type</b>	W	API, D	D	API, D	API, D	API, D
<b>License</b>	F	O	C	O	O	O
<b>Save/Load exp.</b>	Y	Y	Y	Y	Y	Y
<b>Composite comp.</b>	Y	N	Y	P	Y	Y
<b>Comp. Market</b>	Y	N	Y	Y	Y	Y
<b>Prog. Lang.</b>	-	Java	C/C++ Matlab. Fortran	Java	R C Matlab Java	Java R .NET lang. Matlab
<b>Platforms</b>	SAAS	W, L, M	W, L, M	W, L, M	W, L, M	W, L, M

### B. TraceLab Comparison with Other Tools

There are also numerous other frameworks and tools that were designed to support research in other domains, such as information retrieval, machine learning, data mining, and natural language processing, among others. Consequently, reuse of third party tools or APIs is a common practice for making experiments and building research infrastructure in software evolution and maintenance. For example, a common scenario is to reuse WEKA for implementations of machine learning classifiers, R for statistical analysis, or MALLET for topic modeling. However, these tools/APIs were not built to support research on software evolution and maintenance. Moreover, most of the tools were conceived as extensible APIs and only few of them provide features such as experiment composition by using a data-flow GUI, new components implementation, or easy sharing/publishing of experiments; moreover, not all of them can be used across multiple platforms. Table I compares TraceLab to some similar tools that also use a data-flow oriented GUI.

WEKA [18] is a collection of machine learning algorithms that are packaged as an open source Java library that also allows running the algorithms using a graphical user interface (GUI). One of the WEKA modules is the *KnowledgeFlow*, which provides the user with a data-flow oriented GUI for designing experiments. As in TraceLab, the components in the KnowledgeFlow are categorized by tasks (DataSources, DataSinks, Filters, Classifiers, Clusterers, Associations, Evaluation, Visualization), and there is a layout canvas for designing experiments by dragging, dropping, and connecting components. New components can be added to WEKA by extending or modifying the library using Java, and the experiments can be saved and loaded for being executed in the WEKA Experimenter module.

RapidMiner [14] is a data mining application that provides an improved GUI for designing and running experiments. It includes a reusable library for designing experiments and running them and it fully integrates WEKA as the machine learning library.

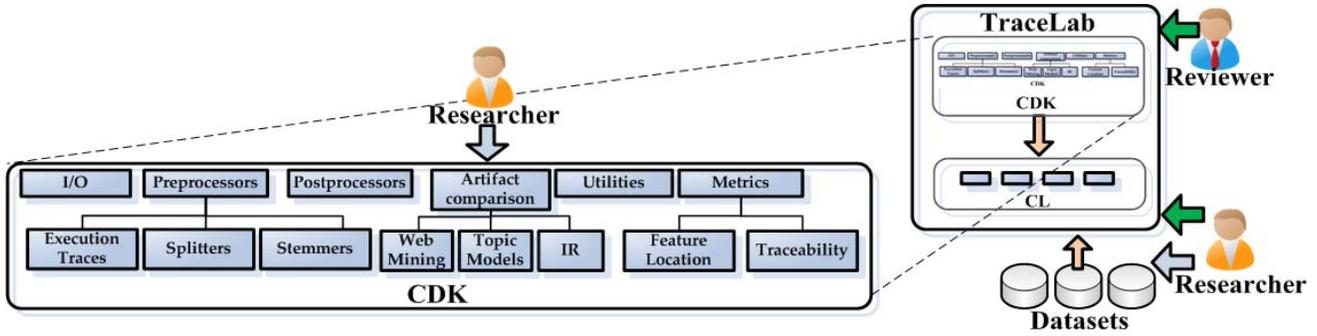


Figure 3 Diagram of the hierarchy of the *CDK* in the context of TraceLab. *CDK* and *CL* are part of TraceLab. Researchers can contribute to the *CDK* and the datasets (gray arrow), and reviewers and researchers (green arrow) can use TraceLab to verify details of existing experiments

Yahoo Pipes [19] is a data mashup tool with components for web retrieval, filtering and aggregation of web feeds, web pages, and other services. As the TraceLab composite components, pipes (*i.e.*, Yahoo pipes composite components) can be reused as building blocks for new pipes. In addition, pipes are shared/published through the Yahoo pipes website.

Simulink [15] is a Matlab-based tool for simulation and model-based design of embedded systems. In Simulink, a model is composed of subsystems (*i.e.*, a group of blocks) or individual blocks, and the blocks can be implemented using Matlab, C/C++, or Fortran.

GATE [20] provides an environment for text processing that includes an IDE with components for language processing, a web application for collaborative annotation of document collections, a Java library, and a cloud solution for large scale text processing.

Kepler [16] is a tool that follows the same philosophy as TraceLab. By using Kepler, it is possible to build, save, and publish experiments/components using a data-flow oriented GUI. It is also possible to extend Kepler because of its collaborative-project nature. However, the main difference with TraceLab is that Kepler was conceived as a tool for experiments in sciences such as Math or Physics.

Although TraceLab is not specialized on simulation, natural language processing, or machine learning, it was specifically designed to allow software engineering and maintenance researchers the possibility to (i) *develop* and *share* their own components/experiments, and (ii) to ensure the *reproducibility* of their results. TraceLab supports all major OS platforms (*e.g.*, Window, MacOS and Linux) and researchers can use Java, any .NET language (*e.g.*, C#, VB, C++), R or Matlab to implement their components.

#### IV. MAPPING STUDY OF SOFTWARE MAINTENANCE TECHNIQUES

In this section we present the methodology, analysis and results of a mapping study [21] aimed at identifying a set of techniques from particular areas of SM, which could be implemented as TraceLab experiments in order to constitute an initial practical body of knowledge that would benefit the SM research community. Moreover, these identified techniques were reverse engineered into basic modules that we implemented as TraceLab components, in order to generate a *Component Development Kit* (see Section V.A) and a *Component Library* (see Section V.B) that serves as a starting

point for any interested researcher to implement new techniques or build upon existing ones.

For our study, we use the systematic mapping process described by Petersen *et al.* [22]. The process consists of five stages: 1) defining the research questions of the study, 2) searching for papers in different venues, 3) screening the papers based on inclusion and exclusion criteria in order to find the relevant ones, 4) classifying the papers, and 5) data extraction and generating the systematic map.

##### 1) Defining the Research Questions

Our goal is to identify a set of representative techniques from specific areas of SM, and use them to generate TraceLab components and experiments to accelerate and support research in SM. Therefore, our main guiding research question was formulated as: *Which SM techniques are suitable to form an initial actionable body of knowledge that other researchers could benefit from?* In particular, we focused on a subset of SM areas where the authors have expertise, which allowed generating this initial body of knowledge that could support the research community, and one that we, and the research community, could contribute to, by constantly adding new techniques and components.

##### 2) Conducting the Search

In order to find these techniques, we narrowed the search space to the publications from the last ten years of a subset of journals and software engineering conferences (see our online appendix for more details). In addition, in our search we incorporated the "snowballing" discovery technique (*i.e.*, following references in the related work) discussed by Kitchenham *et al.* [21].

##### 3) Screening Criteria

The primary *inclusion* criterion consisted in identifying whether the research paper described a technique that addressed one of the following maintenance tasks: traceability link recovery, feature location, program comprehension and duplicate bug report identification. In most cases, this information was determined by the authors of this paper by reading the title, abstract, keywords, and if necessary the introduction and the conclusion of the investigated paper.

The *exclusion* criteria were as follows. First, we discarded techniques that could not have been implemented effectively in TraceLab due to various reasons, such as (i) lack of sufficient implementation details, (ii) lack of tool availability or (iii) the technique was not fully automated, and would require interaction with the user. Second, we did not implement complex techniques that would have required a lengthy

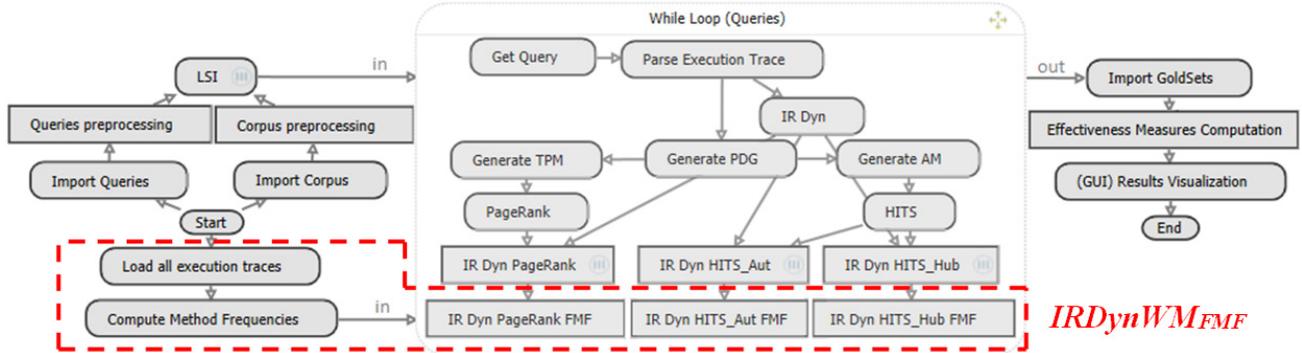


Figure 4 TraceLab experiment for reproducing the *IRDyWMM* FLT [23] (without the components highlighted with red dashed border). TraceLab experiment for implementing *IRDyWMM<sub>FMF</sub>* (all components, including the highlighted ones)

development time, or techniques that are outside the expertise of the authors. Third, we discarded techniques with numerous dependencies to deprecated libraries or other techniques, as our goal was to implement the most popular techniques that can be incorporated or built upon.

#### 4) Classification

In our mapping study we used two independent levels of classification. The first one consisted of categorizing the papers based on the type of technique (e.g., traceability link recovery, feature location, program comprehension and detecting duplicate bug reports) they presented (see Section IV.3)). The second level of classification was identifying common functionality between the basic building blocks used in an approach (e.g., all the functionality related to identifier splitting, stemming, stopwords removal and others, were grouped under "preprocessing").

#### 5) Data extraction

The list of papers that we identified in our study is presented in Table II in the first column along with the Google Scholar citation count as of August 9, 2013 (second column). The papers are grouped by their primary maintenance tasks they address, and are sorted chronologically.

The remaining columns constitute the individual building blocks and components we identified in each approach, grouped by their common functionality. A checkmark (✓) denotes that we implemented the component in the *CL*. An *X* denotes that the code related to the components appears in the approach, but is not implemented in the *CL* at this time (see Section V.B and Section VII).

Table II shows only a subset of the information. For the complete information, we refer the interested reader to our online appendix.

## V. COMPONENT LIBRARY AND DEVELOPMENT KIT

From the 27 papers identified in the mapping study, we reverse engineered their techniques in order to create a comprehensive library of components and techniques with the aim of providing the necessary functionality that SM researchers would need to reproduce experiments and create new techniques.

This process resulted in generating (i) a *Component Development Kit (CDK)* that contains the implementation of all the SM techniques from the study, (ii) a *Component Library (CL)* that adapts the *CDK* components to be used in TraceLab

and (iii) the associated documentation and usage examples for each.

### A. Component Development Kit

The *Component Development Kit (CDK)* is a multi-tiered library of common tools and techniques used in SM research. These tools are organized in a well-defined hierarchical structure and exposed through a public API. The intent of this compilation is to aid researchers in reproducing existing approaches and creating new techniques. By providing these tools in a clear manner, the *Component Development Kit* facilitates the research evaluation process - researchers no longer have to start from scratch or spend time adapting their pre-existing tools to a new project. Furthermore, researchers can use combinations of these tools to create new techniques and drive new research.

At the top level, the *CDK* is separated into high-level tasks, such as I/O, preprocessing techniques, artifact comparison techniques, and metrics calculations (see Figure 3). Those levels are then further broken down as needed into more specific tasks. This design aids technique developers in locating relevant functionality quickly and easily, as well as providing base points for integrating new functionality in the future.

Based on our findings from the mapping study, we evaluated each technique based on coverage, usefulness, and perceived difficulty and effort in implementation. In addition to our design goals of providing a clean and easy to use API, another goal was to minimize the number of external dependencies necessary to implement the technique. As such, some techniques that have numerous external dependencies were left out.

### B. Component Library

The *Component Library (CL)* is comprised of metadata and wrapper classes registering certain functionality as components in TraceLab. It acts as a layer in between TraceLab and the *CDK*, adapting the functionality of the *CDK* to be used within TraceLab.

To register a component in TraceLab, a class must inherit from the *BaseComponent* abstract class in the TraceLab SDK. All components have a *Compute()* method which contains the desired functionality of the component within the context of a TraceLab experiment. Furthermore, all components have a component declaration attribute (or annotation in Java terminology) that describes information about the component,

Table II Mapping study results (first column) and implementation of these techniques in the *CDK* (✓ represents the component is implemented in *CDK* and X means is not yet implemented in *CDK*)

Technique Year / Venue / Name / Ref	Google Scholar Citation Count	Preprocessing										Artifact Comparison					Metrics		Postprocessing				Other				
		Bag-of-words tokenizer	Stopwords Remover	Porter stemmer	CamelCase splitter	Execution trace logger	Dependency Graph Generator	Samurai splitter	smoothing filter	Snowball Stemmer	Part-of-speech tagger	Latent Semantic Indexing	Vector Space Model	Latent Dirichlet Allocation	Jensen-Shannon divergence	Relational Topic Model	HITS	PageRank	Precision / Recall metrics	Effectiveness Measure	Principal Component Analysis	Execution trace extractor		Affine transformation	O-CSTI	UD-CSTI	
<b>Traceability Link Recovery</b>																											
2008.ICPC.Abadi [24]	50	✓	✓	✓	.	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2009.ICPC.Capobianco [25]	24	✓	✓	✓	✓	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2010.ICPC.Oliveto [26]	63	✓	✓	✓	.	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2010.ICSE.Asuncion [27]	76	✓	✓	✓	.	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2011.ICPC.DeLucia [28]	11	✓	✓	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2011.ICSE.Chen [29]	4	✓	.	.	.	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.
2011.ICSM.Gethers [30]	24	✓	✓	✓	✓	.	.	.	.	.	✓	✓	.	.	.	.	✓	✓	.	.	.	✓	.	.	.	.	.
2013.CSMR.Panichella [31]	NA	✓	✓	.	.	.	.	✓	.	.	.	✓	.	.	.	.	✓	✓	.	.	.	.	✓	.	.	.	.
2013.ICSE.Panichella [32]	5	✓	.	.	.	.	.	.	.	.	.	✓	.	.	.	.	✓	✓	.	.	.	.	✓	.	✓	.	✓
2013.TEFSE.Dit [33]	2	✓	.	.	.	.	.	.	.	.	.	✓	.	.	.	.	✓	✓	.	.	.	.	.	.	.	.	✓
<b>Feature Location</b>																											
2004.WCRE.Marcus [34]	260	✓	.	.	✓	.	.	.	.	.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.	.	.	.
2007.ASE.Liu [35]	96	✓	✓	.	✓	X	.	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2007.TSE.Poshyvanyk [36]	191	✓	✓	.	✓	.	.	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2009.ICPC.Revelle [37]	33	✓	.	.	.	X	✓	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2009.ICSM.Gay [38]	39	✓	✓	✓	✓	.	.	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2011.ICPC.Dit [39]	23	✓	✓	✓	✓	X	.	X	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2011.ICPC.Scanniello [40]	8	✓	✓	✓	✓	.	✓	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2011.ICSM.Wiese [41]	4	✓	✓	✓	.	.	.	✓	.	.	✓	✓	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2012.ICPC.Dit [42]	9	✓	✓	✓	✓	X	.	.	.	.	✓	✓	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2013.EMSE.Dit [23]	6	✓	✓	✓	✓	X	✓	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
<b>Program Comprehension</b>																											
2009.MSR.Enslen [43]	57	✓	.	.	✓	.	.	X	.	.	.	.	.	.	.	.	✓	.	✓	.	.	.	.	.	.	.	.
2009.MSR.Tian [44]	34	✓	✓	✓	✓	.	.	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2010.ICSE.Haiduc [45]	27	✓	✓	✓	✓	.	.	.	.	.	✓	.	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2012.ICPC.DeLucia [46]	5	✓	✓	✓	✓	.	.	.	.	.	✓	✓	✓	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
<b>Identify Duplicate Bug Rep.</b>																											
2007.ICSE.Runeson [47]	161	✓	✓	✓	.	.	.	.	.	.	.	✓	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2008.ICSE.Wang [48]	169	✓	✓	✓	.	X	.	.	.	.	✓	✓	.	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
2012.CSMR.Kaushik [49]	7	✓	✓	✓	.	.	.	.	.	.	✓	✓	✓	.	.	.	✓	.	✓	.	.	✓	.	.	.	.	.
Total:	1,388	27	20	17	14	6	4	2	1	1	1	14	14	7	5	1	1	1	14	7	2	5	3	1	1	1	2

such as its name, description, inputs, and outputs. This declaration allows the class to be registered in TraceLab as a component, and ensures that a component can only be connected with a compatible component. A typical component will import data from TraceLab's data sharing interface (the *Workspace*), call various functions on the data using the *CDK*, and then store the results back to the *Workspace*.

The structure of the *Component Library* mirrors the *CDK* hierarchy, providing a mapping from TraceLab to the *CDK*. Components can be organized in TraceLab through the use of developer and user *Tags*, another feature of the TraceLab SDK. Components are grouped via *Tags* into the same high-level tasks as the *CDK*.

From the building blocks of the *CDK* identified in the mapping study, we implemented 25 out of 51 as TraceLab components. In many cases, this was done as a one-to-one mapping from the *CDK* to the *CL*. However, some techniques

could be broken down into more general ones which were desirable for component re-use. For example, the Vector Space Model (VSM) is a straightforward technique, but there can be many variations on its implementation (see Section II). We implemented a few weighting schemes (*e.g.*, binary term frequency, *tf-idf*, and *log-idf*) and similarity functions (*e.g.*, cosine, Jaccard), which a component developer could pick and choose from the desired schemes.

Another example is the precision and recall metrics in traceability link recovery. Although this component consists of only one column in the mapping study, the *CDK* covers many of the commonly used metrics in the literature (*e.g.*, precision, recall, average precision, mean average precision, F-measure, and precision-recall curves). Component developers could choose from any of these measures in their experiments.

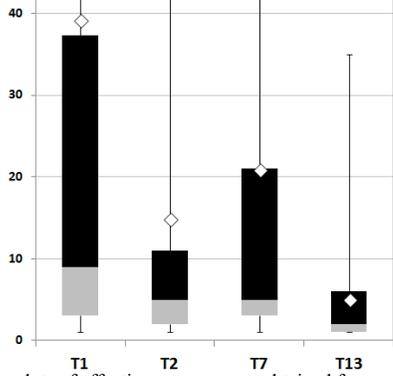


Figure 5 Box plots of effectiveness measure obtained from reproducing the experiments in [23]. The results of techniques  $T_1$ ,  $T_2$ ,  $T_7$  and  $T_{13}$  correspond to  $IR_{LSI}Dyn_{bin}$ ,  $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{80}$ ,  $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{60}$ ,  $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{80}$  from [23] Figure 4(c)

### C. Documentation

Documentation of the *CDK* and *CL* plays a key role in assisting researchers and component developers new to TraceLab. In addition to code examples and API references, documentation provides vital information about a program's functionality, design, and intended use. This adds a wealth of knowledge to someone who wants to use TraceLab and start designing new experiments from components. We provide this information in a wiki format on our website<sup>2</sup>.

### D. Extending the *CDK* and *CL*

The *CL* and *CDK* themselves are not the definitive collection of all the SM tools that researchers will ever need. However, their design and implementation in conjunction with TraceLab's framework provide a foundation for extending SM research in the future.

The *CL* and *CDK* are released under an Open Source license (GPL) in order to facilitate collaboration and community contribution. As new techniques are invented, they can be added to the existing hierarchy and thus into TraceLab.

In creating the *CL* and *CDK*, we leveraged TraceLab's ability to create (custom / user made) components through the TraceLab SDK. As the body of SM techniques grows, researchers can utilize our components and extend them to new ones via the same process. Part of our future work will be investigating effective ways of incorporating user-made components into the *CL* and *CDK*.

## VI. REPRODUCING EXISTING EXPERIMENTS AND EVALUATING NEW IDEAS USING THE COMPONENT LIBRARY

This section presents the details of reproducing an existing feature location technique (FLT) [23] using the *CDK* and the *CL* proposed in this paper. We describe the original technique, the details of reproducing it in TraceLab, and compare the results of the original and reproduced technique. In addition, we illustrate the process of experimenting with two new ideas that are based on the reproduced technique.

### A. Reproducing a Feature Location Technique

The FLT introduced by Dit *et al.* [23], called  $IR_{LSI}Dyn_{bin}WM$  (or *IRDynWM* for short), was reproduced in

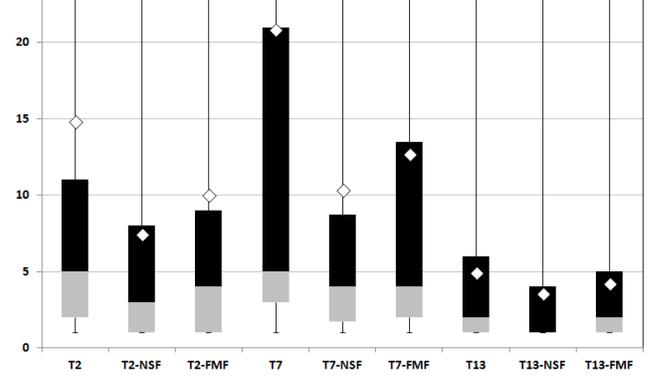


Figure 6 Box plots of effectiveness measure comparing (i) the techniques that produced the best results in [23] for the jEdit4.3 dataset (e.g.,  $T_2$ ,  $T_7$ , and  $T_{13}$  - see Figure 5 for exact names) against the (ii) No Scenario Filter (suffix NSF) and (iii) Frequent Methods Filter (FMF)

TraceLab using a subset of components from the proposed *CL*. The high-level idea behind *IRDynWM* is to (i) identify a subset of methods from an execution trace with high or low rankings using advanced web mining analysis algorithms and to (ii) remove those methods from the results produced by the SITIR approach [35]. The SITIR approach (or *IRDyn*) uses information retrieval (*IR*) techniques to rank all the methods from an execution trace (*Dyn*) based on their textual similarities to a maintenance task used as a query.

The *IRDynWM* FLT takes as input a description of a maintenance task in natural language (e.g., bug report description), the source code of the system, and an execution trace of a scenario that exercises the feature described in the maintenance task. The execution trace is processed and converted into a program dependence graph (PDG), where a pair of connected nodes represents a caller-callee relation between two methods from the execution trace. The PDG is used as an input for two link analysis algorithms, namely PageRank [50] and HITS [51], which generate a score for each node from the PDG (*i.e.*, each method from the execution trace). PageRank produces one score for each method, which represents the popularity or importance of that method within the graph [50]. HITS produces two scores for each method: (i) an authority score, based on the content of the method and the number of methods pointing to it (*i.e.*, methods that are called by other methods should have a higher authority score), and (ii) a hub score, based on the outgoing links of a method (*i.e.*, methods that call numerous other methods have higher hub values) [51]. The different scores produced by PageRank and HITS are used to rank methods and identify the ones with high or low importance scores in order to remove them from the list of results produced by the SITIR (*IRDyn*) approach [35].

The reproduced *IRDynWM* FLT, where  $WM = \{PageRank \text{ or } HITS_{Aut} \text{ or } HITS_{Hub}\}$ , is presented as a TraceLab experiment in Figure 4 (see components in the upper part of the figure, which are not highlighted by the red dashed line).

The experiment uses the *loop structure* introduced in the latest version of TraceLab to iterate through all the queries in the dataset and (i) retrieves and parses its execution trace (*Parse Execution Trace*), (ii) generates a program dependence graph based on the caller-callee relations identified in the trace (*Generate PDG*), (iii) generates a transition probability matrix for PageRank (*Generate TPM*) and applies PageRank

<sup>2</sup> <http://coest.org/coest-projects/projects/semeru/wiki>

Table III Descriptive statistics for the box plots presented in Figure 6. The first row (Percentage Features) represents the percentage of features for which the technique was able to locate at least one relevant method

	$T_2$	$T_2$ -NSF	$T_2$ -FMF	$T_7$	$T_7$ -NSF	$T_7$ -FMF	$T_{13}$	$T_{13}$ -NSF	$T_{13}$ -FMF
<b>Percentage Features</b>	68%	59%	64%	73%	59%	68%	67%	59%	66%
<b>Min</b>	1	1	1	1	1	1	1	1	1
<b>25<sup>th</sup></b>	2	1	1	3	1.75	2	1	1	1
<b>Median</b>	5	3	4	5	4	4	2	1	2
<b>75<sup>th</sup></b>	11	8	9	21	8.75	13.5	6	4	5
<b>Max</b>	237	81	145	170	141	142	35	40	35
<b>Mean</b>	14.81	7.47	10.02	20.85	10.36	12.72	4.92	3.56	4.25
<b>Standard Deviation</b>	34.24	11.71	21.18	32.98	19.68	21.45	6.19	5.96	5.51

(PageRank) to generate the importance scores, and similarly, it generates an adjacency matrix (*Generate AM*) used by HITS (HITS) to generate the authorities and hubs scores associated with these methods. The parsed methods from the execution trace are used by the *IR Dyn* component to produce the results of the SITIR approach, and these results, along with the results produced by the *PageRank* component, are used to generate the results for the *IRDynPageRank* FLT (see *IR Dyn PageRank* component). Similarly, using the HITS authorities and hubs scores, the *IRDynHITS\_Aut* and *IRDynHITS\_Hub* FLTs are computed. It is important to note that the components associated with the *IRDynWM* FLT can be configured with user defined thresholds for the percentage of methods to filter [23].

The results produced by the replicated technique are the same as the ones reported in the original paper, even though the original technique used different implementations of LSI, PageRank and HITS algorithms, as well as other scripts to compute the results. Figure 5 shows a subset of the results produced by our TraceLab implementation, which are the same as the ones that were reported in [23] Figure 4(c) for the jEdit dataset. Figure 5 represents the box plots of the effectiveness measure for the techniques  $T_1$ ,  $T_2$ ,  $T_7$  and  $T_{13}$  corresponding to  $IR_{LSI}Dyn_{bin}$ ,  $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{t80}$ ,  $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{b60}$ ,  $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{b80}$  from [23] Figure 4(c), using the notation from [23]. For simplification,  $T_2$ ,  $T_7$ , and  $T_{13}$  correspond to the *IRDynPageRank*, *IRDynHITS\_Aut* and *IRDynHITS\_Hub* respectively. These techniques were chosen as an example in Figure 5 because they produced the best results for the jEdit dataset [23] and to illustrate that the implemented technique produces the same results as the original technique.

### B. Experimenting with new Ideas

Using the *IRDynWM* FLT [23] as a starting point we experimented with incorporating new ideas for further improving the results. We describe the two new ideas and present their results in comparison with the original ones.

#### 1) Filtering Frequent Methods from Execution Traces

The first idea that we instantiated in TraceLab consisted of filtering out some of the "noise" found in execution traces. More specifically, given a set of execution traces we identify the methods that appear in more than  $X\%$  (*i.e.*, a user defined threshold) of execution traces and we filter them out from the results produced by the *IRDynWM* technique. For example, consider our jEdit 4.3 dataset [23, 52] which contains 150 execution traces generated while exercising particular scenarios. Based on specified threshold (*e.g.*, 66%) we (i)

identified the methods that appear in 100 traces or more, and we (ii) filtered them out of the results produced by *IRDynWM*. Our intuition was that if a particular method captured in an execution trace appears in a large number of traces, the probability of that method to be part of a specific feature is low and therefore, could be eliminated. In a way, this filtering technique is similar to the process of eliminating stop words from corpora, where the stop words were identified as appearing frequently and carrying no real meaning in the corpus.

We implemented this idea based on the existing *IRDynWM*, which resulted in the *IRDynWM\_{FrequentMethodFilter}* or *IRDynWM\_{FMF}* technique (see the bottom part highlighted with a red rectangle in Figure 4). The implementation required the following steps. First, we added two new components to (i) examine all the execution traces from the dataset (component *Load All Execution Traces*) and (ii) identify the methods that appear in more than  $X\%$  of traces, with  $X\%$  being the threshold specified by the user (component *Compute Method Frequencies*). Second, for each query in the while loop we instantiated the same component three times to filter out the most frequent execution trace methods from each technique in the original experiment. For example, the results produced by the *IRDynPageRank* FLT were used as input for the *IRDynPageRank\_{FMF}* (see section VI.B.3) for results).

#### 2) Filtering "No Scenario" Methods from a Trace

In case a large set of execution traces is not available (*i.e.*, the prerequisite for *IRDynWM\_{FMF}* is not satisfied), a developer can use only one execution trace to get improved results, by collecting an execution trace that exercises *no scenario* (*i.e.*, without exercising any specific features of the software). The execution trace was collected from the moment the application started to the moment the application terminated, without exercising any user features in the meantime. The methods captured in the *No Scenario* trace were filtered from the results produced by *IRDynWM*, resulting in the *IRDynWM\_{NoScenarioFilter}* or *IRDynWM\_{NSF}* technique. The intuition behind this idea is that the *No Scenario* trace contains a number of methods that are not associated with any specific scenario (*i.e.*, generic methods), which can be filtered in order to improve the results.

The implementation of this technique is similar to the one presented in Figure 4, but for brevity, the diagram is not included in this paper. The major modification was that the *Load All Execution Traces* and *Compute Method Frequencies* components were replaced with a component that loaded a user-specified *no scenario* execution trace and extracted the methods that will be filtered. In addition, the *IRDynWM\_{FMF}*

composite nodes were replaced with corresponding  $IRDynWM_{NSF}$  composite nodes.

### 3) Results of the New Ideas

Figure 6 shows side by side the box plots of the effectiveness measure produced by  $IRDynWM$ ,  $IRDynWM_{FMF}$  and  $IRDynWM_{NSF}$ . For the comparison, we choose the best three configurations of PageRank, HITS Authority and HITS Hubs that produced the best results for the jEdit dataset in [23], which are  $T_2$ ,  $T_7$  and  $T_{13}$  (see Figure 5 for the labels). A complementary view of Figure 6 is given by Table III, which contains descriptive statistics of the box plots generated by those techniques.

Figure 6 and Table III show that the  $IRDynWM_{FMF}$  (e.g.,  $T_2-FMF$ ,  $T_7-FMF$  and  $T_{13}-FMF$ ) techniques generate better results in terms of the effectiveness measure than  $IRDynWM$ , and that  $IRDynWM_{NSF}$  produces better results than  $IRDynWM_{FMF}$ . For example, for  $T_2$ , the median value was 5, whereas for  $T_2-FMF$  and  $T_2-NSF$  the median values for 4 and 3 respectively. The same trend is observed for the average values: 14.81, 10.02 and 7.47 for  $T_2$ ,  $T_2-FMF$  and  $T_2-NSF$ , respectively.

Our two experimental ideas produced better results than the best results presented in [23] for the jEdit dataset. However, the improvement in "precision", comes at the cost of potentially filtering out relevant methods. For example in Table III row *Percentage Features* shows the percentage of features for which that particular technique was able to identify at least one relevant method. As the table indicates, filtering additional methods removes noise (i.e., irrelevant methods to the feature), as well as some relevant methods.

### C. Discussion

Although for this particular dataset the two experimental ideas produced better results than the ones reported in [23], there is still more research to be done (e.g., investigate the impact of removing also relevant methods, automatically setting the threshold for  $IRDynWM_{FMF}$ , ensuring generalizability, considering more advanced techniques for analyzing traces [53, 54], etc.) before considering these ideas as viable techniques, but this is beyond the scope of this paper.

The main goal of these examples was to illustrate the support that our *Component Library* and the TraceLab framework can offer to researchers, who can test new ideas and get some preliminary results to assess the feasibility of those ideas, and decide if it is worth pursuing them or not.

## VII. LIMITATIONS

This section discusses some potential limitations for conducting research using TraceLab, the *Component Development Kit* and the *Component Library*.

TraceLab was not designed for real-time feedback from the user, and although this could be implemented it would impose some overhead upon the developer. Additionally, running code hosted in a .NET process is slower than running it natively. Therefore the time or speed factors in evaluating an approach would need to be considered.

We attempted to identify papers which covered a number of topics in SM, which we were familiar with or had expertise with. Within the papers we covered, in some cases we were unable to obtain exact implementations due to lack of specific details or availability of tools. Additionally, many experiments

cannot be reproduced directly because the datasets under study were undisclosed or unavailable.

The *CL* and *CDK* do not implement every technique and building block found in the mapping study. The amount of time, manpower, and testing required to do so would be far beyond the resources available. That being said, we tried to implement as many of the techniques that we could in order to show the efficacy and usefulness of TraceLab as a research tool. We are continuously working on driving new research with TraceLab and encourage others to do so as well.

A major issue that prevented us from using or implementing certain tools was their copyright licensing. In some cases they do not use permissive licenses, and even if the source code was available its license did not permit distribution. TraceLab is released under the open source license GPL, which we follow as well with the *CL* and *CDK*. Developers may release their own components under any license they wish, but if they wish to extend or modify the *CL* or *CDK*, they must release under GPL as well.

## VIII. CONCLUSIONS AND FUTURE WORK

This paper addressed the *reproducibility* problem associated with experiments in SM research. Our goal was to support and accelerate research in SE by providing a body of actionable knowledge in the form of reproduced experiments and a *Component Library* and *Component Development Kit* that can be used as the basis to generate novel, and most importantly reproducible techniques.

After conducting a mapping study of SM techniques in the areas of traceability link recovery, feature location, program comprehension and duplicate bug report detection, we identified 27 papers and techniques that we used to generate a library of TraceLab components. We implemented a subset of these techniques as TraceLab experiments to illustrate TraceLab's potential as a research framework and to provide a basis for implementing new techniques.

It is obvious that this does not cover the entire range of SM papers or techniques. Therefore, in the future, we are determined to continually expand the TraceLab *Component Library* and *Development Kit* by including more techniques and expanding it to other areas of SM (e.g., impact analysis). In addition, we encourage other researchers to contribute to this body of knowledge for the benefit of conducting research.

## ACKNOWLEDGMENT

This work is supported in part by the United States NSF CNS-0959924, CCF-1218129, and CCF-1016868 grants. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. We would also like to acknowledge the team of researchers from DePaul University: Jane Cleland-Huang, Ed Keenan, Adam Czauderna, and Greg Leach. This work would not have been possible without their continuous support on the TraceLab project.

## REFERENCES

- [1] B. Dit, M. Revelle, M. Gethers, and D. Poshvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process (JSEP)*, vol. 25, pp. 53–95, 2013.

- [2] G. Robles, "Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings," in *MSR*, 2010, pp. 171-180.
- [3] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "The Effect of Omitted-Variable Bias on the Evaluation of Compiler Optimizations," *IEEE Computer*, vol. 43, pp. 62-67, 2010.
- [4] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison," *Empirical Software Engineering (ESE)*, vol. 17, pp. 531-577, 2012.
- [5] E. Barr, C. Bird, E. Hyatt, T. Menzies, and G. Robles, "On the Shoulders of Giants," in *FoSER*, 2010, pp. 23-28.
- [6] J. M. González-Barahona and G. Robles, "On the Reproducibility of Empirical Software Engineering Studies based on Data Retrieved from Development Repositories," *Empirical Software Engineering (ESE)*, vol. 17, pp. 75-89, 2012.
- [7] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012). *The PROMISE Repository of Empirical Software Engineering Data*. Available: <http://promisedata.googlecode.com>
- [8] S. J. Sayyad and T. J. Menzies. (2005 July 17). *The PROMISE Repository of Software Engineering Databases*. Available: <http://promise.site.uottawa.ca/SERpository>
- [9] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *PROMISE*, 2007.
- [10] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, pp. 405-435, 2005
- [11] J. Cleland-Huang, A. Czaundera, A. Dekhtyar, G. O., J. Huffman Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshvyanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder, "Grand Challenges, Benchmarks, and TraceLab: Developing Infrastructure for the Software Traceability Research Community," in *TEFSE*, 2011.
- [12] E. Keenan, A. Czaundera, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshvyanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hussein, and D. Hearn, "TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions," in *ICSE*, 2012, pp. 1375-1378.
- [13] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czaundera, G. Leach, E. Moritz, M. Gethers, D. Poshvyanyk, J. H. Hayes, and W. Li, "Toward Actionable, Broadly Accessible Contests in Software Engineering," in *ICSE*, 2012, pp. 1329-1332.
- [14] Rapid-I. *Rapid Miner*. Available: <http://rapid-i.com/content/view/181/190/>
- [15] Mathworks. *Simulink*. <http://www.mathworks.com/products/simulink/>
- [16] U. of California. *The Kepler Project*. Available: <https://kepler-project.org/>
- [17] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM (CACM)*, vol. 18, pp. 613-620, 1975.
- [18] T. U. of Waikato. *WEKA*. <http://www.cs.waikato.ac.nz/ml/weka/>
- [19] Yahoo. *Yahoo Pipes*. Available: <http://pipes.yahoo.com/pipes/>
- [20] T. U. of Sheffield. *GATE*. Available: <http://gate.ac.uk/>
- [21] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using Mapping Studies as the Basis for Further Research - A Participant-Observed Case Study," *Information and Software Technology*, vol. 53, pp. 638-651, 2011.
- [22] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering," in 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), 2008.
- [23] B. Dit, M. Revelle, and D. Poshvyanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *Empirical Software Engineering*, vol. 18, pp. 277-309, 2013.
- [24] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in *ICPC*, 2008, pp. 103-112.
- [25] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *ICPC*, 2009, pp. 148-157.
- [26] R. Oliveto, M. Gethers, D. Poshvyanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," in *ICPC*, 2010, pp. 68-71.
- [27] H. Asuncion, A. Asuncion, and R. Taylor, "Software Traceability with Topic Modeling," in *ICSE*, 2010, pp. 95-104.
- [28] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based Traceability Recovery Using Smoothing Filters," in *ICPC*, 2011, pp. 21-30.
- [29] X. Chen, J. Hosking, and J. Grundy, "A Combination Approach for Enhancing Automated Traceability," in *ICSE*, 2011, pp. 912-915.
- [30] M. Gethers, R. Oliveto, D. Poshvyanyk, and A. De Lucia, "On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Link Recovery," in *ICSM*, 2011, pp. 133-142.
- [31] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshvyanyk, and A. De Lucia, "Using Structural Information and User Feedback to Improve IR-based Traceability Recovery," in *CSMR*, 2013, pp. 199-208.
- [32] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshvyanyk, and A. De Lucia, "How to Effectively Use Topic Models for Software Engineering Tasks? An Approach based on Genetic Algorithms," in *ICSE*, 2013, pp. 522-531.
- [33] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. Di Penta, D. Poshvyanyk, and A. De Lucia, "Configuring Topic Models for Software Engineering Tasks in TraceLab," in *TEFSE*, 2013, pp. 105-109.
- [34] A. Marcus, A. Sergeev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *WCRE*, 2004, pp. 214-223.
- [35] D. Liu, A. Marcus, D. Poshvyanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in *ASE*, 2007, pp. 234-243.
- [36] D. Poshvyanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering (TSE)*, vol. 33, pp. 420-432, 2007.
- [37] M. Revelle and D. Poshvyanyk, "An Exploratory Study on Assessing Feature Location Techniques," in *ICPC*, 2009, pp. 218-222.
- [38] G. Gay, S. Haiduc, M. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-Based Concept Location," in *ICSM*, 2009, pp. 351-360.
- [39] B. Dit, L. Guerrouj, D. Poshvyanyk, and G. Antoniol, "Can Better Identifier Splitting Techniques Help Feature Location?," in *ICPC*, 2011, pp. 11-20.
- [40] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," in *ICPC*, 2011, pp. 1-10.
- [41] A. Wiese, V. Ho, and E. Hill, "A Comparison of Stemmers on Source Code Identifiers for Software Search," in *ICSM*, 2011, pp. 496-499.
- [42] B. Dit, E. Moritz, and D. Poshvyanyk, "A TraceLab-based Solution for Creating, Conducting, and Sharing Feature Location Experiments," in *ICPC*, 2012, pp. 203-208.
- [43] E. Enslin, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining Source Code to Automatically Split Identifiers for Software Analysis," in *MSR*, 2009, pp. 71-80.
- [44] K. Tian, M. Revelle, and D. Poshvyanyk, "Using Latent Dirichlet Allocation for Automatic Categorization of Software," in *MSR*, 2009, pp. 163-166.
- [45] S. Haiduc, J. Aponte, and A. Marcus, "Supporting Program Comprehension with Source Code Summarization," in *ICSE*, 2010, pp. 223-226.
- [46] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR Methods for Labeling Source Code Artifacts: Is it Worthwhile?," in *ICPC*, 2012, pp. 193-202.
- [47] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *ICSE*, 2007, pp. 499-510.
- [48] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information," in *ICSE*, 2008, pp. 461-470.
- [49] N. Kaushik and L. Tahvildari, "A Comparative Study of the Performance of IR Models on Duplicate Bug Detection," in *CSMR*, 2012, pp. 159-168.
- [50] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *7th Int. Conference on World Wide Web*, 1998, pp. 107-117.
- [51] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *Journal of the ACM*, vol. 46, pp. 604-632, 1999.
- [52] B. Dit, A. Holtzhauer, D. Poshvyanyk, and H. Kagdi, "Generating Benchmarks from Change History Data to Support Evaluation of Software Maintenance Tasks," in *MSR Data Track*, 2013, pp. 131-134.
- [53] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *Transactions on Software Engineering (TSE)*, vol. 29, pp. 116 - 132, 2003.
- [54] T. Eisenbarth, R. Koschke, and D. Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces," presented at the IWPC, 2001, pp. 300-309.

# Social Activities Rival Patch Submission For Prediction of Developer Initiation in OSS Projects

Mohammad Gharehyazie  
Computer Science Department  
University of California, Davis  
Davis, CA 95616  
gharehyazie@ucdavis.edu

Daryl Posnett  
Computer Science Department  
University of California, Davis  
Davis, CA 95616  
dposnett@ucdavis.edu

Vladimir Filkov  
Computer Science Department  
University of California, Davis  
Davis, CA 95616  
filkov@cs.ucdavis.edu

**Abstract**—Maintaining a productive and collaborative team of developers is essential to Open Source Software (OSS) success, and hinges upon the trust inherent among the team. Whether a project participant is initiated as a developer is a function of both his technical contributions and also his social interactions with other project participants. One’s online social footprint is arguably easier to ascertain and gather than one’s technical contributions *e.g.*, gathering patch submission information requires mining multiple sources with different formats, and then merging the aliases from these sources. In contrast to prior work, where patch submission was found to be an essential ingredient to achieving developer status, here we investigate the extent to which the likelihood of achieving that status can be modeled solely as a social network phenomenon. For 6 different OSS projects we compile and integrate a set of social measures of the communications network among OSS project participants and a set of technical measures, *i.e.* OSS developers patch submission activities. We use these sets to predict whether a project participant will become a developer. We find that the social network metrics, in particular the amount of two-way communication a person participates in, are more significant predictors of one’s likelihood to becoming a developer. Further, we find that this is true to the extent that other predictors, *e.g.* patch submission info, need not be included in the models. In addition, we show that future developers are easy to identify with great fidelity when using the first three months of data of their social activities. Moreover, only the first month of their social links are a very useful predictor, coming within 10% of the three month data’s predictions. Finally, we find that it is easier to become a developer earlier in the projects lifecycle than it is later as the project matures. These results should provide insight on the social nature of gaining trust and advancing in status in distributed projects.

## I. INTRODUCTION

Open Source Software (OSS) are developed by volunteers who are often geographically and temporally distributed, and yet work together effectively and productively. Well known examples of successful OSS projects, like the Linux operating system, Apache web server, and many others, rival or even exceed the quality of commercial competitors.

Participants in OSS projects have different roles, most prominently developers, patchers and users. Developers in-

troduce changes to the code by *committing* directly to the project’s *source code repository*; they have the highest level of access to the project and also share the greatest responsibility of delivering a viable product. Patchers submit proposed changes in the form of small updates to the code, known as *patches*, such as bug fixes, feature improvements, or documentation, which then are reviewed by developers and added to the project code at their discretion. Users are the downstream consumers of OSS. Project participants may migrate between these roles through the life of a project. This process has received a lot of attention in the empirical software research literature where it is variously referred to as developer initiation [1], entering the circle of trust [1], migration [2], and immigration [3]. A typical trajectory to becoming a developer is to start communicating with other participants in the project and then gradually get more involved, by earning a more central position in the project’s social networks and/or producing more valuable technical contributions, like submitting patches, or working on bug activities [3].

The congruence of one’s social and technical activities makes for a successful participation in a project [4]. In a sense, a project participant is as integral to the project as their contributions, be they communications or bug fixes. Strong working code leads to trust in the participant’s ability to develop within the context of the project, and additionally, strong social skills signify to the team that the participant can be trusted to work within the project’s team setting. The more trustworthy a participant, the more likely it is that he may eventually achieve developer status. Generally, only those participants who have sufficiently proven themselves through their activities, become developers.

Developer initiation, thus, depends on the social and technical actions of project participants, *e.g.*, who they talk to, the number of social links they have with other project participants, their communication patterns, patch submission activity, bug identification and fixing, *etc.*. But to what extent do social activities and technical activities work together to increase one’s chance of advancing to the ranks of developers?

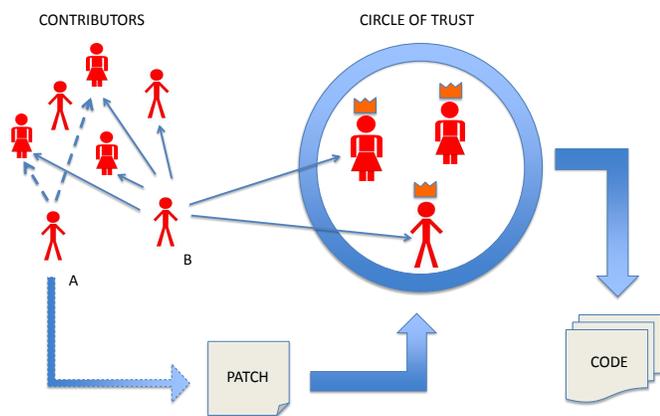


Figure 1. An OSS project community’s circle of trust is comprised of developers. Project participants A and B are candidates for entering the circle. The activities of Person B are more significant positive predictors of him gaining developer status because he communicates more (thin solid arrows) than A does (thin dashed arrows), even though A contributes patches and B doesn’t.

In this paper, we revisit the issue of migration between roles in OSS projects studying it from a social network analysis perspective. We collect developer communication, patch submissions, and code change data for 6 projects from the Apache Software Foundation (ASF). From the data we build statistical predictors for future developers in OSS projects and compare models with different predictors, as well as models across different time periods. We find that:

- Developer initiation in OSS can be modeled very well as a social network phenomenon based solely on people’s communication activities, in particular the number of (two-way) social links they establish, *i.e.* messages participants *respond to* or *receive*, in response to their own message. The (two-way) social links are different from a one-way communication, most of which might not attract any attention or response. Social links based models exhibit better predictive ability for developer initiation than models incorporating patch submissions.
- Whether a participant will eventually become a developer can be predicted with great accuracy from the first three months of their tenure with the project. In most cases, this is based solely on the number of their (two-way) social links. Furthermore, models learned on only a single month of data containing the social links that one establishes, yields good predictions results that are within 10% of the three month data.
- The impact of both social and technical participation declines with project age. It becomes more difficult to attain developer status as the project matures.

The rest of the paper is organized as follows. We first focus

on the background behind OSS development and migration and present our research questions. Then, we describe the data and data gathering process, followed by our methods, results, and conclusion sections.

### A. Background

To become a new developer, a participant must first gain acceptance within the community. Trust in the developer’s technical and social skills is believed to increase with time as a result of increased contribution and interaction by new participants [5]. Participation indicators, including the number of emails sent, their degree in the social network, the number of developers among a participant’s neighbors, the numbers of bugs reported, and the number of patches submitted, may all be indicators of the community’s trust in the candidate’s abilities.

In most cases, to become a developer in an ASF project a new participant will first engage in discussion with other users and developers by joining the developer mailing list. Actively contributing interesting discussions and valuable insights to the project are key ways in which a new user can attract the attention of existing developers and gain social reputation within the community. The activity and the reputation of a user can be quantified through measures such as the number of emails he sends and the degree of the corresponding nodes in the email social network [6], [7].

Submitting patches is the preferred way in which unverified code changes are communicated in many OSS projects. Submitting a bug fix patch provides a basic degree of evidence that a user understands the software at a technical level. Such contributions should increase a participant’s trustworthiness within the community. The contribution of the user can be quantified, *e.g.*, by the number of patches accepted.

In the ASF open source community, once a user has contributed sufficiently to a project, they may be nominated for developer status by an existing developer. The existing developers may then choose to grant this user committer status, allowing the new developer to make direct changes to the project’s source code repository. Hence, whether a participant will eventually become a developer is a function of all the participants social and technical activities within the project’s ecosystem.

Figure 1 illustrates two contributors, A and B, who want to become developers, but have two different profiles of activities. Contributor A contributes patches, and has a low level of communication with other participants, while B does not submit patches but communicates extensively with other participants. Our results in this paper show that B’s activities are more significant for predicting his future developer status than are those of A, respectively.

It is important to note that this is just the basic framework of how a user becomes a developer in ASF projects, and the situations may vary for different projects in different stages of their evolution. For example, some projects may require the

users to enter the bugs into a bug tracking database, such as *Bugzilla* or *Jira*, while others may require users to submit bug reports to a mailing list. The latter method may increase the interaction between users and developers in these projects, and thus affect the dynamics of earning trust. Additionally, users may have different motivations to join a project [8]–[10], *e.g.*, enjoyment, reputation building, and skill improvement. The commercial backers of some open source projects may also provide incentives for skilled programmers to contribute in order to grow their project in its early stages, while in the later stages, when the project has matured, developers are often more willing to volunteer in order to gain a signaling benefit to prospective employers [9].

### B. Research Questions

In this paper we seek to identify effective social activity predictors of developer initiation in OSS projects, and to improve upon existing models for predicting future developers. Specifically, we ask which social metrics are effective, how do they interact with the technical measures of patch activities, and how soon can we predict that a participant will become a developer?

While the data on mailing list communications within projects are readily available, as are code commits and changes, patches and bug identification data is more difficult to gather [11] since it requires parsing message texts and mining multiple sources of predicting developers both when patch information is available and also when it is not.

**Research Question 1:** To what extent can developer initiation in OSS projects be modeled as a function of patch activities and social communication? How about just as a function of social communications?

Early in their tenure as OSS project participants, people’s patterns of technical and social activities are rapidly changing. Participants usually take some time to familiarize themselves with the code before submitting patch fixes. Additionally, they also might try to assimilate the project culture and available knowledge initially before asking questions of their own. Therefore, predictions of future status may be unreliable early in a person’s tenure. Here, we seek to predict project participants’ likelihood of becoming a developer based on the patterns of their activities early in their tenure.

**Research Question 2:** How well can we predict if a person will become a developer based on information early in their tenure with the project? *i.e.* can we tell if someone will become a developer based on their activities in the first three months? Six months? Or just one month?

Finally, as projects evolve, determinants of trust are also likely to evolve and change, consequently, measures of trust

and predictors of status change may not be static. This is mirrored in other fields, *e.g.* clandestine operations, where communication is over public channels but action traces are rarely readily observable. While the number of developers in a project grows proportional to its size, the number of participants in a project’s mailing list grows exponentially. This increasing gap between potential developers and new position openings in turn change how trust is earned as a project matures.

**Research Question 3:** Is it easier or more difficult to become a developer later in the project?

## II. RELATED WORK

There have been a fair number of studies on the motivations of developers for joining OSS projects and migration in OSS projects. Some projects have clear guidelines on how a new participant can contribute. The structure of this hierarchical process is known in the literature as the “onion model” [8], [12].

Von Krogh *et al.* performed a detailed case study of the freenet project; they interviewed participants and developers recording their patterns of individual activity and concluded that individuals following these guidelines are highly more likely to become developers [13]. Ducheneaut examined a single individual and his process of promotion to a core developer in the Python project [14]. Jensen and Scacchi studied role sets and the process of role migration in Mozilla, ASF and Netbeans [2].

Sinha *et al.* studied how developers enter the “circle of trust” by identifying key factors that lead to committer status [1]. They hypothesized that developers who contribute to the projects’ bug tracking system, have prior experience contributing code to OSS, and who work for the same organization as some member of the core group, are more likely to obtain committer status.

The path to becoming a developer is not necessarily a step-by-step process. Herraiz *et al.* found that apart from gradual progression, there is another common developer joining pattern, *viz.*, the quick initiation of employees of enterprises invested in that OSS project as new developers [15]. Shibuya and Tamai performed case studies and confirmed these findings on other OSS projects (GNOME, OpenOffice.org, MySQL) [16].

Qureshi and Fang have identified different classes of developers based on socialization patterns using Growth Mixture models. They found that for each class of social behavior, the “Lead Time” *i.e.* the time it takes to become a developer, is unique and correlates with the amount of social activity of that class [17].

Bird *et al.* quantitatively modeled the relationship between time spent with the project and the probability of becoming a developer; their model used patch activities, social network attributes, and the time to first commit from the time of first

communication on an email network [3]. Using proportional hazard rate modeling they observed that a developer’s *tenure* is related to his skill and commitment as measured by his participation in the email network and his contribution of patches prior to first commit. Further, they identified and described a non-monotonic trend in the likelihood of becoming a developer that rises with tenure, peaks, and then declines with project maturity. While their work is similar to ours in some aspects, their approach of predicting the “time” until one becomes a developer is in contrast to our work in that we focus on identifying “who” is more likely to become a developer rather than “when”. The hazard analysis techniques used in their work support their approach.

Zhou and Mockus have modeled the status of “Long Term Contributors” based on three dimensions: environment, willingness, and capacity [18]. Their work focuses on issue tracking systems and workflows within those systems as sources of information.

Our work differs in that we focus on understanding how soon can we predict developer initiation after initial participation and, additionally, to what degree can social metrics replace technical attributes.

### III. DATA GATHERING

In this study we focus on six projects from the Apache Software Foundation. Their summary is given in Table I. For each project we mined three datasets, the *developer* mailing list, the issue/bug tracking systems and the source code repository.

From the developer mailing lists, we construct the Email Social Networks, *ESN*. We extract patch submissions from both the issue tracking systems and the developer mailing lists. Lastly, we obtain the history of source code changes from the source code repository. All of the sources were mined from the first date of available data, until the date of mining (March 2012), and the dates in Table I denote the intersection of available data in all three datasets.

We extract patch submissions from both the issue tracking systems and the developer mailing lists. Lastly, we obtain the history of source code changes from the source code repository.

TABLE I. OSS projects used in this study show diversity both in size and in relative activity. #Users refers to the number of individuals in the ESN and #Devs refers to the number of distinct developers in each project’s source code repository.

Project	#Users	#Devs	#Mails	#Patches	Start	End
Ant	1416	44	17300	1482	2000-01	2012-03
Axis2_c	600	24	11152	754	2004-01	2012-03
Log4j	539	18	3811	166	2000-12	2012-03
Lucene	2155	41	43922	5576	2001-09	2012-02
Pluto	266	24	3017	259	2003-10	2011-09
Solr	840	19	14411	4090	2006-01	2010-04

#### A. Email Social Networks

It is a general policy for OSS projects to channel as much communication as possible through project mailing lists so that all participants can benefit from the exchange of ideas and information [11]. Any message sent to these lists will be broadcast to all subscribed participants. Such broadcast messages differ from point to point email messages in that they do not have a clear and distinct recipient. Even given this difficulty, however, we can still infer a communication network. When person *A* sends message *M*, in response to message *N*, that was sent by person *B*, then there is a high chance that *A* primarily intended to communicate to *B* in response to their initial message that was broadcast to no particular recipient [11]. Such links are in fact two-way social links, in that the communication occurs both ways. Self-loops were removed when constructing ESNs. We note that the final network is a *multigraph*, since we permit multiple edges between people.

1) *Unmasking Aliases*: Project participants often use different aliases within the same project, *e.g.*, (John Smith, smith@gmail.com), (Smith, John@smith.com), (John S., J.smith@ucdavis.edu). Since these aliases represent a single person they must be merged if we are to accurately capture an individual’s social activity. Unmasking aliases is a well-known problem in the literature [11], [19], [20]. Bird *et al.* used a string similarity based approach to address this problem [11]. We improved and automated their procedure reducing the need for human interaction. We first remove all suffixes, prefixes, and generic names, *e.g.*, Dr., Mr., Jr., Admin, for comparison. Then, for each alias pair a similarity score is calculated as a combination of four sub-scores comparing aliases’ names and emails. Perfect matches are automatically merged, whereas less than perfect matches that achieve a minimum threshold of 0.93 on a unit scale are presented to the researcher to disambiguate. The chosen threshold was selected empirically to reduce the number of false positives while limiting false negatives.

#### B. Issue Tracking Systems

Issue tracking systems such as Jira and Bugzilla maintain a database of issue reports submitted by developers and users. Issues are of various types including new features, improvements, or defects. Developers can submit new issues to the tracking system or report results on existing issues by interacting with the system either through the web, or in some cases, directly via the source code repository.

#### C. Patch Extraction

Patches are the preferred method of fixing bugs or issues and/or making small adjustments to the code. Patch submission is not limited to developers, and submitting a patch does not imply that the mentioned patch will be applied. It is up to developers to review the patches and apply them if they are accepted.

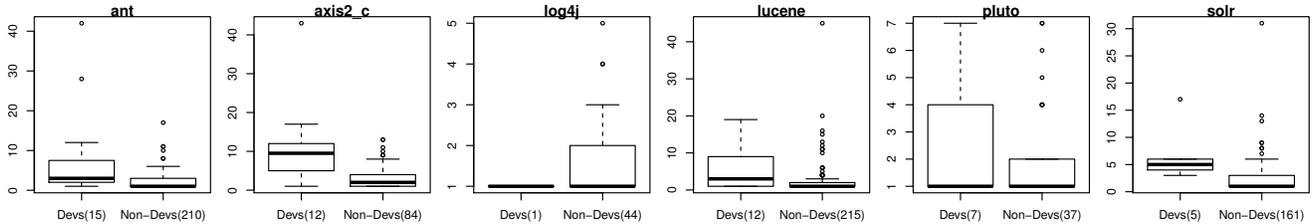


Figure 2. Distribution of number of patch submissions by developers and non-developers in each project. Only individuals with at least one patch submission are plotted, since adding those with no submission would highly skew the plots towards zero. The numbers in the parentheses show each group’s population. A Wilcoxon signed rank test across each project yields p-values in order: 0, 0.01, 0.44, 0, 0.72, 0, indicating that in Log4j and Pluto, patch submission is not statistically different for those participants who become developers and those who don’t when measured in the first three months of participation.

There are multiple methods and formats in which one can submit a patch to an OSS project. The common accepted format for a patch submission is to send the “diff” of the code changes to existing developers. ASF provides two popular choices of issue tracking systems for OSS projects: Jira and Bugzilla. Patches can be submitted as attachments to an issue, or simply as inline text, *i.e.* pasting the patch “diff” code, either as part of the issue’s description text, or as a comment on that specific issue.

The developer mailing lists can also act as a medium for patch submission [21]. In this case patches are submitted through messages to the mailing list, either with the code pasted in the text message or as an attachment.

To extract inline patches, we used regular expression pattern matching queries in mailing list messages, and issues/bugs comments and description. Patches submitted as attachments are generally named with a “.diff” or “.patch” extension. However, we found that sometimes patch submissions do not follow this naming convention, e.g. “patch.txt” is used, or multiple patch files are combined in an archive, requiring the extraction of that file, and subsequent manual examination of all filenames contained within the archive.

Table II shows the distribution of patch submission among the multiple sources of patches in each project.

#### D. Source Code Repository

A source code repository and version control system, *e.g.* Git, SVN, and CVS, facilitates collaboration among developers by maintaining a history of changes and an associated log entry for each change. These systems can provide information such

TABLE II. Different projects have different paths for patch submission. The preferred method of patch submission is differs across projects and we attribute this to project culture.

	ML_inline	ML_attach	bugzilla_attach	jira_inline	jira_attach
Ant	413	976	93	0	0
Axis2_c	63	200	0	27	464
Log4j	87	57	19	0	3
Lucene	141	107	0	88	5240
Pluto	15	26	0	8	210
Solr	30	4	0	48	4008

as list of files, list of developers and a detailed record of all changes to all files by any developer. This information allows us to distinguish developers from non-developers in the ESNs and to track their history of activity.

All selected OSS projects for this study use Git as their version control system, however they initially used SVN migrating later to Git, Some projects continue to maintain both repositories.

Since this dataset and the ESN use separate sets of aliases (not necessarily identical) as personal identifiers, a manual matching between people across the two datasets was performed.

## IV. METHODOLOGY AND MODELING

### A. Social Communication Measures

For each participant in our datasets, we gather the following metrics to model the relationship between their social and technical characteristics to their potential developer status.

- *Project age* The number of days from the start of the project where a node has its first edge in the graph. This is the first message received by or replied by that person in the ESN.
- *Is a New Developer*: A binary variable indicating whether this person ever commits into the repository *after* they have joined the mailing list. Those who were developers prior to this time are beyond the scope of this paper as we are interested in the process of attaining developer status.
- *Number of Patches*: The number of patches one has submitted.
- *Number of Messages*: The number of edges connected to a node in the multigraph *i.e.* a node’s degree. This is not just the number of messages one sends, but rather the number of messages one **sends in response to** others plus the number of messages one **receives in response to** their messages.
- *Number of Threads*: The number of threads started by a person. A thread is a message that is *not* in response to other messages. Threads are not included in the ESN due to inability to associate them to a specific pair of nodes.

TABLE III. Patch submission is a significant predictor when no social variables are included in the model. This basic logistic regression model only uses “number of patches” in 3 months and includes “project age” as a control variable. For all variables, the log of that variable plus 0.5 was used in the modeling. The values in the first column are the model coefficients and the highlighted coefficients are statistically significant ( $p < 0.05$ ).

Ant	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	-1.73	1.12	-1.55	0.12
Project age	-0.33	0.18	-1.87	0.06
Number of patches	<b>1.06</b>	0.18	5.82	0
Axis2_c	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	-0.91	1.01	-0.9	0.37
Project age	<b>-0.39</b>	0.16	-2.46	0.01
Number of patches	<b>0.93</b>	0.21	4.53	0
Log4j	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	-3.53	1.98	-1.78	0.07
Project age	-0.11	0.29	-0.38	0.7
Number of patches	0.36	0.83	0.43	0.67
Lucene	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	1.08	0.84	1.28	0.2
Project age	<b>-0.7</b>	0.12	-5.71	0
Number of patches	<b>1.16</b>	0.18	6.42	0
Pluto	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	-0.88	0.84	-1.05	0.3
Project age	<b>-0.34</b>	0.15	-2.3	0.02
Number of patches	<b>1.11</b>	0.32	3.45	0
Solr	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	0.44	1.19	0.37	0.71
Project age	<b>-0.72</b>	0.19	-3.69	0
Number of patches	<b>0.75</b>	0.29	2.58	0.01

- *Neighbors*: The number of unique nodes that a node is connected to. This is different from number of messages in that a node can connect to other nodes through multiple edges thus differing these two measurements.
- *Neighbor Developers*: The number of unique nodes with developer status that a given node is connected to.

### B. Modeling Developer Initiation

We use logistic regression, a generalized linear model designed to model probabilities for dichotomous outcomes, to model whether or not a project participant will become a developer based on several social explanatory variables.

Previous work in this area has used survival modeling to model the trajectory over time of developer initiation [3]. In this work we are focused on early detection of developer initiation which limits the amount of data available to model the trajectory. Moreover, since we are interested in the dichotomous outcome of whether a participant becomes a developer, logistic regression is a more appropriate choice.

The dataset used for our studies, contains the features and metrics described above for the first  $k$  months, of each individual’s activity ( $k = 3$  unless explicitly stated). We attempt to predict “*Is a New Developer*”. The variables used for each of the models are described in the results. The project age at which one joins that project, is added to all models as a control variable. For all numeric variables, the log of that

TABLE IV. The second logistic regression model, adding “number of messages” to the previous model. It is seen that “number of patches” slightly loses its significance.

Ant	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	<b>-4.32</b>	1.32	-3.28	0
Project age	-0.2	0.19	-1.08	0.28
Number of patches	<b>0.61</b>	0.2	3.06	0
Number of messages	<b>1.07</b>	0.2	5.35	0
Axis2_c	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	<b>-2.85</b>	1.33	-2.15	0.03
Project age	-0.3	0.17	-1.81	0.07
Number of patches	<b>0.53</b>	0.25	2.11	0.03
Number of messages	<b>0.57</b>	0.22	2.62	0.01
Log4j	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	<b>-7.28</b>	2.51	-2.9	0
Project age	-0.13	0.3	-0.42	0.67
Number of patches	-0.8	0.98	-0.82	0.41
Number of messages	<b>1.89</b>	0.44	4.26	0
Lucene	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	0.26	0.89	0.29	0.77
Project age	<b>-0.8</b>	0.13	-6.04	0
Number of patches	<b>0.69</b>	0.22	3.14	0
Number of messages	<b>0.74</b>	0.2	3.75	0
Pluto	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	-1.44	1	-1.44	0.15
Project age	<b>-0.37</b>	0.15	-2.46	0.01
Number of patches	0.81	0.42	1.95	0.05
Number of messages	0.42	0.4	1.04	0.3
Solr	Estimate	Std. Error	z value	Pr(>  z )
(Intercept)	<b>-2.86</b>	1.45	-1.97	0.05
Project age	<b>-0.67</b>	0.2	-3.39	0
Number of patches	-0.15	0.35	-0.44	0.66
Number of messages	<b>1.18</b>	0.31	3.83	0

variable plus 0.5 was used to stabilize variance and reduce heteroscedasticity [22]. Since all untransformed values in our data are skewed, the increase in the value of a variable by one unit does not have the same effect at high values as it does in low values, *e.g.*, the number of patches changing from 1 to 2 is much more meaningful than changing from 100 to 101.

Since we are trying to predict who *is going to* become a developer, sample data for developers who were initiated in fewer than  $k$  months were removed from the dataset.

For each learned model, we evaluate its validity based on two criteria. The first is project independence, *i.e.* we want our model to hold across projects. In support of this, we look at the stability of each model coefficient’s statistical significance, as determined by the coefficient’s *p-value*. Commonly, a *p-value* of less than 0.05 is an indicator of significant results.

Excessive multicollinearity is a concern in regression models and it can occur when predictors are highly correlated. To check for this we use the *Variance Inflation Factor (VIF)*. A common rule of thumb is that for any variable  $x$  in a model,  $VIF(x) \geq 5$  indicates high collinearity. In all of our models in this paper, VIF of all variables remained well below 2 except for some models with highly correlated variables. These models were discarded as it will be explained later in the paper.

To evaluate a model’s predictive power, we use the *Area Under the Receiver Operating Characteristic (AUROC)* measure [22]. ROC illustrates the performance of a binary classi-

fier in a TP-FP space, while varying the cutoff threshold. A random predictor would be a line with the slope of 1 and the area of 0.5 while a perfect predictor will have an area of 1.

Overfitting is a concern with any statistical model so to help alleviate any concern and to yield a stable estimate of the predictive power of our models we employ resampling methods. We define training and testing sets using 2/3 holdout for our training sets. To maintain a similar distribution of developers vs. non developers in the training and testing sets we employ stratified sampling. Each of the test and training sets will then have roughly the same ratio of developers to non-developers. This ensures that the resulting model is not extremely biased in that the training set would contain almost all, or none of the developers. The first case would cause a testing set with no positive samples, and the latter would result in a zero model due to the lack of positive samples in the training set. We resample 250 times and average the AUROC over all these models to indicate the overall predictive power of a model.

## V. RESULTS AND DISCUSSION

### A. Research Question 1

We evaluate here the stability and predictive power of models using patches, length of time with the project, and a number of social measures. We motivate this question with Fig. 2. This figure shows that, although most of the time there is a meaningful difference between developers and non-developers who submit patches, still there are many non-developers who behave like developers in terms of patch submission. Consequently, it is likely that using patches alone may not yield the best prediction model.

This simple model, using “Number of patches” and no social network measures shows that patch submission is often a statistically significant predictor (Table III). However, adding “number of messages” (as an indicator of social collaboration) to this model results in patches slightly losing their significance (Table IV), We used a Chi-Squared goodness of fit statistic to verify that the additional predictor explained a statistically significant amount of the deviance in the model, *viz.*, is the addition of the new variable justified. For all projects projects except Pluto adding “number of messages” was significant with a Chi-Squared test p-value of  $< 0.01$ . A Spearman correlation test between “number of messages” and “number of patches” does not show significant correlation between them, mostly below 0.3 over all projects, in concordance with our VIF values of less than 2.

The predictive power of these two models and the additional models with only “Number of messages” and the combination “Number of messages + Threads” is shown in Fig. 3. We see that not only adding number of messages dramatically improves the predictive power, but removing the patches variable from the model does not lower the predictive power of the model. A Kruskal-Wallis test followed by a post-hoc pairwise Wilcoxon test for each project reveals that in

TABLE V. Spearman’s Correlation between different variables in all projects. Correlation values higher than 0.5 are highlighted.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
messages vs. dev neighbors	<b>0.62</b>	<b>0.52</b>	0.49	<b>0.55</b>	<b>0.53</b>	<b>0.65</b>
neighbors vs. dev neighbors	<b>0.69</b>	<b>0.61</b>	<b>0.60</b>	<b>0.72</b>	<b>0.72</b>	<b>0.80</b>
messages vs. neighbors	<b>0.82</b>	<b>0.76</b>	<b>0.70</b>	<b>0.67</b>	<b>0.66</b>	<b>0.75</b>
threads vs. dev neighbors	0.21	0.38	0.12	0.38	0.25	<b>0.56</b>
threads vs. neighbors	0.31	0.54	0.14	0.37	0.25	0.48
threads vs. messages	0.31	<b>0.74</b>	0.17	<b>0.61</b>	<b>0.53</b>	<b>0.64</b>

all projects except Pluto and Lucene either the models with messages or messages and threads have the highest mean AUC, and this difference is statistically significant. In Lucene, the best model uses both patches and messages. In Pluto, although the patches model has the highest AUC, there is no statistical difference between the models. Using patch information alone is not a bad predictor, but it is evident that using social network metrics yields more accurate predictions.

We ask next whether we can improve this simple model by adding additional features from the ESN. Since we have observed that sending and receiving messages is an important indicator of whether someone will become a developer or not, naturally we ask whether it is the number of messages that is important or the number of distinct individuals one keeps in contact with? More precisely, are these contacts the same, or is communicating with developers more important than communicating with other participants? Also we want to see whether starting threads and discussions in contrast to replying and being replied to, is also an important factor in gaining the trust of the community.

These variables are quite highly correlated and we expect that this will impact model performance (The spearman correlation between these variables can be seen in Table V). We added the number of started threads, neighbors and neighboring developers to our existing models. While some predictors are statistically significant in some models as can be seen in Table VII, most are hampered by high variance

TABLE VI. Social metrics yield better performing predictive models for developer status across most projects. Mean AUC values over 250 runs using stratified sampling for each project. Italicized values indicate models that include an insignificant variable in the explanatory model. Values in bold are the highest mean AUC value over all models that remained significant after a post-hoc pairwise Wilcoxon test out of all explanatory models with significant variables. Projects that do not have a value in bold were statistically indistinguishable.

project	patches	patches+msgs	msgs	msgs+threads
Ant	0.71	0.87	0.87	<b>0.89</b>
Axis2_c	0.83	0.84	0.83	<i>0.82</i>
Log4j	0.30	0.75	<b>0.91</b>	<i>0.88</i>
Lucene	0.84	<b>0.85</b>	0.83	<i>0.84</i>
Pluto	0.79	0.77	0.76	<i>0.74</i>
Solr	0.76	0.90	0.91	<b>0.96</b>

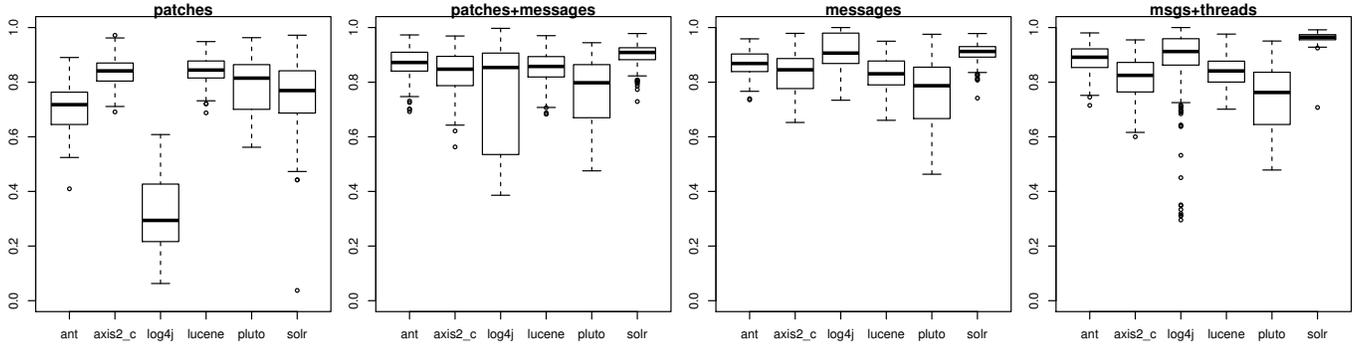


Figure 3. Social measures outperform patch submission in a predictive setting. AUROC of 4 models, on 250 iterations of modeling using stratified data.

TABLE VII. High multicollinearity limits the effectiveness of additional social variables. None of the added social network measures are stable across projects. Number of threads is significant in two projects, ant, and solr.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	<b>-4.13</b>	<b>-4.24</b>	<b>-6.28</b>	-0.27	<b>-2.49</b>	-2.78
Number of messages	<b>1.2</b>	<b>0.9</b>	<b>1.67</b>	<b>1</b>	0.58	<b>1.01</b>
Number of neighbor devs	0.08	-0.26	0.25	0.05	0.72	0.27
Project age	-0.29	-0.21	-0.17	<b>-0.83</b>	<b>-0.33</b>	<b>-0.65</b>
(Intercept)	<b>-5.64</b>	<b>-3.78</b>	<b>-6.87</b>	-0.39	<b>-2.04</b>	<b>-5.33</b>
Number of messages	<b>1.04</b>	<b>0.82</b>	<b>1.51</b>	<b>1.14</b>	<b>0.69</b>	<b>2.74</b>
Number of threads	<b>0.69</b>	-0.01	0.6	-0.15	0.28	<b>-1.26</b>
Project age	-0.17	-0.26	-0.12	<b>-0.82</b>	<b>-0.43</b>	<b>-0.77</b>
(Intercept)	<b>-5.7</b>	<b>-4.71</b>	<b>-16.87</b>	-1.66	<b>-2.66</b>	<b>-5.64</b>
Number of messages	<b>1.01</b>	0.21	-2.09	0.25	0.23	<b>2.21</b>
Number of threads	<b>0.69</b>	0.12	<b>1.07</b>	-0.03	0.34	<b>-1.24</b>
Number of neighbors	0.05	1.12	<b>7.3</b>	<b>1.58</b>	0.89	1.08
Project age	-0.16	-0.15	0.9	<b>-0.67</b>	<b>-0.34</b>	<b>-0.74</b>
(Intercept)	<b>-5.63</b>	<b>-4.28</b>	<b>-7.06</b>	-0.39	<b>-2.26</b>	<b>-5.56</b>
Number of messages	<b>1.03</b>	<b>0.94</b>	1.15	<b>1.12</b>	0.34	<b>2.6</b>
Number of threads	<b>0.69</b>	-0.04	0.68	-0.14	0.35	<b>-1.27</b>
Number of neighbor devs	0.02	-0.27	1.02	0.03	0.85	0.37
Project age	-0.17	-0.21	-0.11	<b>-0.82</b>	<b>-0.39</b>	<b>-0.72</b>

inflation factor owing to the high correlation between “Number of Messages”, “Number of neighbors”, and “Number of developers” Table V. “Number of threads”, however, was significant in two projects, Ant, and Solr, and had sufficiently low variance inflation factor that inclusion of the variable improved prediction results. We discuss this further in the next subsection.

**Result 1:** *Developer initiation can be modeled using social activity alone, performing no worse than models which also incorporate patch submission. The basic model of social activity only uses “Number of Messages”, however adding “Number of Threads” improved prediction results in 2 of the projects, hinting this might be a matter of “project culture”.*

TABLE VIII. Number of messages is a statistically significant predictor with as little as only one month of data. Stability of models with log of number of messages, for 1 to 6 months. Models using two or three months time window are slightly more stable across all projects

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	<b>-3.04</b>	<b>-2.97</b>	<b>-2.94</b>	0.55	-1.43	0.42
Messages in 1 month	<b>1.09</b>	<b>0.73</b>	<b>1.43</b>	<b>0.72</b>	0.49	0.48
Project age	<b>-0.34</b>	-0.27	<b>-0.45</b>	<b>-0.81</b>	<b>-0.36</b>	<b>-0.84</b>
(Intercept)	<b>-3.63</b>	<b>-3.64</b>	<b>-5.8</b>	-0.09	<b>-1.8</b>	-0.92
Messages in 2 months	<b>1.2</b>	<b>0.83</b>	<b>1.63</b>	<b>0.87</b>	<b>0.46</b>	<b>0.83</b>
Project age	-0.32	-0.24	-0.17	<b>-0.79</b>	<b>-0.31</b>	<b>-0.78</b>
(Intercept)	<b>-4.15</b>	<b>-3.77</b>	<b>-6.3</b>	-0.25	<b>-2.24</b>	-2.64
Messages in 3 months	<b>1.24</b>	<b>0.81</b>	<b>1.76</b>	<b>1.02</b>	<b>0.84</b>	<b>1.11</b>
Project age	-0.29	-0.26	-0.17	<b>-0.83</b>	<b>-0.38</b>	<b>-0.67</b>
(Intercept)	<b>-4.9</b>	<b>-3.27</b>	<b>-6.75</b>	-0.83	<b>-3.24</b>	<b>-3.13</b>
Messages in 4 months	<b>1.4</b>	<b>0.68</b>	<b>1.77</b>	<b>1.13</b>	<b>0.91</b>	<b>1.31</b>
Project age	-0.24	-0.32	-0.15	<b>-0.81</b>	-0.27	<b>-0.72</b>
(Intercept)	<b>-6.05</b>	<b>-3.51</b>	<b>-7.74</b>	-0.92	<b>-4.15</b>	<b>-3.37</b>
Messages in 5 months	<b>1.42</b>	<b>0.7</b>	<b>1.86</b>	<b>1.13</b>	<b>0.96</b>	<b>1.32</b>
Project age	-0.1	-0.32	-0.1	<b>-0.82</b>	-0.18	<b>-0.72</b>
(Intercept)	<b>-6.73</b>	<b>-3.48</b>	<b>-7.78</b>	-0.98	<b>-4.46</b>	<b>-3.57</b>
Messages in 6 months	<b>1.4</b>	<b>0.69</b>	<b>1.8</b>	<b>1.1</b>	<b>1.07</b>	<b>1.41</b>
Project age	0	-0.34	-0.08	<b>-0.82</b>	-0.19	<b>-0.77</b>

## B. Research Question 2

In the previous section we used information on the first three months of individuals’ activity to model their likelihood of obtaining committer status. But how early can such models provide useful predictions? Is one month of information sufficient, or should we increase to 6 months or more to yield better prediction models?

A limitation in evaluating models for longer time periods is that participants who become developers in a shorter period must be discarded from the training set, yielding a smaller dataset and consequently a less reliable model. The median time to become a developer in our OSS projects ranges from 8 months to almost a year (except for Log4j which is 4 months), and a range from 1 to 6 months includes more than half of the developers in the dataset.

To evaluate the sufficient-time-for-prediction hypothesis, we use the simple model discussed in the previous section (only

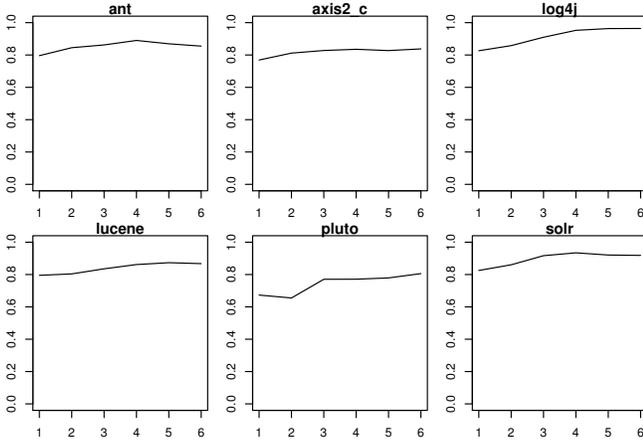


Figure 4. The predictive power of the model using “number of messages” from 1 to 6 months, each on 250 iterations of modeling using stratified data. The AUROC for each project slightly improves until 3rd or 4th month, and then stabilizes.

using “Number of Messages” as a predictor) with information on the first  $n = 1, 2, \dots, 6$  months of each participant’s activity in the OSS’ ESN. The results can be seen in Table VIII. For one or two months the models are not as significant as other models. But afterwards all the models are statistically significant, valid, and surprisingly stable.

Model stability only tells one part of the story, *viz.*, it can explain how the model fits the data. However, there is always risk of over-training and evaluation of the predictive power of the models will more effectively demonstrate the value of this model in a realistic setting. We see in Fig. 4 that the predictive powers of the models differ slightly from one time window to another. Time windows less than 3 months slightly suffer from low predictive power and time windows of greater than 4 months are almost no better than 3 or 4 months. We choose 3 months as our default because of best overall stability and predictive power (4 months is almost just as good, but with our goal of prediction, the smaller the time window, the better).

Additionally, adding the number of threads to our model improved the prediction results in two projects. Fig. 3 (bottom) and Fig. 5 show that adding the number of threads to the model slightly improves prediction performance.

**Result 2:** *Developer initiation can be modeled with as little as one month’s information about the social activity of individuals; using three months yields stronger and more stable result.*

### C. Research Question 3

We are also interested in understanding how trust evolves over the life of each project. Looking back at previous models, we see that in all cases, the coefficient for “Project Age” is negative, implying it becomes increasingly difficult to become a developer over time. To verify this hypothesis, we

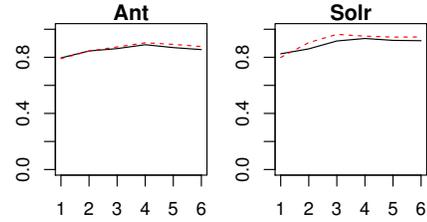


Figure 5. The AUROC of two projects with two different models. Black lines represent models using only “Number of messages” while red dashed lines represent models with “Number of Threads” added to them. The latter performs slightly better than the former.

replaced “Project Age” with a dummy binary variable called “IsSecond” which is true for the second half of population (sorted in ascending order by their “Project Age”) and is FALSE for the first half. If the coefficient of this variable is still negative across projects it will confirm our hypothesis that being initiated a developer becomes increasingly more difficult over time. Ideally “Project Age” should be broken to smaller partitions (4 or more) to give us a higher resolution view, but increasing the resolution would result in even less sample points in each partition, making the results less reliable.

Two different logistic regression models were fit to the data, and the models are given in Table IX. We see that for all projects, the coefficient of “IsSecond” is negative, although only statistically significant across three of the projects. Based on these observations, we conclude:

**Result 3:** *It becomes more difficult for individuals to become developers as the project matures; late stage developers may have to put more effort to gain the same level of trust.*

## VI. CONCLUSION AND THREATS TO VALIDITY

We presented strong evidence for the determining role that social networking activities play in becoming a developer in an OSS project. Surprisingly, to this end, social communications are a better predictor than patching activity. We also present

TABLE IX. It becomes increasingly more difficult to earn trust in an OSS. Models show a dummy variable “IsSecond” which is true for individuals that  $p\_age > median(p\_age)$ . It is seen that joining the project later has a negative effect on one’s chance of becoming a developer.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	-5.5	-4.21	-7.69	-4.74	-2.99	-6.59
Number of patches	0.62	0.59	-0.76	0.62	0.85	-0.2
Number of messages	1.08	0.56	1.89	0.72	0.49	1.17
IsSecond	-0.36	-2.08	-0.9	-2.46	-2.1	-1.34
(Intercept)	-5.76	-4.93	-7.04	-5.42	-3.8	-6.33
Number of messages	1.24	0.82	1.78	0.99	0.88	1.07
IsSecond	-0.57	-1.84	-0.99	-2.67	-2.01	-1.29

evidence that developers' early social activities in the project identify them as such. Expectedly, we also find that community trust is more difficult to attain with time as the community likely takes longer to identify trustworthy contributors.

Our methods are based solely on (two-way) social links representing messages sent between project participants, but is oblivious to the content of those messages. Clearly, knowing the content of the emails would add another layer of information that can be mined. However, the quality of our predictions while disregarding content is an indication of the strong influence of the social link structure. This may be of independent interest to the management and security communities.

Our results in no way imply causality, rather a strong statistical correlation between the measured attributes that can be used for prediction and further research.

We recognize several threats to the validity of our approach and conclusions. The dataset gathered here was from 6 projects, all from the Apache Software Foundation. This might impose a limitation on the pattern of communication and contribution in these projects that will limit the applicability of our results to other OSS projects. It also may be that there is a systematic bias in our data, meaning what we measure is not the likelihood of obtaining developer status *e.g.* people may be assigned to be developers (rather than being chosen) and are using ESNs to familiarize themselves with the community. Although this assumption is quite contrary to ASF's guidelines [23], it is not hard to imagine other scenarios where developers are not chosen as we think they are.

Having more projects is desirable, but practically, we had to select projects with a large number of developers for the predictive models to have reasonable statistical power. We cannot address private communication between developers which may impact the structure of the social network. This limitation, however, affects all such work of this nature and we do not believe that it severely limits the usefulness of our results.

## VII. ACKNOWLEDGEMENTS

The authors would like to thank Bogdan Vasilescu for his contributions in mining issue tracking systems and mailing lists attachments, and for his insightful comments and feedback on this work. All authors gratefully acknowledge support from the Air Force Office of Scientific Research, award FA955-11-1-0246.

## REFERENCES

- [1] V. Sinha, S. Mani, and S. Sinha, "Entering the circle of trust: developer initiation as committers in open-source projects," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 133–142.
- [2] C. Jensen and W. Scacchi, "Role migration and advancement processes in ossd projects: A comparative case study," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 364–374.
- [3] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," in *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*. IEEE, 2007, pp. 6–6.
- [4] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 2–11.
- [5] K. Stewart and S. Gosain, "An exploratory study of ideology and trust in open source development groups," in *Proceedings of the 22nd International Conference on Information Systems*. ACM, 2001, pp. 1–6.
- [6] M. Newman, S. Forrest, and J. Balthrop, "Email networks and the spread of computer viruses," *Physical Review E*, vol. 66, no. 3, pp. 035 101(R):1–4, 2002.
- [7] C. Fershtman and N. Gandal, "Direct and indirect knowledge spillovers: the "social network" of open-source projects," *The RAND Journal of Economics*, vol. 42, no. 1, pp. 70–91, 2011.
- [8] Y. Ye and K. Kishida, "Toward an understanding of the motivation of open source software developers," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2003, pp. 419–429.
- [9] G. Krogh and E. Hippel, "The promise of research on open source software," *Management Science*, vol. 52, no. 7, pp. 975–983, 2006.
- [10] W. Scacchi, "Free/open source software development: Recent research results and methods," *Advances in Computers*, vol. 69, pp. 243–295, 2007.
- [11] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 137–143.
- [12] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel," *Research Policy*, vol. 32, no. 7, pp. 1159 – 1177, 2003, open Source Software Development. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0048733303000477>
- [13] G. Von Krogh, S. Spaeth, and K. Lakhani, "Community, joining, and specialization in open source software innovation: a case study," *Research Policy*, vol. 32, no. 7, pp. 1217–1241, 2003.
- [14] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 4, pp. 323–368, 2005.
- [15] I. Herraiz, G. Robles, J. Amor, T. Romera, and J. González Barahona, "The processes of joining in global distributed software projects," in *Proceedings of the 2006 international workshop on Global software development for the practitioner*. ACM, 2006, pp. 27–33.
- [16] B. Shibuya and T. Tamai, "Understanding the process of participating in open source communities," in *Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009. FLOSS'09. ICSE Workshop on*. IEEE, 2009, pp. 1–6.
- [17] I. Qureshi and Y. Fang, "Socialization in open source software projects: A growth mixture modeling approach," *Organizational Research Methods*, vol. 14, no. 1, pp. 208–238, 2011.
- [18] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in oss community," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 518–528.
- [19] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload: a case study of the gnome ecosystem community," *Empirical Software Engineering*, pp. 1–54, 2013.
- [20] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "Who's who in gnome: Using lsa to merge software repository identities," in *ICSM*. IEEE Computer Society, 2012, pp. 592–595.
- [21] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in oss projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 26.
- [22] J. Cohen, *Applied multiple regression/correlation analysis for the behavioral sciences*. Lawrence Erlbaum, 2003.
- [23] (2013) Apache software foundation - frequently asked questions. [Online]. Available: <http://www.apache.org/foundation/faq.html>

# How does Context Affect the Distribution of Software Maintainability Metrics?

Feng Zhang\*, Audris Mockus<sup>†</sup>, Ying Zou<sup>‡</sup>, Foutse Khomh<sup>§</sup>, and Ahmed E. Hassan\*

\* School of Computing, Queen's University, Canada

<sup>†</sup> Department of Software, Avaya Labs Research, Basking Ridge, NJ 07920. USA

<sup>‡</sup> Department of Electrical and Computer Engineering, Queen's University, Canada

<sup>§</sup> SWAT, École Polytechnique de Montréal, Canada

feng@cs.queensu.ca, audris@avaya.com, ying.zou@queensu.ca, foutse.khomh@polymtl.ca, ahmed@cs.queensu.ca

**Abstract**—Software metrics have many uses, *e.g.*, defect prediction, effort estimation, and benchmarking an organization against peers and industry standards. In all these cases, metrics may depend on the context, such as the programming language. Here we aim to investigate if the distributions of commonly used metrics do, in fact, vary with six context factors: application domain, programming language, age, lifespan, the number of changes, and the number of downloads. For this preliminary study we select 320 nontrivial software systems from SourceForge. These software systems are randomly sampled from nine popular application domains of SourceForge. We calculate 39 metrics commonly used to assess software maintainability for each software system and use Kruskal Wallis test and Mann-Whitney U test to determine if there are significant differences among the distributions with respect to each of the six context factors. We use Cliff's delta to measure the magnitude of the differences and find that all six context factors affect the distribution of 20 metrics and the programming language factor affects 35 metrics. We also briefly discuss how each context factor may affect the distribution of metric values. We expect our results to help software benchmarking and other software engineering methods that rely on these commonly used metrics to be tailored to a particular context.

**Index Terms**—context; context factor; metrics; static metrics; software maintainability; benchmark; sampling; mining software repositories; large scale.

## I. INTRODUCTION

The history of software metrics predates software engineering. The first active software metric is the number of lines of code (LOC) which was used in the mid 60's to assess the productivity of programmers [1]. Since then, a large number of metrics have been proposed [2], [3], [4], [5], and extensively used in software engineering activities, *e.g.*, defect prediction, effort estimation and software benchmarks. For example, software organizations use benchmarks as management tools to assess the quality of their software products in comparison to industry standards (*i.e.*, benchmarks). Since software systems are developed in different environments, for various purposes, and by teams with diverse organizational cultures, we believe that context factors, such as application domain, programming language, and the number of downloads, should be taken into account, when using metrics in software engineering activities (*e.g.*, [6]). It can be problematic [7] to apply metric-based benchmarks derived from one context to software systems in

a different context, *e.g.*, applying benchmarks derived from small-size software systems to assess the maintainability of large-size software systems. COCOMO II<sup>1</sup> model supports a number of attribute settings (*e.g.*, the complexity of product) to fine tune the estimation of the cost and system size (*i.e.*, source lines of code). However, to the best of our knowledge, no study provides empirical evidence on how contexts affect such metric-based models. The context is overlooked in most existing approaches for building metric-based benchmarks [8], [9], [10], [11], [12], [13], [14].

This preliminary study aims to understand if the distributions of metrics do, in fact, vary with contexts. Considering the availability and understandability of context factors and their potential impact on the distribution of metric values, we decide to investigate seven context factors: application domain, programming language, age, lifespan, system size, the number of changes, and the number of downloads. Since system size strongly correlates to the number of changes, we only examine the number of changes of the two factors.

In this study, we select 320 nontrivial software systems from SourceForge<sup>2</sup>. These software systems are randomly sampled from nine popular application domains. For each software system, we calculate 39 metrics commonly used to assess software maintainability. To better understand the impact on different aspects of software maintainability, we further classify the 39 metrics into six categories (*i.e.*, complexity, coupling, cohesion, abstraction, encapsulation and documentation) based on Zou and Kontogiannis [15]'s work. We investigate the following two research questions:

**RQ1:** *What context factors impact the distribution of the values of software maintainability metrics?*

We find that all six context factors affect the distribution of the values of 51% of metrics (*i.e.*, 20 out of 39). The most influential context factor is the programming language, since it impacts the distribution of the values of 90% of metrics (*i.e.*, 35 out of 39).

<sup>1</sup><http://sunset.usc.edu/csse/research/COCOMOII>

<sup>2</sup><http://www.sourceforge.net>

**RQ2:** *What guidelines can we provide for benchmarking<sup>3</sup> software maintainability metrics?*

We suggest to group all software systems into 13 distinct groups using three context factors: application domain, programming language, and the number of changes, and consequently, we obtain 13 benchmarks.

The remainder of this paper is organized as follows. In Section II, we summarize the related work. We describe the experimental setup of our study in Section III and report our case study results in Section IV. Threats to validity of our work are discussed in Section V. We conclude and provide insights for future work in Section VI.

## II. RELATED WORK

In this section, we review previous attempts to build metric-based benchmarks. Such attempts are usually made up of a set of metrics with proposed thresholds and ranges.

### A. Thresholds and ranges of metric values

Deriving appropriate thresholds and ranges of metrics is important to interpret software metrics [17]. For instance, McCabe [2] proposes a widely used complexity metric to measure software maintainability and testability, and further interprets the value of his metric in such a way: sub-functions with the metric value between 3 and 7 are well structured; sub-functions with the metric value beyond 10 are unmaintainable and untestable [2]. Lorenz and Kidd [3] propose thresholds and ranges for many object-oriented metrics, which are interpretable in their context. However, direct application of these thresholds and ranges without taking into account the contexts of systems might be problematic. Erni and Lewerentz [4] propose to consider mean and standard deviations based on the assumption that the metrics follow normal distributions. Yet many metrics follow power-law or log-normal distributions [18], [19]. Thus many researchers propose to derive thresholds and ranges based on statistical properties of metrics. For example, Benlarbi *et al.* [8] apply a linear regression analysis. Shatnawi [9] use a logistic regression analysis. Yoon *et al.* [10] use a k-means cluster algorithm. Herbold *et al.* [12] use machine learning techniques. Sánchez-González *et al.* [20] compare two techniques: ROC curves and the Bender method.

A recent attempt to build benchmarks is by Alves *et al.* [11]. They propose a framework to derive metric thresholds by considering metric distributions and source code scales, and select a set of software systems from a variety of contexts as measurement data. Baggen *et al.* [21] present several applications of Alves *et al.* [11]’s framework. Bakota *et al.* [13] propose a different approach using probabilities other than thresholds or ranges, and focus on aggregating low-level metrics to maintainability as described in ISO/IEC 9126 standard.

The aforementioned studies do not consider the potential impact by the contexts of software systems. The contexts can

<sup>3</sup>A benchmark is generally defined as “a test or set of tests used to compare the performance of alternative tools or techniques” [16]. In this study, we refer to “benchmark” as a set of metric-based evaluations of software maintainability.

affect the effective values of various metrics [7]. As complements to these studies, our work investigates how contexts impact the distribution of software metric values. We further provide guidelines to split software systems using contexts before applying these approaches (*e.g.*, [9], [10], [11], [12]) to derive thresholds and ranges of metric values.

### B. Context factors

Contexts of software systems are considered in Ferreira *et al.* [14]’s work. They propose to identify thresholds of six object-oriented software metrics using three context factors: application domain, software types and system size (in terms of the number of classes). However, they directly split all software systems using the context factors without examining whether the context factors affect the distribution of metric values or not, thus result in a high ratio of duplicated thresholds. To reduce duplications and maximize the samples of measurement software systems, a split is necessary only when a context factor impacts the distribution of metric values. Different from Ferreira *et al.* [14]’s study, we propose to split all software systems based on statistical significance and corresponding effect size. We study four additional context factors and 33 more metrics than their study.

Open source software systems are characterized by Capiluppi *et al.* [22] using 12 context factors: age, application domain, programming language, size (in terms of physical occupation), the number of developers, the number of users, modularity level, documentation level, popularity, status, success of project, and vitality. Considering not all software systems provide information for these factors, we decide to investigate five commonly available context factors: application domain, programming language, age, system size (we redefined it as the total number of lines), and the number of downloads (measured using average monthly downloads). Since software metrics are also affected by software evolution [23], we study two additional context factors: lifespan and the number of changes.

To the best of our knowledge, we are the first to propose detailed investigation of context factors when building metric-based benchmarks.

## III. CASE STUDY SETUP

This section presents the design of our case study.

### A. Factor Description

In this subsection, we briefly describe the definition and motivation of each factor.

1) *Application Domain (AD)*: describes the type of software systems. In general, software systems designed as *frameworks* might contain more classes than other types of software systems.

2) *Programming Language (PL)*: describes the nature of programming paradigms. Generally speaking, software systems written in Java might have deeper inheritance tree than C++, since C++ supports both object oriented programming and structural programming.

3) *Age (AG)*: is the time duration after creating a software system. As software development techniques evolve fast, older software systems might be more difficult to maintain than newly created software systems.

4) *Lifespan (LS)*: describes the time duration of development activities in the life of a software system. Software systems developed over a long period of time might be harder to maintain than software systems developed over a shorter time period.

5) *System Size (SS)*: is the total lines of code of a software system. Small software systems might be easier to maintain than large ones.

6) *Number of Changes (NC)*: describes the total number of commits made to a software system. It might be more difficult to maintain heavily-modified software systems than lightly-modified ones.

7) *Number of Downloads (ND)*: describes the external quality of a software system. It is of interest to find if popular software systems have better maintainability than less popular ones. In this study, the number of downloads is measured using the average monthly downloads which were collected directly from SourceForge.

## B. Data Collection

**Data Source.** We use the SourceForge data initially collected by Mockus [24]. There are some updates after that work, and the new data collection was finished on February 05, 2010. The dataset contains 154,762 software systems. However, we find 97,931 incomplete software systems which contain fewer than 41 files, and an empty CVS repository has 40 files. There are 56,833 nontrivial software systems in total from SourceForge. FLOSSMole [25] is another data source, from where we download descriptions (*i.e.*, application domain) of SourceForge software systems. Furthermore, we download latest application domain information<sup>4</sup> and monthly download data<sup>5</sup> of studied software systems directly from SourceForge.

**Sampling.** Investigating all the 56,833 software systems requires a large amount of computation resources. For example, the snapshots of our selected 320 software systems occupy about 8 GB hard drive, and the computed metrics take more than 15 GB hard drive. The average time for computing metrics of one software system is 6 minutes. The bottleneck is the slow disk I/O, since we intensively access disks (*e.g.*, to dump snapshots of source code, and to store metric values). Applying SSD or RAID storage or using RAM drive might eliminate this bottleneck. Yet our resource is limited to apply such solution at this moment. For a preliminary study, we perform stratified sampling of software systems by application domains to explore how context factors affect the distribution of metric values. The limitation of stratified sampling is discussed in Section V. Moreover, we plan to stratify by the

<sup>4</sup><http://sourceforge.net/projects/ProjectName> (NOTE: the ProjectName needs to be substituted by the real project name, *e.g.*, a2ixlibrary)

<sup>5</sup>[http://sourceforge.net/projects/ProjectName/files/stats/json?start\\_date=1990-01-01&end\\_date=2012-12-31](http://sourceforge.net/projects/ProjectName/files/stats/json?start_date=1990-01-01&end_date=2012-12-31) (NOTE: the ProjectName needs to be substituted by the real project name, *e.g.*, gusi)

remaining six factors in future. In this exploratory study, we pick nine popular application domains containing over 1,000 software systems. We conduct simple random sampling to select 100 software systems from each application domain and obtain 900 software systems in total. Yet there are only 824 different software systems, since a software system may be categorized into several application domains.

## C. Factor Extraction

1) *Application Domain (AD)*: We extract the application domain of each software system using the data collected in June 2008 by FLOSSMole [25]. We rank all application domains using the number of software systems and pick nine popular application domains: Build Tools, Code Generators, Communications, Frameworks, Games/Entertainment, Internet, Networking, Software Development (excluding Build Tools, Code Generators, and Frameworks), and System Administration. We replace sub-domains, if exist, by their parent application domain.

2) *Programming Language (PL)*: In this study, we only investigate software systems that are mainly written in C, C++, C#, Java, or Pascal. For each software system, we dump the latest snapshot, and determine the main programming language by counting the total number of files per file type (*i.e.*, \*.c, \*.cpp, \*.cxx, \*.cc, \*.cs, \*.java, and \*.pas).

3) *Age (AG)*: For each software system, we compute the age using the date of the first CVS commit. In the sampled 824 software systems, the oldest software system<sup>6</sup> was created on November 02, 1996, and the latest software system<sup>7</sup> was created on May 28, 2008.

4) *Lifespan (LS)*: For each software system, we compute the lifespan by computing the intervals between the first and the last CVS commits. The quantiles of lifespan in a unit of day in the sampled 824 software systems are: 0 (minimum), 51 (25%), 338 (median), 930 (75%), and 4,038 (maximum).

5) *System Size (SS)*: For each software system, we count the total lines of code from the latest snapshot. The quantiles of the total lines of code in the sampled 824 software systems are: 0 (minimum), 1,124 (25%), 3,955 (median), 14,945 (75%), and 2,792,334 (maximum).

6) *Number of Changes (NC)*: For each software system, we count the total number of commits from the whole history. The quantiles of the number of changes in the sampled 824 software systems are: 12 (minimum), 123 (25%), 413 (median), 1,142 (75%), and 94,853 (maximum).

7) *Number of Downloads (ND)*: For each software system, we first sum up all the monthly downloads to get the total number of downloads, and search the first and the last month with at least one download to determine the downloading period. We divide the total downloads by the total number of months of the downloading period to obtain the average monthly downloads. The quantiles of the average monthly downloads in the sampled 824 software systems are: 0 (minimum), 0 (25%), 6 (median), 16 (75%), and 661,247 (maximum).

<sup>6</sup>gusi, <http://sourceforge.net/projects/gusi>

<sup>7</sup>pic-gcc-library, <http://sourceforge.net/projects/pic-gcc-library>

#### D. Data Cleanliness

1) *Lifespan (LS)*: The 25% quantile of the lifespan of the 824 software systems is 51 days. We suspect that software systems with lifespans less than the 25% quantile are never finished or are just used as prototypes. After manually checking such software systems, we find that most of them are incomplete with very few commits. We exclude such software systems from our study. The number of subject systems drops from 824 to 618.

2) *System Size (SS)*: We manually check the software systems with lines of code less than the 25% quantile (*i.e.*, 1,124) of the 824 software systems. We find that most of such software systems are incomplete (*e.g.*, `vcgen8`), or mainly written in other languages (*e.g.*, `jajax9` is written mainly in javascript). We exclude such software systems from our study. The number of subject systems drops from 618 to 506.

3) *Programming Language (PL)*: We filter out software systems that are not mainly written in C, C++, C#, Java, or Pascal. The number of subject systems drops from 506 to 478.

4) *Number of Downloads (ND)*: Some of the remaining 478 software systems have no downloads. It might be because that such software systems are still incomplete to be used, or they are absolutely useless software. We exclude software systems without downloads from our study. The number of subject systems drops from 478 to 390.

5) *Application Domain (AD)*: A software system might be categorized into several application domains. The combinations of multiple application domains are considered as different application domains from single application domains. The 75% quantile of the number of software systems of all single and combined application domains is seven. We exclude combined application domains which have less than seven software systems, and yield six combined application domains. The number of subject systems drops from 390 to 323.

The collection date of the application domain is not the same as the date of collecting source code. To verify whether the application domains of the 323 software systems remain the same as time elapses, we checked the latest application domain information directly downloaded from SourceForge in April 2013. We find that only three software systems (*i.e.*, `cdstatus`, `g3d-cpp`, and `satyr10`) have changed their application domains, indicating that application domains collected in June 2008 are adequate. The application domain of `g3d-cpp` is removed, and the application domains of `cdstatus` and `satyr` are changed to Audio/Video. We exclude the three software systems from our study. The number of subject systems drops from 323 to 320.

**Refine Factors.** The context factors like age, lifespan, system size, and the number of changes seem to be strongly related. Hence, we compute Spearman correlation among these context factors of the 320 software systems. As shown in Table I, system size strongly correlates to the number of changes. We choose to examine the number of changes only.

<sup>8</sup><http://sourceforge.net/projects/vcgen>

<sup>9</sup><http://sourceforge.net/projects/jajax>

<sup>10</sup><http://sourceforge.net/projects/ProjectName> (NOTE: the ProjectName needs to be substituted by the real project name, *e.g.*, `cdstatus`)

TABLE I: The Spearman correlations among four context factors: age, lifespan, system size, and the number of changes.

Context Factor	Lifespan	System Size	Number of Changes
Age	0.35	0.06	0.14
Lifespan		0.25	0.46
System Size			0.67

TABLE II: The number of software systems per group divided by each context factor.

Context Factor	Group	Number of Systems
Application Domain (AD)	$G_{build}$	31
	$G_{codegen}$	26
	$G_{comm}$	23
	$G_{frame}$	29
	$G_{games}$	49
	$G_{internet}$	19
	$G_{network}$	16
	$G_{sudev}$	41
	$G_{sysadmin}$	29
	$G_{build;codegen}$	14
	$G_{comm;internet}$	13
	$G_{comm;network}$	7
	$G_{games;internet}$	7
	$G_{internet;sudev}$	9
$G_{sudev;sysadmin}$	7	
Programming Language (PL)	$G_c$	57
	$G_{cpp}$	85
	$G_{c\#}$	18
	$G_{java}$	146
	$G_{pascal}$	14
Age (AG)	$G_{lowAG}$	80
	$G_{moderateAG}$	160
	$G_{highAG}$	80
Life Span (LS)	$G_{lowLS}$	80
	$G_{moderateLS}$	160
	$G_{highLS}$	80
Number of Changes (NC)	$G_{lowNC}$	80
	$G_{moderateNC}$	160
	$G_{highNC}$	80
Number of Downloads (ND)	$G_{lowND}$	90
	$G_{moderateND}$	150
	$G_{highND}$	80

**Summary.** When investigating the impact of application domain (respectively programming language) on the distribution of metric values, we break down the 320 software systems into 15 (respectively five) groups as shown in Table II. When investigating the impact of the other four context factors on the distribution of metric values, we divide the 320 software systems into three groups, respectively, in the following way: 1) low (below or at the 25% quantile); 2) moderate (above the 25% quantile and below or at the 75% quantile); and 3) high (above the 75% quantile). The 25% quantile of lifespan (respectively the number of changes and the number of monthly downloads) is: 287 days (respectively 364 and 6). The 75% quantile of lifespan (respectively the number of changes and the number of monthly downloads) is: 1,324 days (respectively 2,195 and 38). The detailed groups are shown in Table II.

### E. Metrics Computation

In this study, we select 39 metrics related to the five quality attributes (*i.e.*, modularity, reusability, analyzability, modifiability, and testability) of software maintainability (as defined in ISO/IEC 25010). We further group the 39 metrics into six categories (*i.e.*, complexity, coupling, cohesion, abstraction, encapsulation, and documentation) based on Zou and Kontogiannis[15]'s work. These categories can measure different aspects of software maintainability. For example, low complexity indicates high analyzability and modifiability; low coupling improves analyzability and reusability; high cohesion increases modularity and modifiability; high abstraction enhances reusability; high encapsulation implies high modularity; and documentation might contribute to analyzability, modifiability, and reusability. The metrics and their categories are shown in Table III. Most metrics can be computed by a commercial tool, called Understand<sup>11</sup>. For the remaining metrics, we computed them by ourselves<sup>12</sup> using equations from the work of Aggarwal *et al.* [26].

### F. Analysis Methods

For each factor, we divide all software systems into non-overlapping groups, as described in Section III-D.

**Analysis method for RQ1:** To study how context factors impact the distribution of metric values, we analyze the overall impact of each context factor. To examine the overall impact of a factor  $f$  on a metric  $m$ , we test the following null hypothesis for the grouping based on factor  $f$ .

$H_{01}$ : *there is no difference in the distributions of metric values among all groups.*

To compare the distribution of metric  $m$  values among all groups, we apply Kruskal Wallis test [32] using the 5% confident level (*i.e.*,  $p$ -value<0.05). The Kruskal Wallis test assesses whether two or more samples originate from the same distribution. It does not assume a normal distribution since it is a non-parametric statistical test. As we investigate six context factors and 39 metrics in total, we apply Bonferroni correction which adjusts the threshold  $p$ -value by dividing the number of tests ( $39 \times 6=234$  tests). If there is a statistically significant difference (*i.e.*,  $p$ -value is less than  $0.05/234=2.14e-04$ ), we reject the null hypothesis and report that factor  $f$  impacts the distribution of metric  $m$  values.

**Analysis method for RQ2:** To provide guidelines on how to group software systems for benchmarking maintainability metrics, we break down our analysis method into three steps: 1) for each impacting factor, we examine the impact in detail by comparing every pair of groups separated by the factor; 2) for any comparison exhibiting a statistically significant difference, we further compute the corresponding effect size to quantify the importance of the difference; and 3) we discuss how to use effect sizes to split software systems. We present the detailed steps as follows.

<sup>11</sup><http://www.scitools.com>

<sup>12</sup><https://bitbucket.org/serap/contextstudy>

TABLE III: List of metrics characterizing maintainability. The metrics are from three levels: project level (P), class level (C), and method level (M).

Category	Metric	Level
Complexity	Total Lines of Code (TLOC)	P
	Total Number of Files (TNF)	P
	Total Number of Classes (TNC)	P
	Total Number of Methods (TNM)	P
	Total Number of Statements (TNS)	P
	Class Lines of Code (CLOC)	C
	Number of local Methods (NOM) [27]	C
	Number of Instance Methods (NIM) [28]	C
	Number of Instance Variables (NIV) [28]	C
	Weighted Methods per Class (WMC) [29]	C
	Number of Method Parameters (NMP)	M
	McCabe Cyclomatic Complexity (CC) [2]	M
Number of Possible Paths (NPATH) [28]	M	
Max Nesting Level (MNL) [28]	M	
Coupling	Coupling Factor (CF) [30]	P
	Coupling Between Objects (CBO) [29]	C
	Information Flow Based Coupling (ICP) [5]	C
	Message Passing Coupling (MPC) [27]	C
	Response For a Class (RFC) [29]	C
	Number of Method Invocation (NMI)	M
Number of Input Data (FANIN) [28]	M	
Number of Output Data (FANOUT) [28]	M	
Cohesion	Lack of Cohesion in Methods (LCOM) [29]	C
	Tight Class Cohesion (TCC) [31]	C
	Loose Class Cohesion (LCC) [31]	C
	Information Flow Based Cohesion (ICH) [26]	C
Abstraction	Number of Abstract Classes/Interfaces (NACI)	P
	Method Inheritance Factor (MIF) [30]	P
	Number of Immediate Base Classes (IFANIN) [28]	C
	Number of Immediate Subclasses (NOC) [29]	C
Depth of Inheritance Tree (DIT) [29]	C	
Encapsulation	Ratio of Public Attributes (RPA)	C
	Ratio of Public Methods (RPM)	C
	Ratio of Static Attributes (RSA)	C
	Ratio of Static Methods (RSM)	C
Documentation	Comment of Lines per Class (CLC)	C
	Ratio Comments to Codes per Class (RCCC)	C
	Comment of Lines per Method (CLM)	M
	Ratio Comments to Codes per Method (RCCM)	M

### 1) Pairwise comparison of the distribution of metric values:

To investigate the effects of factor  $f$  on metric  $m$ , we test the following null hypothesis for every pair of groups divided by factor  $f$ .

$H_{02}$ : *there is no difference in the distributions of metric values between the two groups of any pairs.*

To examine the difference in the distribution of the metric  $m$  values between every two groups, we apply Mann-Whitney U test [32] using the 5% confident level (*i.e.*,  $p$ -value<0.05). The Mann-Whitney U test assesses whether two independent distributions have equally large values. It does not assume a normal distribution since it is a non-parametric statistical test. As we conduct multiple tests, we also apply Bonferroni correction to the threshold  $p$ -value. If there is a statistically significant difference, we reject the null hypothesis and claim that factor  $f$  is important to metric  $m$ . To quantify the importance, we further compute the effect size.

TABLE IV:  $p$ -value of Kruskal Wallis test. Non statistically significant is denoted as n.s., which means  $p$ -value is not less than  $2.14e-04$  (i.e.,  $0.05/39/6$ ).

Category	Metric	Application Domain (AD)	Programming Language (PL)	Age (AG)	Lifespan (LS)	Number of Changes (NC)	Number of Downloads (ND)
Complexity	TLOC	n.s.	n.s.	n.s.	1.94e-05	< 2.2e-16	n.s.
	TNF	n.s.	1.11e-05	n.s.	5.97e-06	< 2.2e-16	n.s.
	TNC	3.41e-05	< 2.2e-16	n.s.	n.s.	8.05e-12	n.s.
	TNM	1.46e-04	< 2.2e-16	n.s.	n.s.	1.03e-11	n.s.
	TNS	n.s.	n.s.	n.s.	6.26e-06	< 2.2e-16	n.s.
	CLOC	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.37e-14	n.s.	< 2.2e-16
	NOM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NIM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NIV	< 2.2e-16	< 2.2e-16	< 2.2e-16	5.27e-10	5.76e-08	5.27e-11
	WMC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NMP	< 2.2e-16	< 2.2e-16	< 2.2e-16	2.93e-07	< 2.2e-16	< 2.2e-16
	CC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NPATH	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	MNL	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	2.22e-12	< 2.2e-16
Coupling	CF	n.s.	n.s.	n.s.	9.55e-05	< 2.2e-16	n.s.
	CBO	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16
	ICP	< 2.2e-16	< 2.2e-16	8.34e-11	< 2.2e-16	< 2.2e-16	n.s.
	MPC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.50e-04
	RFC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NMI	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	FANIN	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	FANOUT	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
Cohesion	LCOM	< 2.2e-16	< 2.2e-16	< 2.2e-16	5.36e-10	n.s.	6.34e-15
	TCC	< 2.2e-16	< 2.2e-16	1.11e-14	< 2.2e-16	8.87e-14	< 2.2e-16
	LCC	< 2.2e-16	< 2.2e-16	8.09e-15	< 2.2e-16	5.68e-14	< 2.2e-16
	ICH	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16	5.60e-12	n.s.
Abstraction	NACI	6.30e-05	< 2.2e-16	n.s.	n.s.	5.78e-09	n.s.
	MIF	n.s.	< 2.2e-16	n.s.	n.s.	n.s.	n.s.
	IFANIN	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.75e-05	1.69e-04
	NOC	< 2.2e-16	< 2.2e-16	3.50e-08	8.20e-05	n.s.	1.95e-06
DIT	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.52e-14	n.s.	< 2.2e-16	
Encapsulation	RPA	< 2.2e-16	n.s.	n.s.	n.s.	n.s.	n.s.
	RPM	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.13e-06	< 2.2e-16	< 2.2e-16
	RSA	4.45e-16	3.26e-05	n.s.	3.38e-07	< 2.2e-16	4.85e-07
	RSM	< 2.2e-16	1.83e-08	1.41e-05	< 2.2e-16	n.s.	< 2.2e-16
Documentation	CLC	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.80e-15	1.51e-11	< 2.2e-16
	RCCC	< 2.2e-16	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16	< 2.2e-16
	CLM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	RCCM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

TABLE V: Mapping Cohen's  $d$  to Cliff's  $\delta$ .

Cohen's Standard	Cohen's $d$	Pct. of Non-overlap	Cliff's $\delta$
small	0.20	14.7%	0.147
medium	0.50	33.0%	0.330
large	0.80	47.4%	0.474

2) **Quantifying the importance of the difference:** We apply Cliff's  $\delta$  as effect size [33] to quantify the importance of the difference, since Cliff's  $\delta$  is reported [33] to be more robust and reliable than Cohen's  $d$  [34]. As Cliff's  $\delta$  estimates non-parametric effect sizes, it makes no assumptions of a particular distribution. Cliff's  $\delta$  represents the degree of overlap between two sample distributions [33]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [35].

3) **Interpreting the effect sizes:** Cohen's  $d$  is mapped to Cliff's  $\delta$  via the percentage of non-overlap as shown in Table V [33].

Cohen [36] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably larger than medium. In this study, we choose the large effect size as the threshold. If the effect size is large, we suggest that software systems should be split into different groups based on factor  $f$  when benchmarking metric  $m$ . Otherwise, all software systems can be put in the same group along with factor  $f$ .

#### IV. CASE STUDY RESULTS

In this section, we report and discuss the results of our study.

##### **RQ1: What context factors impact the distribution of the values of software maintainability metrics?**

**Motivations.** This question is preliminary to the other question. It determines the number of pairwise tests in the other question. In this research question, we determine if each factor impacts the distribution of each metric values, and should be considered in pairwise comparison.

**Approach.** To address this research question, we examine each factor individually. For each factor, we divide all software systems into non-overlapping groups as described in Section III-D. On all groups divided by a single factor, we test the null hypothesis  $H0_1$  from Section III-F using Kruskal Wallis test with the 5% confident level (*i.e.*,  $p\text{-value} < 2.14e-04$  after Bonferroni correction). If the difference is statistically significant, we reject the null hypothesis  $H0_1$  and state that the corresponding factor has an overall impact on the distribution of the values of the corresponding metric.

**Findings.** We present  $p$ -value of Kruskal Wallis test in Table IV. For each factor, statistically significant results indicate the impacting factors. In general, 51% of metrics (*i.e.*, 20 out of 39) are impacted by all six factors. On the other hand, programming language, application domain, and lifespan are three most important factors since they impact over 80% of metrics (*i.e.*, 35, 34, and 33 out of 39, respectively). Moreover, the number of changes, age, and the number of downloads affect more than 70% of metrics (*i.e.*, 31, 28 and 28 out of 39, respectively).

Overall, we conclude that all six factors impact the distribution of the maintainability metric values. The programming language is the most influential factor, since it affects 90% of metrics (*i.e.*, 35 out of 39). In the next research question, we examine types (of programming language, application domain) and levels (of lifespan, the number of changes, age, the number of downloads) in more detail to determine what factors should be considered when benchmarking software maintainability.

## RQ2: What guidelines can we provide for benchmarking software maintainability metrics?

**Motivations.** In RQ1, we found that each of the six factors impact the values of at least 75% of metrics. However, considering all six factors when benchmarking software maintainability can result in a number of small groups, and can increase the possibility of duplicated benchmarks. To effectively build benchmarks, we suggest to follow three steps: a) separate software systems into distinct groups to ensure each group contains only systems that share a similar context; b) apply existing approaches (*e.g.*, [14], [21]) to build benchmarks of each group; c) for a given software system, determine which groups it belongs to and apply corresponding benchmarks to evaluate its maintainability. The results of several benchmarks can be aggregated when evaluating the maintainability of software. In this research question, we aim to find the factors that impact the distribution of the maintainability metric values. Such factors can affect the derivation of the thresholds/ranges of the corresponding metrics. Moreover, we provide guidelines in splitting software systems into distinct groups for building benchmarks to measure software maintainability.

**Approach.** To address each research question, we divide all software systems into non-overlapping groups by each factor, respectively (as described in Section III-D). If examining all possible interactions of all six context factors, the number of groups will be 6,075 ( $= 15 \times 5 \times 3 \times 3 \times 3 \times 3$ ). However, the number of our subject systems is 320, then a large number

TABLE VI: List of the threshold  $p$ -values after Bonferroni correction. The number of pairwise tests required for each context factor is determined by  $C(n, 2)$ , which denotes the number of 2-combinations from a given set  $S$  of  $n$  elements. The number of groups by factors AD, PL, AG, LS, NC, and ND are: 15, 5, 3, 3, 3, and 3, respectively. Hence, the number of pairwise tests are:  $C(15, 2) = 105$ ,  $C(5, 2) = 10$ ,  $C(3, 2) = 3$ ,  $C(3, 2) = 3$ ,  $C(3, 2) = 3$ , and  $C(3, 2) = 3$ , respectively.

Metric	Number of Pairwise Tests by Each Factor						Total Number of Pairwise Tests	Corrected Threshold $p$ -value
	AD	PL	AG	LS	NC	ND		
TLOC	0	0	0	3	3	0	6	8.33e-03
TNF	0	10	0	3	3	0	16	3.13e-03
TNC	105	10	0	0	3	0	118	4.24e-04
TNM	105	10	0	0	3	0	118	4.24e-04
TNS	0	0	0	3	3	0	6	8.33e-03
CLOC	105	10	3	3	0	3	124	4.03e-04
NOM	105	10	3	3	3	3	127	3.94e-04
NIM	105	10	3	3	3	3	127	3.94e-04
NIV	105	10	3	3	3	3	127	3.94e-04
WMC	105	10	3	3	3	3	127	3.94e-04
NMP	105	10	3	3	3	3	127	3.94e-04
CC	105	10	3	3	3	3	127	3.94e-04
NPATH	105	10	3	3	3	3	127	3.94e-04
MNL	105	10	3	3	3	3	127	3.94e-04
CF	0	0	0	3	3	0	6	8.33e-03
CBO	105	10	3	3	0	3	124	4.03e-04
ICP	105	10	3	3	3	0	124	4.03e-04
MPC	105	10	3	3	3	3	127	3.94e-04
RFC	105	10	3	3	3	3	127	3.94e-04
NMI	105	10	3	3	3	3	127	3.94e-04
FANIN	105	10	3	3	3	3	127	3.94e-04
FANOUT	105	10	3	3	3	3	127	3.94e-04
LCOM	105	10	3	3	0	3	124	4.03e-04
TCC	105	10	3	3	3	3	127	3.94e-04
LCC	105	10	3	3	3	3	127	3.94e-04
ICH	105	10	0	3	3	0	121	4.13e-04
NACI	105	10	0	0	3	0	118	4.24e-04
MIF	0	10	0	0	0	0	10	5.00e-03
IFANIN	105	10	3	3	3	3	127	3.94e-04
NOC	105	10	3	3	0	3	124	4.03e-04
DIT	105	10	3	3	0	3	124	4.03e-04
RPA	105	0	0	0	0	0	105	4.76e-04
RPM	105	10	3	3	3	3	127	3.94e-04
RSA	105	10	0	3	3	3	124	4.03e-04
RSM	105	10	3	3	0	3	124	4.03e-04
CLC	105	10	3	3	3	3	127	3.94e-04
RCCC	105	10	3	0	3	3	124	4.03e-04
CLM	105	10	3	3	3	3	127	3.94e-04
RCCM	105	10	3	3	3	3	127	3.94e-04

of groups might be empty. Therefore, interactions of all six context factors are not investigated in this study.

For each pair of groups divided by a factor  $f$ , we test the null hypothesis  $H0_2$  as discussed in Section III-F using Mann-Whitney U test with the 5% confident level. We apply Bonferroni correction to adjust the threshold  $p$ -value based on the findings of RQ1. The corrected threshold  $p$ -values are shown in Table VI. If the difference is statistically significant, we reject the null hypothesis  $H0_2$  and further compute the Cliff's  $\delta$  effect size to determine the importance of the factor.

If the Cliff's  $\delta$  effect size is large, we conclude that the

TABLE VII: Cliff's  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *complexity* metrics).

Metric	Factor	Group1	Group2	Cliff's $\delta$
TLOC	NC	$G_{lowNC}$	$G_{highNC}$	0.498
TNF	NC	$G_{lowNC}$	$G_{moderateNC}$	0.573
		$G_{moderateNC}$	$G_{highNC}$	0.639
TNC	AD	$G_{frame}$	$G_{network}$	-0.519
		$G_{frame}$	$G_{comm;network}$	-0.759
	PL	$G_c$	$G_{c\#}$	0.596
		$G_c$	$G_{java}$	0.667
		$G_{pascal}$	$G_{cpp}$	0.560
	NC	$G_c$	$G_{c\#}$	0.729
$G_{pascal}$		$G_{java}$	0.885	
TNM	AD	$G_{lowNC}$	$G_{moderateNC}$	0.476
		$G_{lowNC}$	$G_{highNC}$	0.552
	PL	$G_{frame}$	$G_{network}$	-0.599
		$G_c$	$G_{c\#}$	0.614
		$G_c$	$G_{java}$	0.591
	NC	$G_{cpp}$	$G_{c\#}$	0.683
$G_{cpp}$		$G_{java}$	0.758	
TNS	NC	$G_{lowNC}$	$G_{highNC}$	0.541

TABLE VIII: Cliff's  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *coupling* metrics).

Metric	Factor	Group1	Group2	Cliff's $\delta$	
CBO	AD	$G_{comm;network}$	$G_{build;codegen}$	0.482	
	PL	$G_{pascal}$	$G_{c\#}$	0.486	
RFC	AD	$G_{comm;network}$	$G_{network}$	0.524	
			$G_{internet}$	0.578	
			$G_{sysadmin}$	0.492	
			$G_{codegen}$	0.764	
			$G_{frame}$	0.620	
			$G_{build}$	0.703	
			$G_{swdev}$	0.847	
			$G_{games;internet}$	0.643	
			$G_{internet;swdev}$	0.647	
			$G_{comm;internet}$	0.642	
			$G_{build;codegen}$	$G_{comm;network}$	-0.923
			$G_{build;codegen}$	$G_{swdev;sysadmin}$	-0.531
			PL	$G_{c\#}$	$G_{java}$
	CF	NC	$G_{lowNC}$	$G_{highNC}$	-0.554
NMI	PL	$G_{java}$	$G_{c\#}$	-0.516	
		$G_{java}$	$G_{pascal}$	-0.516	

corresponding factor  $f$  has a large impact on the distribution of the corresponding metric  $m$ . Hence, factor  $f$  should be considered when benchmarking metric  $m$ .

**Findings.** To better understand the impact of context factors on different aspects of software maintainability, we report our findings along the six categories of metrics: complexity, coupling, cohesion, abstraction, encapsulation and documentation.

### 1) Complexity.

As shown in Table VII, the factor impacting TLOC, TNF, and TNS is the number of changes. The factors impacting

TABLE IX: Cliff's  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *cohesion* metrics).

Metric	Factor	Group1	Group2	Cliff's $\delta$
LCOM	AD	$G_{network}$	$G_{comm;network}$	0.552

TABLE X: Cliff's  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *abstraction* metrics).

Metric	Factor	Group1	Group2	Cliff's $\delta$
NACI	PL	$G_{java}$	$G_{pascal}$	-0.773
IFANIN	AD	$G_{comm;network}$	$G_{network}$	0.794
			$G_{internet}$	0.751
			$G_{sysadmin}$	0.657
			$G_{comm}$	0.776
			$G_{codegen}$	0.780
			$G_{frame}$	0.748
			$G_{build}$	0.738
			$G_{swdev}$	0.736
			$G_{games}$	0.598
			$G_{games;internet}$	0.620
			$G_{swdev;sysadmin}$	0.828
			$G_{internet;swdev}$	0.704
			$G_{comm;internet}$	0.745
	$G_{build;codegen}$	0.824		
		$G_{games}$	$G_{swdev;sysadmin}$	0.486
PL	$G_{java}$	$G_{cpp}$	$G_{pascal}$	-0.514
		$G_{cpp}$	$G_{pascal}$	-0.708
DIT	AD	$G_{comm;network}$	$G_{network}$	0.820
			$G_{internet}$	0.861
			$G_{sysadmin}$	0.772
			$G_{comm}$	0.907
			$G_{codegen}$	0.954
			$G_{frame}$	0.899
			$G_{build}$	0.870
			$G_{swdev}$	0.962
			$G_{games}$	0.839
			$G_{games;internet}$	0.746
			$G_{swdev;sysadmin}$	0.910
			$G_{internet;swdev}$	0.915
			$G_{comm;internet}$	0.910
			$G_{sysadmin}$	$G_{comm;network}$
		$G_{build;codegen}$	$G_{comm;network}$	-0.983
		$G_{build;codegen}$	$G_{games;internet}$	-0.517
MIF	PL	$G_{java}$	$G_{cpp}$	-0.777
			$G_{c\#}$	-0.849
			$G_{pascal}$	-0.666
		$G_{cpp}$	$G_{c\#}$	0.657

TNC and TNM are application domain, programming language, and the number of changes. Overall, the distributions of metric values in the *complexity* category are strongly impacted by three context factors: application domain, programming language, and the number of changes.

### 2) Coupling.

As shown in Table VIII, the factors impacting CBO and RFC are application domain and programming language. The factor impacting CF (respectively NMI) is the number of changes (respectively programming language). Overall, the distributions of metric values in the *coupling* category are strongly impacted by three context factors: application domain, programming language, and the number of changes.

TABLE XI: Cliff’s  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *encapsulation* metrics).

Metric	Factor	Group1	Group2	Cliff’s $\delta$
RPA	AD	$G_{build}$	$G_{internet}$	0.483
			$G_{codegen}$	0.558
			$G_{frame}$	0.619
			$G_{swdev}$	0.576
			$G_{games}$	0.682
			$G_{swdev;sysadmin}$	0.543
			$G_{internet;swdev}$	0.550
			$G_{comm;internet}$	0.505
			$G_{build;codegen}$	0.527
RSM	AD	$G_{comm;network}$	$G_{comm;internet}$	0.710

TABLE XII: Cliff’s  $\delta$  and  $p$ -value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *documentation* metrics).

Metric	Factor	Group1	Group2	Cliff’s $\delta$
RCCC	AD	$G_{build;codegen}$	$G_{network}$	-0.513
	PL	$G_{java}$	$G_{pascal}$	-0.611

### 3) Cohesion.

As shown in Table IX, the factor impacting LCOM is application domain. Overall, the distributions of metric values in the *cohesion* category are strongly impacted by application domain only.

### 4) Abstraction.

As shown in Table X, the factor impacting NACI and MIF is programming language. The factor impacting DIT is application domain. The factors impacting IFANIN are application domain and programming language. Overall, the distributions of metric values in the *abstraction* category are strongly impacted by application domain and programming language.

### 5) Encapsulation.

As shown in Table XI, the factor impacting RPA and RSM is application domain. Overall, the distributions of metric values in the *encapsulation* category are strongly impacted by application domain only.

### 6) Documentation.

As shown in Table XII, the factors impacting RCCC are application domain and programming language. Overall, the distributions of metric values in the *documentation* category are strongly impacted by application domain and programming language.

### Guidelines for Benchmarking Maintainability Metrics.

Based on our findings, application domain, programming language, and the number of changes strongly impact the distribution of maintainability metric values.

When benchmarking the 39 metrics, we suggest to partition software systems into 13 groups: 1) five groups along application domain (i.e.,  $G_{build}$ ,  $G_{games}$ ,  $G_{frame}$ ,  $G_{build;codegen}$ , and  $G_{comm;network}$ ); 2) five groups along programming language (i.e.,  $G_c$ ,  $G_{cpp}$ ,  $G_{c\#}$ ,  $G_{java}$ , and  $G_{pascal}$ ); and 3) three groups along the number of changes (i.e.,  $G_{lowNC}$ ,  $G_{moderateNC}$ , and  $G_{highNC}$ ). When benchmarking metrics from a particular

TABLE XIII: Guidelines on partitioning software systems when building metric based benchmarks.

Metric Category	Factor	Group
Complexity	AD	$G_{frame}$ and others
	PL	$G_c$ , $G_{pascal}$ and others
	NC	$G_{lowNC}$ , $G_{moderateNC}$ , and $G_{highNC}$
Coupling	AD	$G_{comm;network}$ , $G_{build;codegen}$ , and others
	PL	$G_{pascal}$ , $G_{java}$ , and others
	NC	$G_{lowNC}$ , $G_{moderateNC}$ , and $G_{highNC}$
Cohesion	AD	$G_{comm;network}$ , and others
Abstraction	AD	$G_{comm;network}$ , $G_{games}$ , $G_{build;codegen}$ , and others
	PL	$G_{java}$ , $G_{cpp}$ , and others
Encapsulation	AD	$G_{build}$ , $G_{comm;network}$ , and others
Documentation	AD	$G_{build;codegen}$ , and others
	PL	$G_{java}$ , and others

category, we provide detailed suggestions in Table XIII. Moreover, our approach can be applied to other software metrics and other software systems for generating guidelines on building benchmarks of such software metrics.

## V. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines provided in [37].

**Threats to conclusion validity** concern the relation between the treatment and the outcome. Our conclusion validity threats are mainly due to sampling errors. Since stratified sampling was performed only along application domain, sampled software systems may not well represent along other five factors. Some differences along these factors, thus, may not be detected, and the detected differences are likely to be only a subset of differences. We plan to stratify along other factors.

**Threats to internal validity** concern our selection of subject systems and analysis methods. We randomly sample 320 software systems from SourceForge, some of the findings might be specific to software systems hosted on SourceForge. Future studies should consider using software systems from other hosts, and even commercial software systems.

**Threats to external validity** concern the possibility to generalize our results. Some of the findings might not be directly applicable to different software systems. Yet our approach can be applied to find guidelines for benchmarking maintainability of different open source and commercial software systems.

**Threats to reliability validity** concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. SourceForge is publicly available to obtain the same data. We make our data and R script available<sup>13</sup> as well.

## VI. CONCLUSION

In this work, we perform a large scale empirical study to investigate how the six context factors affect the distribution

<sup>13</sup><https://bitbucket.org/serap/contextstudy>

of maintainability metric values. We apply statistical methods (*i.e.*, Kruskal Wallis test, Mann-Whitney U test and Cliff's  $\delta$  effect size) to analyze 320 software systems, and provide empirical evidence of the impact of context factors on the distribution of maintainability metric values. Our results show that all six context factors impact the distribution of the values of 51% of metrics. The most influential factors are application domain, programming language, and the number of changes. Based on our findings, we further provide guidelines on how to group software systems according to the six context factors. We expect our findings to help software benchmarking and other software engineering methods using the 39 software maintainability metrics.

In the future, we plan to extend our study using more software systems from SourceForge, GoogleCode, and GitHub, and to perform stratified sampling along all six context factors. Moreover, we want to derive the thresholds and ranges of metric values based on our findings and provide benchmarks to measure maintainability. We also want to verify whether our findings can help sample representative software systems for empirical studies.

#### REFERENCES

- [1] N. E. Fenton and M. Neil, "Software metrics: roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 357–370.
- [2] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering (TSE)*, vol. SE-2, no. 4, pp. 308 – 320, dec. 1976.
- [3] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.
- [4] K. Erni and C. Lewerentz, "Applying design-metrics to object-oriented frameworks," in *Proceedings of the 3rd International Software Metrics Symposium (METRICS'96)*, mar 1996, pp. 64 –74.
- [5] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. New York, NY, USA: ACM, 1999, pp. 345–354.
- [6] L. B. L. De Souza and M. D. A. Maia, "Do software categories impact coupling metrics?" in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 217–220.
- [7] T. Dybå, D. I. Sjøberg, and D. S. Cruzes, "What works for whom, where, when, and why?: on the role of context in empirical software engineering," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12)*. New York, NY, USA: ACM, 2012, pp. 19–28.
- [8] S. Benlarbi, K. El Emam, N. Goel, and S. Rai, "Thresholds for object-oriented measures," in *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, 2000, pp. 24 –38.
- [9] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 2, pp. 216 –225, mar 2010.
- [10] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae, "An approach to outlier detection of software measurement data using the k-means clustering method," in *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*, sept. 2007, pp. 443 –445.
- [11] T. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, Sept. 2010, pp. 1 –10.
- [12] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Journal of Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, Dec. 2011.
- [13] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy, "A probabilistic software quality model," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, 2011, pp. 243–252.
- [14] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, Feb. 2012.
- [15] Y. Zou and K. Kontogiannis, "Migration to object oriented platforms: a state transformation approach," in *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, 2002, pp. 530 – 539.
- [16] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: a challenge to software engineering," in *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 74–83.
- [17] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.
- [18] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, pp. 2:1–2:26, Oct. 2008.
- [19] I. Herraiz, D. M. Germán, and A. E. Hassan, "On the distribution of source code file sizes," in *Proceedings of the 6th International Conference on Software and Data Technologies (ICSOFIT'11)*, 2011, pp. 5–14.
- [20] L. Sánchez-González, F. García, F. Ruiz, and J. Mendling, "A study of the effectiveness of two threshold definition techniques," in *EASE*, 2012, pp. 197–205.
- [21] R. Baggen, J. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, pp. 287–307, 2012.
- [22] A. Capiluppi, P. Lago, and M. Morisio, "Characteristics of open source projects," in *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, Mar. 2003, pp. 317 – 327.
- [23] K. Johari and A. Kaur, "Effect of software evolution on software metrics: an open source case study," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 1–8, Sep. 2011.
- [24] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR'09, may 2009, pp. 11 –20.
- [25] J. Howison, M. Conklin, and K. Crowston, "Flossmole: A collaborative repository for floss research data and analyses," *International Journal of Information Technology and Web Engineering*, vol. 1, pp. 17–26, 07/2006 2006.
- [26] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical study of object-oriented metrics," *Journal of Object Technology*, vol. 5, no. 8, pp. 149–173, 2006.
- [27] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, ser. Prentice-Hall Object-Oriented Series. Prentice Hall, 1996.
- [28] Scitools, "Metrics computed by understand," <http://www.scitools.com/documents/metricsList.php>.
- [29] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476 –493, jun 1994.
- [30] R. Harrison, S. Counsell, and R. Nithi, "An evaluation of the mood set of object-oriented software metrics," *IEEE Transactions on Software Engineering (TSE)*, vol. 24, no. 6, pp. 491 –496, jun 1998.
- [31] L. C. Briand, J. W. Daly, and J. K. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 25, no. 1, pp. 91–121, Jan. 1999.
- [32] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.
- [33] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys?" in *Annual Meeting of the Florida Association of Institutional Research*, February 2006, pp. 1–33.
- [34] J. Cohen, *Statistical power analysis for the behavioral sciences : Jacob Cohen.*, 2nd ed. Lawrence Erlbaum, Jan. 1988.
- [35] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, Nov. 1993.
- [36] J. Cohen, "A power primer," *Psychological Bulletin*, vol. 112, no. 1, pp. 155–159, 1992.
- [37] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

# Refactoring Clones: An Optimization Problem

Giri Panamoottil Krishnan, Nikolaos Tsantalis

Department of Computer Science and Software Engineering

Concordia University, Montreal, Quebec, Canada

giri.krishnan@concordia.ca, nikolaos.tsantalis@concordia.ca

**Abstract**—The refactoring of software clones is achieved by extracting their common functionality into a single method. Any differences in identifiers and literals between the clones have to become parameters in the extracted method. Obviously, a large number of differences leads to an extracted method with limited reusability due to the large number of introduced parameters. We support that minimizing the differences between the matched statements of clones is crucial for the purpose of refactoring and propose an algorithm that treats the matching process as an optimization problem.

## I. INTRODUCTION

The problem of source code matching or differencing has been investigated within the context of various applications, including change evolution analysis, plagiarism detection, code reuse, aspect mining, clone detection and refactoring. However, current approaches either do not explore the entire search space of possible matches, and thus may return non-optimal solutions, or face scalability issues due to the problem of combinatorial explosion. To facilitate the refactoring of duplicated code, an optimal solution should not only contain the maximum number of possible mapped statements, but also the minimum number of differences between them. To this end, we propose an algorithm to tackle both optimality and scalability issues. The contributions of the paper are: **(a)** we support that the problem of finding a mapping between the statements of two clones is an optimization problem with two objectives, namely maximizing the number of mapped statements and at the same time minimizing the number of differences between the mapped statements, **(b)** we express this optimization problem as finding the *Maximum Common Subgraph* (MCS) with the minimum number of differences in the *Program Dependence Graphs* (PDGs) of the clones, and **(c)** we apply a bottom-up mapping process based on the control dependence structure of the PDGs. This divide-and-conquer approach breaks the initial mapping problem into smaller sub-problems and avoids the risk of combinatorial explosion that might occur in the initial problem.

## II. RELATED WORK

Komondoor and Horwitz [1] apply slicing on PDGs to find isomorphic subgraphs that represent code clones. The advantage of this approach is the detection of non-contiguous clones (i.e., clones with gaps), clones with re-ordered statements, and clones intertwined with each other. Two nodes are matched if the corresponding statements are syntactically identical (i.e., their AST representation has the same structure) allowing only for differences in variable names and literal values. Krinke [2] proposed an approach to identify code clones by finding the maximal similar subgraphs in two PDGs by induction from

a pair of starting vertices. To reduce the complexity of the algorithm, he considers only a subset of vertices (i.e., predicate vertices) as starting points, and restricts the maximum length of the explored paths using a  $k$ -limit. One important limitation is that the running time of the algorithm explodes as  $k$ -limit increases. Another limitation is that the use of  $k$ -limit may lead to an incomplete solution (i.e., the selected  $k$ -limit is insufficient for detecting all possible matching vertices). Shepherd et al. [3] implemented an automated aspect mining technique exploiting the PDG and AST representations of a program. The proposed algorithm, inspired by [2] and [1], starts by matching the control dependence subgraphs of two compared PDGs to extract all possible matching solutions. Next, it filters out the undesirable matching solutions based on data dependence information. A limitation is that the algorithm always starts from the method entry nodes, and thus will fail to match control dependence subgraphs nested in different levels. More recently, Higo and Kusumoto [4] improved Komondoor's technique [1] by extending the PDG representation and introducing some heuristics to enhance code clone detection. The common limitation of all aforementioned techniques is that they do not explore the entire search space of possible solutions and therefore may return a non-optimal solution. In contrast to MCS approach that builds a search tree examining all possible combinations in the case of multiple node matches, the aforementioned techniques always select one match for each node, essentially exploring only a single path of the entire search tree.

Balazinska et al. [5] focus on the extraction of differences between cloned methods and their contextual dependencies as a means to help the developer make a decision on the actual refactoring to be performed. The comparison of the cloned methods is based on the Dynamic Pattern Matching algorithm, which is applied on the sequences of tokens forming the code fragment being compared and finds an optimal distance between them (i.e., the minimum amount of tokens that have to be inserted, deleted, or substituted to transform one code fragment into the other). A limitation of the algorithm is that it does not take into account the control dependence structure of the cloned code fragments during the token alignment process. Liu et al. [6] developed a software plagiarism detection tool called GPLAG. They support that the PDG structures of the original and the plagiarized code remain invariant and exploit this property to find plagiarism through relaxed subgraph isomorphism testing i.e., by checking if a PDG is  $\gamma$ -isomorphic to another, where  $\gamma$  is a relaxation parameter. To increase the efficiency of the algorithm, they prune the search space (i.e., reduce the number of PDG pairs to be checked) by applying some filters. Fluri et al. [7] describe an approach to extract the fine-grained changes that occur across different versions

of a program. Their method is based on the tree alignment algorithm proposed by Chawathe et al. [8], which takes as input two trees and produces a minimum edit script that can transform one tree into the other. A limitation of the proposed approach is that string-based similarity matching is not resilient to extensive renaming of identifiers. In addition, the best match approach applied for leaf level nodes may match reoccurring statements that are not at the same position in the method body. Cottrell et al. [9] present an approach to help developers integrate reusable source code. Their algorithm takes as input two ASTs and tries to produce the best correspondences between the nodes. A limitation is that the approach is semi-automated, since user intervention is required to resolve the conflicts when multiple matches are found. Additionally, it tries to find a best fit in a greedy way, which may lead to a non-optimal solution for the entire problem.

### III. MOTIVATING EXAMPLE

In this section, we will present an example that motivated our research and at the same time demonstrates the limitations of previous approaches. Figure 1 illustrates two code fragments taken from methods `drawDomainMarker` and `drawRangeMarker`, respectively, found in class `AbstractXYItemRenderer` of the `JFreeChart` open-source project (version 1.0.14). These two methods contain over 90 duplicated statements extending through their entire body. However, for the sake of simplicity, we have included only a small portion of the duplicated code. Figure 1 depicts a possible mapping of the statements as obtained from the PDG-based clone detection approaches discussed in section II. These techniques always select one match in the case of multiple possible node matches (e.g., statement 67 on the left side can be mapped to statements 68, 71, 80, and 83 on the right side), which, in the solution of Figure 1, coincides with the ‘first’ match according to the actual order of the statements. As it can be observed from Figure 1, the solution is maximum, since all 25 statements have been successfully mapped; however, it contains a large number of differences between the mapped statements. The minimization of the differences is of key importance for the refactoring of clones, since it directly affects the number of parameters that have to be introduced in the extracted method containing the common functionality, as well as the feasibility of the refactoring transformation. Figure 2 depicts the optimal mapping solution, which is again maximum in terms of the number of mapped statements, but it has also the minimum number of differences between the mapped statements. Clearly, the bodies of the `if/else if` statements in the left and right side of Figure 2 are ‘symmetrical’ to each other. Consequently, parameterizing the differences in the conditional expressions of the ‘symmetrical’ `if/else if` statements makes easier the refactoring of the clones and introduces less parameters to the extracted method.

### IV. PROPOSED SOLUTION

#### A. Maximum Common Subgraph algorithm

The detection of the *Maximum Common Subgraph* (MCS) is a well known NP-complete problem for which several optimal and suboptimal algorithms have been proposed in the literature. Conte et al. [10] compared the performance of the three most representative optimal algorithms, which are

```

60 if (im.getOutlinePaint() != null &&
   im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
70     }
71     if (range.contains(end)) {
72       line.setLine(end2d, y0, end2d, y1);
73       g2.draw(line);
74     }
75   } else if (orientation == HORIZONTAL) {
76     Line2D line = new Line2D.Double();
77     double x0 = dataArea.getMinX();
78     double x1 = dataArea.getMaxX();
79     g2.setPaint(im.getOutlinePaint());
80     g2.setStroke(im.getOutlineStroke());
81     if (range.contains(start)) {
82       line.setLine(x0, start2d, x1, start2d);
83       g2.draw(line);
84     }
85     if (range.contains(end)) {
86       line.setLine(x0, end2d, x1, end2d);
87       g2.draw(line);
88     }
89   }
90 }

```

Fig. 1. Non-optimal solution with 25 mapped nodes and 24 differences.

```

60 if (im.getOutlinePaint() != null &&
   im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
70     }
71     if (range.contains(end)) {
72       line.setLine(end2d, y0, end2d, y1);
73       g2.draw(line);
74     }
75   } else if (orientation == HORIZONTAL) {
76     Line2D line = new Line2D.Double();
77     double x0 = dataArea.getMinX();
78     double x1 = dataArea.getMaxX();
79     g2.setPaint(im.getOutlinePaint());
80     g2.setStroke(im.getOutlineStroke());
81     if (range.contains(start)) {
82       line.setLine(x0, start2d, x1, start2d);
83       g2.draw(line);
84     }
85     if (range.contains(end)) {
86       line.setLine(x0, end2d, x1, end2d);
87       g2.draw(line);
88     }
89   }
90 }

```

Fig. 2. Optimal solution with 25 mapped nodes and 2 differences.

based on depth-first tree search. All three algorithms have an exponential (more precisely, factorial) worst case time complexity with respect to the number of nodes in the graphs, in the order of  $\frac{(N_2+1)!}{(N_2-N_1+1)!}$ , where  $N_1$  and  $N_2$  is the number of nodes in graphs  $G_1$  and  $G_2$ , respectively [10]. The differences among the three algorithms actually lie only in the information used to represent each state of the search space, and in the kind of the heuristic adopted for pruning search paths [10]. We have adopted the McGregor algorithm [11] because it is simpler to implement and has a lower space complexity, in the order of  $O(N_1)$ , since only the states associated to the nodes of the currently explored path need to be stored in memory. The other two algorithms require the construction of the association graph between the two given graphs, which in the worst case can be a complete graph with a space complexity in the order of  $O(N_1 \cdot N_2)$ . Algorithm 1 is an adaptation of the McGregor algorithm to the particular characteristics of the PDGs. More specifically, given two PDGs, namely  $PDG_i$  and  $PDG_j$ , Algorithm 1 enforces the following constraints:

- 1) An edge of  $PDG_i$  is traversed only once in each path of the search tree (line 5).
- 2) A node from  $PDG_i$  is mapped to only one node from  $PDG_j$  (and vice versa) in each path of the search tree (lines 13 and 14).

```

1 Function search(pState, nodeMapping)
   Data: pState represents a parent state in the tree
   nodeMapping represents a pair of PDG nodes
   ( $node_i, node_j$ ) that have been already mapped
   Result: Builds recursively a search tree.
   The leaf nodes in the deepest level are states
   corresponding to maximum common subgraphs
   /* get incoming & outgoing edges */
2 Edgesi ← nodei.inEdges ∪ nodei.outEdges
3 Edgesj ← nodej.inEdges ∪ nodej.outEdges
4 foreach edgei ∈ Edgesi do
5   if edgei ∉ pState.visitedEdges then
6     add edgei → pState.visitedEdges
7     foreach edgej ∈ Edgesj do
8       if compatibleEdges(edgei, edgej) then
9         vNi ← edgei.otherEndPoint
10        vNj ← edgej.otherEndPoint
11        if compatibleAST(vNi, vNj) and
12        mappedCtrlParents(vNi, vNj) and
13        not alreadyMapped(vNi) and
14        not alreadyMapped(vNj) then
15          mapping ← (vNi, vNj)
16          state ← createState(mapping)
17          add state → pState.children
18          search(state, mapping)
19        end if
20      end if
21    end foreach
22  end if
23 end foreach
24 end

```

**Algorithm 1:** Recursive function building a search tree.

- 3) The control dependence structure of  $PDG_i$  and  $PDG_j$  is preserved throughout the mapping process. This means that if two control predicate nodes  $cp_i$  and  $cp_j$  have been mapped at a given stage of the search process, then a node nested under  $cp_i$  can only be mapped to nodes nested under  $cp_j$  (and vice versa) at later stages of the search process (line 12).

Algorithm 1 builds recursively a search tree by visiting the pairs of mapped PDG nodes in depth-first order. Each node in the search tree is created when a new pair of PDG nodes is mapped and represents a state of the search space. Each state keeps track of all visited edges and mapped PDG nodes in its path starting from the root state (function `createState` copies the visited edges and mapped nodes from the parent state to the child state). The leaf states in the deepest level of the search tree correspond to the maximum common subgraphs.

### B. PDG Node and Edge Compatibility

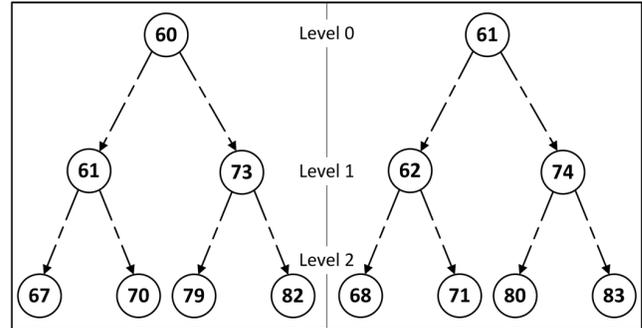
Two PDG nodes are considered compatible if they have a compatible AST structure. AST structural compatibility requires an identical AST tree structure and allows only for the replacement of expressions that return values of a given type (e.g., method call, field/variable/array access, class instance creation, array creation, literal, infix expressions) with other expressions (in the aforementioned list) as long as they return the same type or types being subclasses of a common superclass (excluding `Object`). In the case of control predicate

nodes (e.g., `if`, `for` statements), the part of the AST structure being compared is only their conditional expression.

Two PDG edges are considered compatible if they connect nodes which are compatible (i.e., the nodes in the starting and ending points of the edges, respectively, should be compatible with each other) and they have the same dependence type (i.e., they are both control or data flow dependences). In the case of control dependences, both should have the same control attribute (i.e., `True` or `False`). In the case of data dependences, the data attributes should correspond to variables having the same name, or to variables detected as renamed during the AST compatibility check of the connected nodes. Finally, if both data dependences are *loop-carried*, then the loop nodes through which they are carried should be compatible too.

### C. Divide-and-Conquer Based on Control Structure

Despite the constraints that we have set for our MCS search algorithm, it is still subject to the combinatorial explosion effect. As the number of possible matches for the nodes increases, the width of the search tree grows rapidly as a result of the numerous combinatorial considerations to be explored. In order to reduce the risk of combinatorial explosion, we decided to take advantage of the control dependence structure of the two compared PDGs. More specifically, we first build the *Control Dependence Tree* (CDT) of each PDG. The CDT has exactly the same structure with the *Control Dependence Graph* (CDG) with the only difference being that it includes only the control predicate nodes of the PDG. Figure 3 shows the CDTs for the duplicated code fragments of Figure 1. In this particular example, the CDTs are isomorphic. In the general case, we have to find the largest common bottom-up subtrees [12] in the CDTs, since only complete AST-subtrees having the same structure can be valid candidates for refactoring.



**Fig. 3.** The Control Dependence Trees for the code fragments of Figure 1.

Assuming that we have two isomorphic CDTs, namely  $CDT_i$  and  $CDT_j$  we can apply Algorithm 2 as a divide-and-conquer approach to the problem of PDG mapping. Starting from the deepest level of the CDTs, at each level the algorithm uses all possible pairwise combinations of the control predicate nodes nested at that level as starting points for Algorithm 1. Assuming that node  $cp_i$  of  $CDT_i$  is examined in the current level, a maximum common subgraph is generated and added in  $mcsStates$  for each starting point that  $cp_i$  participates in. After the examination of all possible matching combinations for node  $cp_i$ , the best solution in  $mcsStates$  (i.e., the solution with the maximum number of mapped nodes and the minimum number of differences between them) is appended to the final

solution. In level 2 of the CDTs (Figure 3), node 67 on the left side can be mapped to nodes 68, 71, 80, and 83 on the right side. Consequently, there are four possible matching nodes for node 67 and four node pairs to be used as starting points. All maximum common subgraphs resulting from the aforementioned starting points have the same number of mapped nodes, but only the subgraph generated from starting point (67, 80) has the minimum number of differences (equal to zero).

```

1 Function PDGMapping(ctrlDepTreei, ctrlDepTreej)
   Data: Two isomorphic CDTs
   Result: The final mapping solution as finalSolution
2   leveli ← ctrlDepTreei.maxLevel
3   levelj ← ctrlDepTreej.maxLevel
   /* an initially empty solution */
4   finalSolution ← ∅
5   while leveli ≥ 0 and levelj ≥ 0 do
6     cpNodesi ← nodes at leveli of ctrlDepTreei
7     cpNodesj ← nodes at levelj of ctrlDepTreej
8     foreach cpi ∈ cpNodesi do
9       mcsStates ← ∅
10      foreach cpj ∈ cpNodesj do
11        if compatibleAST(cpi, cpj) then
12          mapping ← (cpi, cpj)
13          root ← createState(mapping)
14          search(root, mapping)
15          get the maximum common subgraph
           from root & add it to mcsStates
16        end if
17      end foreach
18      select the best state from mcsStates &
       append it to finalSolution
19    end foreach
20    decrement leveli
21    decrement levelj
22  end while
23 end

```

**Algorithm 2:** A divide-and-conquer PDG mapping process based on control dependence structure.

## V. EVALUATION

To evaluate the efficiency and scalability of our algorithm we computed the number of distinct node comparisons required for the optimal mapping of the largest method-level clones detected by ConQAT in 4 open-source systems, namely *JFreeChart-1.0.14*, *Ant-1.9*, *JMeter-2.9*, and *JRuby-1.7.3*. As it can be observed from Table I, the number of node comparisons performed by our algorithm is always significantly smaller than the maximum number of possible node comparisons (equal to  $N_1 \times N_2$ , where  $N_1$  and  $N_2$  is the number of nodes in each PDG) indicating that our approach is more sophisticated and efficient compared to exhaustive search approaches.

## VI. FUTURE WORK

Our research vision is to improve the state-of-the-art in the refactoring of clones by discovering optimal refactoring strategies. The refactoring tool we envision should be able to explain which clone differences can be parameterized or not, through a sophisticated and comprehensive visualization, suggest the changes required to make clones refactorable, detect sub-clones within larger clones that can be directly

TABLE I. NUMBER OF NODE COMPARISONS FOR OPTIMAL MAPPING

ID	Clone Type	# PDG nodes		CDT depth	# CDT leaves	# distinct node comparisons
		PDG <sub>1</sub>	PDG <sub>2</sub>			
1	Type-3	55	55	1	1	936
2	Type-2	50	50	3	7	603
3	Type-2	68	68	3	12	1040
4	Type-1	50	50	1	2	806
5	Type-1	67	67	1	25	1889
6	Type-3	93	94	4	11	1659
7	Type-3	51	50	4	7	455
8	Type-2	45	45	4	5	420
9	Type-2	70	70	7	8	577
10	Type-3	36	36	3	7	212
11	Type-2	42	42	7	7	235
12	Type-2	87	87	4	12	1812
13	Type-3	57	62	4	14	360
14	Type-3	42	43	5	6	136
15	Type-3	53	50	2	8	1021

refactored, and finally perform the corresponding refactoring transformations.

## ACKNOWLEDGMENT

The authors would like to thank NSERC and the Faculty of Engineering and Computer Science at Concordia University for their generous support.

## REFERENCES

- [1] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, 2001, pp. 40–56.
- [2] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–307.
- [3] D. Shepherd, E. Gibson, and L. L. Pollock, "Design and evaluation of an automated aspect mining tool," in *Proceedings of the International Conference on Software Engineering Research and Practice*, 2004, pp. 601–607.
- [4] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 75–84.
- [5] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, 2000, pp. 98–107.
- [6] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 872–881.
- [7] B. Fluri, M. Wuersch, M. Plnzer, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.
- [8] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.
- [9] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 214–225.
- [10] D. Conte, P. Foggia, and M. Vento, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs," *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 99–143, 2007.
- [11] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.
- [12] G. Valiente, *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.

# Multi-Abstraction Concern Localization

Tien-Duy B. Le, Shaowei Wang, and David Lo  
 School of Information Systems,  
 Singapore Management University, Singapore  
 {btdle.2012,shaoweiwang.2010,davidlo}@smu.edu.sg

**Abstract**—Concern localization refers to the process of locating code units that match a particular textual description. It takes as input textual documents such as bug reports and feature requests and outputs a list of candidate code units that need to be changed to address the bug reports or feature requests. Many information retrieval (IR) based concern localization techniques have been proposed in the literature. These techniques typically represent code units and textual descriptions as a bag of tokens at *one level of abstraction*, e.g., each token is a word, or each token is a topic. In this work, we propose *multi-abstraction* concern localization. A code unit and a textual description is represented at multiple abstraction levels. Similarity of a textual description and a code unit, is now made by considering all these abstraction levels. We have evaluated our solution on AspectJ bug reports and feature requests from the iBugs benchmark dataset. The experiment shows that our proposed approach outperforms a baseline approach, in terms of Mean Average Precision, by up to 19.36%.

## I. INTRODUCTION

Developers receive bug reports and feature requests through issue management systems such as Bugzilla and JIRA daily. The amount of these reports are often too many for developers to handle [1]. For each of these reports and requests, developers need to locate the code units that need to be modified to fix bugs or be extended to implement a particular feature. Considering a large code base with thousands or even millions of files, this task is a daunting one. Much manual effort needs to be spent to locate relevant code units. Thus, an automated solution is needed.

A number of approaches have been proposed to link bug reports and feature requests to the corresponding code units, e.g., [5], [12]. The bug reports and feature requests could be viewed as *concerns*<sup>1</sup> and the linking process is referred to as *concern localization*. Many past studies on bug localization, feature location, etc. could be viewed as specific instances of concern localization.

Many existing studies characterize both concerns (e.g., feature requests or bug reports) and code units as a bag (i.e., multi-set) of tokens at *one abstraction level* [5], [12]. A textual document (i.e., feature request, bug report, or code unit) could be represented as a set of words that appear in it. Alternatively, a natural language processing technique, referred to as topic modeling, e.g., [2], can be applied to infer a set of topics that appear in the document. A topic is a distribution of words and

<sup>1</sup>A concern is a concept, requirement, feature, or property related to a software system [10]. In this work, we focus on bug reports and feature requests which are subsets of concerns but the proposed approach could be used for generic concerns.

is a higher level abstraction of the original words. A set of topics can be inferred from documents and these topics would represent these documents. Similarities of documents can then be measured as the similarities of their representations (i.e., their set of words, or their set of topics). The code units that are most similar to the input concerns are output to the end user.

While many past studies only compare two documents at one abstraction level, in this work, we compare documents at multiple abstraction levels. A word can be abstracted at multiple levels of abstraction. For example, Eindhoven, can be abstracted to North Brabant, Netherlands, Western Europe, European Continent, Earth, and so on. Two documents might not share the same word “Eindhoven” but they might be about the same province (i.e., North Brabant), the same country (i.e., Netherlands), the same region (i.e., Western Europe), and so on. By viewing a document at multiple levels of abstractions the similarity or difference of two documents can be better assessed.

To represent documents in multiple abstraction levels, we leverage topic modeling. Topic modeling maps words that appear in a document to topics. Each word is assigned to one topic. The fewer the number of topics, the higher the abstraction level. This is the case as a topic now represents more words. On the other hand, the larger the number of topics, the lower the abstraction level. Thus we can iteratively apply topic modeling using different numbers of topics to create multiple abstraction levels. We can then aggregate these abstractions to measure the similarity between a concern (e.g., a bug report or a feature request) and a code unit.

In the literature, vector space modeling (VSM) has been shown to outperform many other information retrieval (IR)-based techniques for concern localization [9], [12]. In this paper, we extend VSM to consider multi abstraction levels. We refer to the resultant model as multi-abstraction VSM ( $VSM^{MA}$ ). We evaluate our approach on the iBugs dataset [3] which contains a few hundred of AspectJ concerns (i.e., bug reports and feature requests) and their corresponding code units. To demonstrate that the proposed multi-abstraction concept works, we compare our approach with the original VSM. The experiment results show that, in terms of mean average precision (MAP) [4], our multi-abstraction approach can outperform the original VSM by 19.36%.

Our contributions are as follows:

- 1) We propose multi-abstraction concern localization. We represent a document (i.e., a code unit, bug report, or

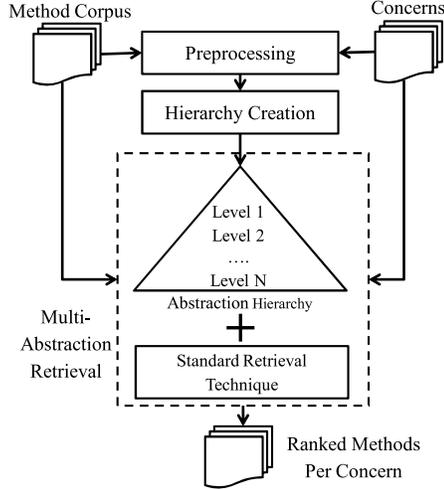


Fig. 1. Overall Framework of Multi-Abstraction Concern Localization

- feature request) at multiple abstraction levels.
- 2) We propose a technique that leverages multiple topic models to capture representations of documents at different abstraction levels. Our technique then uses these representations to compute the similarity between a concern and a code unit.
  - 3) We have evaluated our approach on hundreds of AspectJ concerns from the iBugs dataset. Our experiment shows that our proposed multi-abstraction concept works. In terms of MAP, our proposed approach can outperform the original VSM by 19.36%.

The structure of the paper is as follows. In Section II, we present the overall framework of multi-abstraction concern localization. In Section III, we discuss our multi-abstraction approach namely Multi-Abstraction VSM ( $VSM^{MA}$ ). We present our experimental results in Section IV. We review related work in Section V. We finally conclude and mention future work in Section VI.

## II. OVERALL FRAMEWORK

### A. Overview

Figure 1 presents the overall framework of multi-abstraction concern localization. Our framework takes as input *Method Corpus* and *Concerns*. *Method Corpus* is a collection of textual documents where each document corresponds to a method in the code base. Each document contains identifiers that appear in the source code of the method and comments (Java or Javadoc comments) that appear in or written for the method. *Concerns* is a collection of textual documents where each document is either a bug report or a feature request. For each bug report and feature request, we extract the text that appears in its title and description. The output of our framework is a set of ranked methods for each concern.

Our framework contains three processing steps: *Preprocessing*, *Hierarchy Creation*, and *Multi-Abstraction Retrieval*. The purpose of the *Preprocessing* step is to convert methods and bug reports into a standard representation, i.e., a bag

of words. The resultant bags of words are then input to the *Hierarchy Creation* step. The *Hierarchy Creation* step applies a topic modeling technique a number of times to construct the *abstraction hierarchy*. The *abstraction hierarchy* is a collection of topic models with various number of topics. Each topic model is a level in the hierarchy. The *abstraction hierarchy* is a part of the *Multi-Abstraction Retrieval* step. In this step, we enhance *standard retrieval techniques* by leveraging the *abstraction hierarchy*. The goal of the final processing step is to compare a concern (a query) and a method (a document in the *Method Corpus*) by considering multiple abstraction levels.

We elaborate the first two processing steps in the following subsections. Section III discusses the multi-abstraction retrieval step in more detail.

### B. Preprocessing Step

In this step, we first remove common Java keywords such as *public*, *private*, *class*, *extends*, etc., as well as punctuation marks and special symbols. These words are deemed useless for linking concerns and code units (i.e., methods) as either they appear in most documents or they carry little meaning. Thus, we only retain some word tokens and number literals.

We then break identifiers into word tokens by assuming that identifiers follow Camel casing convention which is the naming convention adopted by most Java programs. Following the Camel casing convention, for every class name, each word starts with a capital letter; for other identifiers, the second and subsequent words start with a capital letter. We use this convention to break an identifier into word tokens. We perform this step to standardize word tokens that are used in *Method Corpus* with those that are used in *Concerns*.

Next, we apply the Porter Stemming Algorithm<sup>2</sup> to reduce English words into their root forms. For example, “models”, “modeled”, “modeling” are all reduced to the same root word “model”. We perform this step to standardize words of the same meaning but are in different forms.

At the end of this step, we create a bag (i.e., a multi-set) of words for each concern and method.

### C. Hierarchy Creation Step

In the hierarchy creation step, we apply a topic modeling algorithm a number of times. We use Latent Dirichlet Allocation (LDA) which is a popular topic modeling algorithm [2]. LDA accepts as input the number of topics  $K$  and a set of documents<sup>3</sup> (in bag of words representation). It produces the following:

- 1)  $K$  topics, where each topic is a distribution of words.
- 2) For each document  $d$  and each topic  $t$ , LDA assigns a probability of topic  $t$  to appear in document  $d$ .
- 3) For each word  $w$  in document  $d$ , LDA assigns a topic to  $w$ . This topic is an abstraction of the word.

Each application of LDA creates a topic model with  $K$  topics. This topic model forms an abstraction level. We repeat

<sup>2</sup><http://tartarus.org/martin/PorterStemmer/>

<sup>3</sup>We set the other LDA parameters to their default values.

this step  $L$  times to create  $L$  abstraction levels. These  $L$  abstraction levels form an abstraction hierarchy  $H$ . Topic models with fewer topics are higher in the hierarchy while those with more topics are lower in the hierarchy. We refer to the number of topic models contained in a hierarchy as the *height* of the hierarchy.

At the end of this step, we create an abstraction hierarchy which is used in the next step: *Multi-Abstraction Retrieval*.

### III. MULTI-ABSTRACTION RETRIEVAL

In this section, we discuss how to combine an abstraction hierarchy with a text retrieval model (i.e., VSM). A retrieval method takes a query (i.e., a bug report) and returns a sorted list of most similar documents in a corpus (i.e., methods).

In standard VSM, a document is represented as a vector of weights. Each element in a vector corresponds to a word, and its value is the weight of the word. Term frequency-inverse document frequency (tf-idf) [4] is often used to assign weights to words. The following is the tf-idf weight of word  $w$  in document  $d$  given a corpus (i.e., a set of documents)  $D$  (denoted as  $tf\text{-idf}(w, d, D)$ ):

$$tf\text{-idf}(w, d, D) = \log(f(w, d) + 1) \times \log \frac{|D|}{|\{d_i \in D | w \in d_i\}|}$$

In the above equation,  $f(w, d)$  is the number of times word  $w$  appears in document  $d$ , and  $w \in d_i$  denotes that word  $w$  appears in document  $d_i$ . Given a query document  $q$ , standard VSM retrieval model would return the most similar documents in the corpus  $D$ . Similarity between two documents is measured by computing the cosine similarity between the two documents' vector representations [4].

In Multi-Abstraction VSM ( $VSM^{MA}$ ), we integrate abstraction hierarchy into standard VSM by extending the vector that represents a document. We added more elements to the vector. Each added element corresponds to a topic of a topic model in the abstraction hierarchy, and its value is the probability of the topic to appear in the document. The size of an extended document vector is  $V + \sum_{i=1}^L K(H_i)$ , where  $V$  is the size of the original document vector,  $L$  is the number of abstraction levels in the hierarchy, and  $K(H_i)$  is the number of topics of the  $i^{th}$  topic model in the abstraction hierarchy  $H$ . Based on this representation, the similarity between a query  $q$  and document  $d$ , considering a corpus  $D$ , calculated using cosine similarity, is as follows:

$$\begin{aligned} sim(q, d, D) &= \frac{\sum_{i=1}^V tf\text{-idf}(w_i, q, D) \times tf\text{-idf}(w_i, d, D) + \sum_{k=1}^L \sum_{i=1}^{K(H_k)} \theta_{q, t_i}^{H_k} \times \theta_{d, t_i}^{H_k}}{\|q\| \times \|d\|} \end{aligned}$$

where

$$\|q\| = \sqrt{\sum_{i=1}^V tf\text{-idf}(w_i, q, D)^2 + \sum_{k=1}^L \sum_{i=1}^{K(H_k)} (\theta_{q, t_i}^{H_k})^2}$$

and

$$\|d\| = \sqrt{\sum_{i=1}^V tf\text{-idf}(w_i, d, D)^2 + \sum_{k=1}^L \sum_{i=1}^{K(H_k)} (\theta_{d, t_i}^{H_k})^2}$$

In the above equations,  $\theta_{d, t_i}^{H_k}$  is the probability of topic  $t_i$  to appear in document  $d$  as assigned by the  $k^{th}$  topic model in the abstraction hierarchy  $H$ .

For example, assuming that a bug report  $br$  after text preprocessing has the following 7 words: “*suppress*”(3), “*warning*”(2), “*pointcut*”(2), “*aj*”(2), “*advice*”(1), “*lint*”(1), “*require*”(1). We also have two methods  $m_1$  and  $m_2$ . Each of them contains 5 words:  $m_1 = \{\text{“}i\text{suppress”}(7), \text{“}i\text{warning”}(4), \text{“}i\text{pointcut”}(3), \text{“}i\text{lint”}(7), \text{“}i\text{require”}(1)\}$  and  $m_2 = \{\text{“}i\text{suppress”}(10), \text{“}i\text{warning”}(10), \text{“}i\text{aj”}(5), \text{“}i\text{advice”}(4), \text{“}i\text{lint”}(6)\}$ . The number in parentheses is the number of times a word appears in a document. Let us assume that an abstraction hierarchy of height 1 is used, and the topic model has 3 topics. Let us also assume that there are 1000 methods, and terms in  $m_1$  and  $m_2$  do not appear in other methods. Considering only the 7 words, the representative vectors of  $br$ ,  $m_1$ , and  $m_2$  are:

$$\begin{aligned} V_{br} &= [1.62, 1.291.43, 1.43, 0.90, 0.81, 0.90, 0.26, 0.72, 0.02] \\ V_{m_1} &= [2.44, 1.89, 1.81, 0.00, 0.00, 2.44, 0.90, 0.00, 0.99, 0.00] \\ V_{m_2} &= [2.81, 2.81, 0.00, 2.33, 2.10, 2.28, 0.00, 0.57, 0.43, 0.00] \end{aligned}$$

The first 7 entries in each vector are the weights of the 7 words computed using the tf-idf formula, and the last 3 entries are rounded probabilities  $\theta_{d, t_i}^{H_1}$  of topics 1, 2 and 3 respectively in the documents. Finally, we calculate cosine similarities between bug report  $br$  and methods  $m_1$  and  $m_2$ . The results are  $sim(br, m_1) = 0.82$  and  $sim(br, m_2) = 0.84$ . Thus,  $m_2$  is more relevant to bug report  $br$  than  $m_1$ .

### IV. EXPERIMENTS & ANALYSIS

We use AspectJ concerns (i.e., bug reports and feature requests) from the iBugs dataset [3]. In iBugs, there are 350 AspectJ faulty versions, but some relevant methods in 65 versions cannot be found in the AspectJ codes that are included in the iBugs dataset. Thus, we exclude 65 concerns corresponding to these 65 versions. For each concern, we have its description, along with methods that are responsible for it, i.e., the method is changed to address the concern.

We measure effectiveness in terms of *mean average precision* (MAP) [4]. MAP has been used in past studies, e.g., [9]. A retrieval technique returns a sorted list of documents (i.e., methods) given a query (i.e., concern). The MAP of retrieval results corresponding to a set of queries (i.e., concerns)  $Q$  to retrieve relevant documents (i.e., methods) from a document corpus  $D$  is:

$$\begin{aligned} MAP(Q, D) &= \frac{\sum_{q \in Q} AvgP(q, D)}{|Q|} \quad (1) \\ AvgP(q, D) &= \frac{\sum_{k=1}^{|D|} P@k \times rel(k)}{\text{Total number of relevant methods}} \end{aligned}$$

TABLE I  
ABSTRACTION HIERARCHIES USED IN THE EXPERIMENTS

Hierarchies	Number of Topics
$H_1$	50
$H_2$	50, 100
$H_3$	50, 100, 150
$H_4$	50, 100, 150, 200

TABLE II  
EFFECTIVENESS OF MULTI-ABSTRACTION VSM OVER STANDARD VSM

	MAP	Improvement
Baseline (i.e., VSM)	0.0669	0%
$H_1$	0.0715	6.82%
$H_2$	0.0777	16.11%
$H_3$	0.0787	17.65%
$H_4$	0.0799	19.36%

In the above equation,  $AvgP(q, D)$  is the *average precision* for query  $q$ .  $P@k$  is the precision at  $k$  defined as the proportion of relevant methods among the top- $k$  methods in the retrieval results. Also,  $rel(k)$  is a function that returns 1 if the method returned at position  $k$  is relevant to the concern, and 0 otherwise.

We experiment with the 4 hierarchies  $H_1$ ,  $H_2$ ,  $H_3$ , and  $H_4$  of heights 1, 2, 3, and 4 respectively (listed in Table I). The number of topics in the topic model(s) of  $H_1$  is 50,  $H_2$  are 50 and 100,  $H_3$  are 50, 100, and 150, and  $H_4$  are 50, 100, 150, and 200. In this preliminary study, we arbitrarily decide the hierarchy heights and the number of topics.

Our experiment results are shown in Table II. The table shows that for Multi-Abstraction VSM, for all hierarchy settings ( $H_1$ ,  $H_2$ ,  $H_3$ , and  $H_4$ ), the performance is better than that of the baseline (standard VSM). Moreover, the MAP improvement for  $H_4$  is 19.36%. Furthermore, we note that the MAP is improved when the height of the abstraction hierarchy is increased from 1 ( $H_1$ ) to 4 ( $H_4$ ). Table III shows the number of concerns where the improvements in the *average precision* ( $AveP$ ) are within a particular range for  $H_1$  to  $H_4$ . We note that for the majority of the concerns the improvements are positive. For  $H_4$ , the improvements are positive for 222 out of the 285 concerns (77.89%).

## V. RELATED WORK

A number of past studies have employed various text retrieval techniques for concern localization [5], [12], [14]. Wang et al. [12] evaluate 10 information retrieval techniques and discover that VSM has the best performance. Rao and Kak also investigate the use of LDA with VSM [9]. However, in their approach, VSM is considered separately from LDA. The results of the two are combined together using a *weighted sum*. The performance of the resulting composite model is *worse* than that of VSM. In this work, we integrate LDA and VSM by constructing a single unified vector and we use a hierarchy of topic models; the resulting approach performs better than VSM. Aside from text, other sources of information, e.g., execution traces [8], have been used to aid concern localization. Our technique does not consider execution traces since most bug reports do not come with execution traces [11].

Petrenko and Rajlich proposes an impact analysis approach which leverages program dependencies at *multiple granulari-*

TABLE III  
NUMBER OF CONCERNS WITH VARIOUS  $AveP$  IMPROVEMENTS

Improvement ( $p$ )	Number of Concerns			
	$H_1$	$H_2$	$H_3$	$H_4$
$p < -10\%$	21	27	30	30
$-10\% \leq p < 0\%$	25	22	25	22
$p = 0\%$	18	14	12	11
$0\% < p \leq 10\%$	113	64	42	41
$p > 10\%$	108	158	176	181

ties (i.e., classes, class members, and code fragments) [7]. In this work, we address a different problem.

## VI. CONCLUSION AND FUTURE WORK

In this study, we propose multi-abstraction concern localization which combines a hierarchy of topic models with VSM. Our experiments on 285 AspectJ concerns shows that we can improve MAP of VSM by up to 19.36%.

In the future, we plan to perform a deeper analysis on cases where our multi-abstraction approach does not work well. Also, we want to extend our study by experimenting with different numbers of topics in each level of the hierarchy, different hierarchy heights ( $> 4$ ), and different topic models. We also want to analyze the effect of document lengths on the effectiveness of the proposed approach for different number of topics and hierarchy heights. Furthermore, we plan to experiment with Panichella et al.’s method [6] to infer good LDA configurations, and leverage other advanced text mining solutions, e.g., paraphrase detection [13], to further improve concern localization results.

## REFERENCES

- [1] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *ETX*, 2005, pp. 35–39.
- [2] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.
- [3] V. Dallmeier and T. Zimmermann, “Extraction of bug localization benchmarks from history,” in *ASE*, 2007, pp. 433–436.
- [4] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, 2008.
- [5] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in *ICSE 2003*.
- [6] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, “How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms,” in *ICSE*, 2013.
- [7] M. Petrenko and V. Rajlich, “Variable granularity for improving precision of impact analysis,” in *ICPC*, 2009, pp. 10–19.
- [8] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *TSE*, 2007.
- [9] S. Rao and A. C. Kak, “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models,” in *MSR*, 2011.
- [10] M. P. Robillard and G. C. Murphy, “Representing concerns in source code,” *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, 2007.
- [11] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, “Towards more accurate retrieval of duplicate bug reports,” in *ASE*, 2011, pp. 253–262.
- [12] S. Wang, D. Lo, Z. Xing, and L. Jiang, “Concern localization using information retrieval: An empirical study on linux kernel,” in *WCRE 2011*.
- [13] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, “Extracting paraphrases of technical terms from noisy parallel software corpora,” in *ACL/IJCNLP*, 2009.
- [14] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports,” in *ICSE*, 2012, pp. 14–24.

# Towards a Weighted Voting System for Q&A Sites

Daniele Romano  
Software Engineering Research Group  
Delft University of Technology  
Delft, The Netherlands  
Email: daniele.romano@tudelft.nl

Martin Pinzger  
Software Engineering Research Group  
University of Klagenfurt  
Klagenfurt, Austria  
Email: martin.pinzger@aau.at

**Abstract**—Q&A sites have become popular to share and look for valuable knowledge. Users can easily and quickly access high quality answers to common questions. The main mechanism to label good answers is to count the votes per answer. This mechanism, however, does not consider whether other answers were present at the time when a vote is given. Consequently, good answers that were given later are likely to receive less votes than they would have received if given earlier.

In this paper we present a *Weighted Votes (WV)* metric that gives different weights to the votes depending on how many answers were present when the vote is performed. The idea behind WV is to emphasize the answer that receives most of the votes when most of the answers were already posted.

Mining the Stack Overflow data dump we show that the WV metric is able to highlight between 4.07% and 10.82% answers that differ from the most voted ones.

**Index Terms**—Mining Repositories; Stack Overflow; Q&A Sites; Software Engineering; Metrics; Social Media; Social Coding

## I. INTRODUCTION

In the last decade question-answering web sites (Q&A) have become large repositories of knowledge. The key factors of their success are the ease and speed with which users can access valuable knowledge [1]. Among all the Q&A websites, Stack Overflow<sup>1</sup> has become the most popular site to share and look for software development knowledge [2].

In Stack Overflow, and in all the Q&A sites, the voting system is the main means to distinguish high quality answers from low quality ones [3]. Users can up-vote good answers, and down-vote bad answers. As consequence, users looking for good answers can easily focus their attention on answers that get more votes. However, such a voting system has a great disadvantage that can put good quality answers in the background. The count of the votes, on which users rely on, does not take into account the number of answers posted when a vote has been given. Most of the votes could be performed when only few answers to a question have been posted. Hence, the number of votes might not highlight the most valuable answer. As consequence users could be misled.

In this paper we propose a new way to count the number of votes that can overcome this problem. We introduce the *Weighted Votes (WV)* metric that gives different weights to votes depending on the number of answers already posted when a vote is given. The goal of the WV metric is to

emphasize the answers that receive most of the votes when most of the answers are present.

To analyze the ability of WV in highlighting answers different from the most voted ones we have mined the Stack Overflow data and computed the values of WV for 4,392,956 answers. The results show that WV ranks between 4.07% and 10.82% of the answers higher than the traditional approach. Moreover, we analyzed the extracted data to give an insight into the amount of answers already posted when votes are performed.

The remainder of this paper is organized as follows. In Section II we introduce the *Weighted Votes* metric, we reason about its integration into Q&A sites and discuss the benefits for their communities. Section III presents our study, its results and the process to extract the necessary data. We conclude this paper and draw directions for future work in Section V.

## II. THE WEIGHTED VOTES METRIC

When a user is looking for the valuable answer to a question of interest she may focus on the most voted answers, especially if the question gets numerous answers. However, the current voting system adopted by Q&A sites is limited to count the number of votes an answer receives along its lifetime. The main limitation of such a system is that most of the votes can be performed immediately after the answer is posted. Hence, they do not take into account the answers posted later.

We propose a new way to count votes that takes into account the number of answers to a question already posted when a vote is performed and the total number of answers. We suggest to give different weights to the votes depending on the number of answers already posted when it is given. For an answer  $A$  to a question  $Q$  we define the *WeightedVotes* metric ( $WV(A)$ ) as follows:

$$WV(A) = \sum_{k=1}^n \frac{Answers_Q < t_k}{Answers_Q} \quad (1)$$

where  $n$  is the number of votes given for the answer  $A$ ;  $Answers_Q$  is the total number of answers to  $Q$ ;  $t_k$  indicates the time when the vote  $k$  was performed and  $Answers_Q < t_k$  indicates the number of answers given to  $A$  and posted before the vote  $k$  was performed.

<sup>1</sup><http://stackoverflow.com>

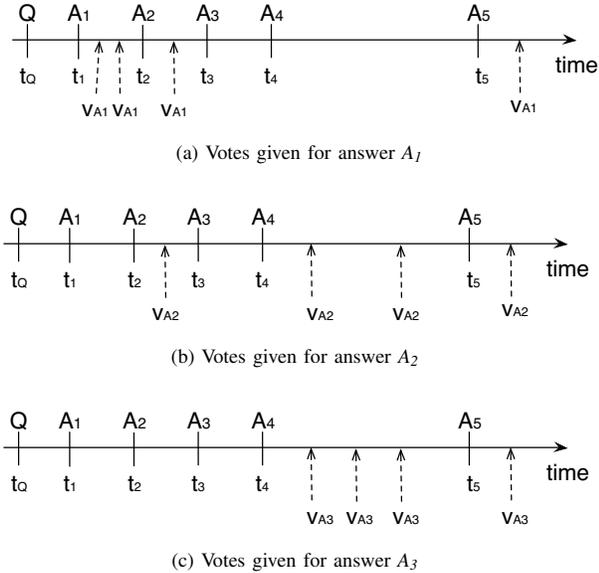


Fig. 1: Timelines showing five answers posted to a question  $Q$  and votes given for three answers ( $A_1, A_2$  and  $A_3$ )

#### A. Working Example

Consider the example shown in Figure 1. The example shows a question  $Q$  with five different answers posted at different times. Figure 1a, Figure 1b and Figure 1c display respectively the votes given for the answers  $A_1, A_2$  and  $A_3$ . According to the current voting system adopted by Stack Overflow all answers  $A_1, A_2$  and  $A_3$  will have the same amount of votes (*i.e.*, 4). As consequence, the user who is looking for the best answer would not know that most of the votes given for answer  $A_1$  had been given without considering the other answers. Precisely, two votes were performed when only the answer  $A_1$  was posted, one vote when the answer  $A_2$  was posted and only one vote when all the answers to  $Q$  were posted. On the other hand, the answer  $A_3$  is not emphasized by the number of votes. Even though its votes were performed when four out five answers were posted.

With our metric defined in 1, the ranking of the three answers differ. In fact, when computing the metric values for each answer after the last vote for the answers of question  $Q$  has been recorded, we obtain for  $WV(A_1) = \frac{1}{5} + \frac{1}{5} + \frac{2}{5} + \frac{5}{5} = \frac{9}{5} = 1.8$ ,  $WV(A_2) = \frac{2}{5} + \frac{4}{5} + \frac{4}{5} + \frac{5}{5} = \frac{15}{5} = 3.0$  and  $WV(A_3) = \frac{4}{5} + \frac{4}{5} + \frac{4}{5} + \frac{5}{5} = \frac{17}{5} = 3.4$ . Our metric  $WV$  clearly highlights the answer, namely in this example  $A_3$ , who obtained most of the votes when most of the answers were present.

#### B. Integration into Q&A sites

The computation of our proposed  $WV$  metric can be easily integrated into Q&A sites. The only requirement needed is the ability to update the value of  $WV$  for an answer when a new answer is posted. For instance, imagine that an answer  $A_6$  is posted after  $A_5$  in our example shown in Figure 1. In this

scenario we should update on the fly the  $WV$  values of the other answers. For example, the  $WV$  of the answer  $A_1$  would be recomputed as follows:  $WV(A_1) = \frac{1}{6} + \frac{1}{6} + \frac{2}{6} + \frac{5}{6} = \frac{9}{6} = 1.5$ .

#### C. Impact to the community

Besides the benefits for users looking for good quality answers explained in Section II-A, the  $WV$  metric can bring another important advantage for Q&A communities. According to our proposed metric, votes to later answers have a higher weight. This can stimulate people to provide more answers in order to receive votes with higher weights affecting consequently their reputation in the community [4].

### III. THE STUDY

In this study we mine the Stack Overflow system in order to measure the values for the  $WV$  metric for each answer posted. The *goal* consists in evaluating whether the  $WV$  metric highlights different answers compared to the ones highlighted by the number of votes. The *quality focus* is the ability of the  $WV$  metric to differentiate the answers with the highest values of  $WV$  from the most voted answers. The *perspective* is that of a Q&A site designer who wants to improve its voting system emphasizing votes performed when most of the answers to a question had already been posted. The *context* of this study consists of the latest official dump of the Stack Overflow data that contains all activities performed since July 2008 until August 2012. Among all Q&A sites we decided to mine the Stack Overflow system because it has become the most popular Q&A site for sharing software development knowledge. Moreover, among all the Q&A sites published on the Stack Exchange network<sup>2</sup> it provides the biggest data set for our analysis.

In this paper we answer the following research question:

*To what extent does the  $WV$  metric highlight answers different from the answers with the highest number of votes?*

In the following subsections, first we describe the process to extract the data necessary for our analysis. Then we report our results and observations about the extracted data.

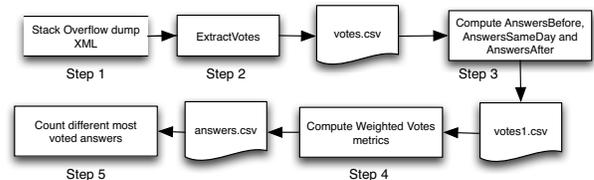


Fig. 2: Process used to extract the data for our analysis.

#### A. Data Extraction

Figure 2 shows the approach we used to extract the data from the Stack Overflow data dump.

<sup>2</sup><http://data.stackexchange.com>

TABLE I: Percentage of answers highlighted by *WV* that differ from the most voted ones for different categories of questions.

	<i>Answers</i> $\geq 2$	<i>Answers</i> = 2	<i>Answers</i> = 3	<i>Answers</i> $\geq 4$
Questions (%)	63.96%	29.38%	16.67%	17.92%
$WV_{high}$	10.31%	5.88%	10.75%	17.17%
$WV_{low}$	3.21%	2.26%	3.52%	4.48%
Average	6.76%	4.07%	7.13%	10.82%

In the first step we downloaded the data dump in XML format from the Stack Exchange website.<sup>3</sup> The data dump consists of five XML files that store information about the users (*users.xml*), the posts (*posts.xml*), the comments (*comments.xml*), the posts' history (*posthistory.xml*) and the badges (*badges.xml*).

In the second step, for each answer contained by *posts.xml* we extract the *up* and *down* votes from the *votes.xml* file. We discarded the votes for answers that have been removed from the database. The output of this step consists of the *votes.csv* file that for each vote contains 1) the *id* of the answer for which the vote has been given, 2) the *id* of the question of the answer and 3) the *creation date* of the vote. In total we extracted 13,700,939 votes of 4,392,956 answers given to 2,421,549 questions.

In the third step, we prepared the data to compute the values for the *WV* metric. To be able to measure the *WV* we needed for each vote  $k$  the count of all answers posted before the vote was given ( $Answers_Q < t_k$ ) and total number of answers ( $Answers_Q$ ) given to a question  $Q$ . However, differently from the *creation date* of answers, the *creation date* of a vote does not contain the information about hours, minutes and seconds. Its format is in the form month-day-year. As consequence we cannot know if the answers posted on the day when the vote  $k$  is given are actually performed before or after the vote. This format is used in all Stack Exchange data dumps and not only for Stack Overflow. For this reason we computed 1) the number of answers posted on the days that precede the day when a vote is given (*AnswersBefore*); 2) the answers posted on the same day (*AnswersSameDay*); and 3) the answers posted on the following days (*AnswersAfter*). These values allow us to estimate the actual value of the *WV* metric as explained in the next step. The output of this step consists of the *votes1.csv* that enriches the *vote.csv* file adding for each vote the values of (*AnswersBefore*), (*AnswersSameDay*) and (*AnswersAfter*).

Since we cannot order the *AnswersSameDay*, in the fourth step we computed the values of two variants of *WV*. We computed the values of  $WV_{low}$  and of  $WV_{high}$ . Computing  $WV_{low}$  we assume that the *AnswersSameDay* have been posted after the vote was given. On the other hand, computing  $WV_{high}$  we assume that the *AnswersSameDay* were posted before the vote has been given. In this way  $WV_{low}$  and  $WV_{high}$  are the lower and upper boundaries of the actual value of *WV*. The values for  $WV_{high}$ ,  $WV_{low}$  and the number of votes for each answer are saved in *answers.csv*.

<sup>3</sup><http://data.stackexchange.com/>

In the last step (Step 5 in Figure 2), for each question we compared the ranking of the answers obtained with the *WV* metric and the traditional approach and computed the ratios of answers for which the ranking differed.

## B. Results

Table I shows the results obtained. Among all the questions analyzed we report the results of questions with a number of answers greater than two ( $Answers \geq 2$ ). They account for 63.96% of all questions. For the questions with only one answer the value for *WV* is equal to the number of votes. Moreover, we report the results for questions with two answers ( $Answers=2$ ), questions with three answers ( $Answers=3$ ) and questions with four or more answers ( $Answers \geq 4$ ). We chose these values because they represents the median number of answers (*i.e.*, three) and the 75th percentile (*i.e.*, four).

From the results we can state that for the questions with more than two answers ( $Answers \geq 2$ ) the *WV* metric emphasizes on average 6.76% different answers. In such cases the user can focus on answers that received most of the votes when most of the answers were already posted. For questions with two, three and four or more answers we registered on average respectively 4.07%, 7.13% and 10.82% of different answers highlighted by the *WV* metric.

In conclusion, we can answer our research question stating that the percentage of different answers highlighted by *WV* is 1) between 3.21% and 10.31% for questions with two or more answers, 2) between 2.26% and 5.58% for questions with two answers, 3) between 3.52% and 10.75% for questions with three answers and 4) between 4.48% and 17.17% for questions with four or more answers. On average the *WV* metric highlights a percentage of different answers that ranges from 4.07% to 10.82%.

## C. Observations

Besides the *WV*'s ability of highlighting different answers we can make two important observations reading the results shown in Table I.

First, we can notice that the percentage of different answers highlighted with *WV* increases when we consider questions with a higher number of answers. For  $WV_{high}$  we registered an increment of  $\approx 292\%$  ( $17.17/5.88$ ) between questions with two answers and questions with four or more answers. For  $WV_{low}$  we registered an increment of  $\approx 198\%$  ( $4.48/2.26$ ) between questions with two answers and questions with four or more answers.

Second, we can notice the difference between the values measured for  $WV_{high}$  and  $WV_{low}$ . In order to understand

TABLE II: Paired Cliff’s delta effect sizes ( $d$ ) between *AnswersBefore*, *AnswersSameDay* and *AnswersAfter*. The effect size is considered negligible for  $d < 0.147$ , small for  $0.147 \leq d < 0.33$ , medium for  $0.33 \leq d < 0.47$  and large for  $d \geq 0.47$  [5].

Distribution1	Distribution2	Cliff’s d
<i>AnswersBefore</i>	<i>AnswersSameDay</i>	0.053
<i>AnswersBefore</i>	<i>AnswersAfter</i>	0.318
<i>AnswersSameDay</i>	<i>AnswersAfter</i>	0.232

this gap we analyzed the difference of the distributions of *AnswersBefore*, *AnswersSameDay* and *AnswersAfter* measured for each vote. We computed the Mann-Whitney p-value for paired samples for each pairs of distributions to test if the distributions were different. For all pairs we registered p-values smaller than 0.01 indicating that the distributions are considered statistically different. Moreover we computed the Cliff’s delta effect size (for paired samples) [5] to measure the magnitude of the difference and we report the results in Table II.

The results show that the difference in magnitude between the distribution of answers posted on days before the day when a vote is given (*AnswersBefore*) and the distribution of answers posted on the same day of a vote (*AnswersSameDay*) is negligible ( $d=0.053 < 0.147$ )[5]. The distribution of answers posted after a vote (*AnswersAfter*) is smaller than the distributions of *AnswersBefore* and *AnswersSameDay* because the effect sizes’ values ( $d=0.318$  and  $d=0.232$ ) are considered to be medium [5]. From these results we can state that the distributions of *AnswersBefore* and *AnswersSameDay* are the biggest ones. This explains the difference between the values for  $WV_{low}$  and  $WV_{high}$  registered in our study.

#### IV. RELATED WORK

In the last years many studies on Stack Overflow have been presented. The closest to our study has been developed by Schall *et al.* [6]. They analyzed the dynamics of the community activities. As part of this analysis they analyzed the answering behavior per question showing the number of answers per different categories of questions. However, they have not analyzed the voting behavior and the main focus of their work is the mining of expertise.

Among all scientific work about mining Q&A sites, mining expertise from Q&A communities is becoming more and more popular. Many of them propose technique to mine expertise of users in the community, such as [7] [8] [9] [10] [11]. These works propose techniques and approaches to infer the expertise from several variables. Among these variable the number of votes plays a crucial role. The  $WV$  metric proposed in this paper can help to improve these approaches. For example, it can be used to filter votes given when only one answer is posted.

#### V. CONCLUSION AND FUTURE WORK

In this paper we proposed a *Weighted Votes* metric aimed at highlighting answers that received most of the votes when

most of the other answers were already given to a question. Mining the Stack Overflow data dump, we showed that the proposed metric is able to emphasize answers different from the most voted ones. This is particularly useful for users who are looking for high quality answers.

In our future work we plan to further validate and improve this metric. First, we plan to look for data in which the complete timestamp of a vote is registered. This allows us to obtain more precise results avoiding the approximations performed in this study (*i.e.*, the computation of  $WV_{high}$  and  $WV_{low}$  to estimate the actual value of  $WV$ ).

Second, we plan to perform a qualitative study to test to which extent the number of votes is relevant to users looking for answers. It is particularly useful to investigate if users go through all the answers or if they read only the most voted ones or the accepted ones.

Finally, we plan to perform a qualitative analysis with questionnaires to find out whether the answers highlighted by the *Weighted Votes* metric are considered of better quality compared to the most voted ones.

#### ACKNOWLEDGMENT

This work has been partially funded by the NWO-Jacquard program within the ReSOS project.

#### REFERENCES

- [1] C. Treude, F. Figueira Filho, B. Cleary, and M.-A. Storey, “Programming in a socially networked world: the evolution of the social programmer,” in *The Future of Collaborative Software Development*, 2012, pp. 1–3.
- [2] L. Manykina, B. Manoim, M. Mittal, G. Hripscak, and B. Hartmann, “Design lessons from the fastest q&a site in the west,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 2857–2866.
- [3] C. Treude, O. Barzilay, and M.-A. D. Storey, “How do programmers ask and answer questions on the web?” in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 804–807.
- [4] A. Anderson, D. P. Huttenlocher, J. M. Kleinberg, and J. Leskovec, “Discovering value from community activity on focused question answering sites: a case study of stack overflow,” in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2012, pp. 850–858.
- [5] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.
- [6] D. Schall and F. Skopik, “An analysis of the structure and dynamics of large-scale q/a communities,” in *Proceedings of the International Conference on Advances in Databases and Information Systems*, Berlin, 2011, pp. 285–301.
- [7] A. Pal, R. Farzan, J. A. Konstan, and R. E. Kraut, “Early detection of potential experts in question answering communities,” in *Proceedings of the International Conference on User Modeling, Adaptation, and Personalization*, 2011, pp. 231–242.
- [8] N. Raj, L. Dey, and B. Gaonkar, “Expertise prediction for social network platforms to encourage knowledge sharing,” in *Proceedings of the International Conference on Web Intelligence*, 2011, pp. 380–383.
- [9] B. V. Hanrahan, G. Convertino, and L. Nelson, “Modeling problem difficulty and expertise in stackoverflow,” in *Proceedings of the International Conference on Computer Supported Cooperative Work and Social Computing*, 2012, pp. 91–94.
- [10] A. Pal, F. M. Harper, and J. A. Konstan, “Exploring question selection bias to identify experts and potential experts in community question answering,” *ACM Trans. Inf. Syst.*, vol. 30, no. 2, pp. 10:1–10:28, 2012.
- [11] A. Pal, S. Chang, and J. A. Konstan, “Evolution of experts in question answering communities,” in *Proceedings of the International AAAI Conference on Weblogs and Social Media*, 2012.

# Latent Co-Development Analysis based Semantic Search for Large Code Repositories

Rahul Venkataramani  
International Institute of Information Technology  
Bangalore, India  
rahul.venkataramani@iiitb.org

Allahbaksh Asadullah, Vasudev Bhat, Basavaraju Muddu  
Infosys Labs, Infosys Ltd.  
Bangalore, India.  
{allahbaksh\_asadullah, vasudev\_d, mbraju}@infosys.com

**Abstract**—Distributed and collaborative software development has increased the popularity of source code repositories like GitHub. With the number of projects in such code repositories exceeding millions, it is important to identify the domains to which the projects belong. A domain is a concept or a hierarchy of concepts used to categorize a project. We have proposed a model to cluster projects in a code repository by mining the *latent co-development network*. These identified clusters are mapped to domains with the help of a taxonomy which we constructed using the metadata from an online Question and Answer (Q&A) website. To demonstrate the validity of the model, we built a prototype for semantic search on source code repositories. In this paper, we outline the proposed model and present the early results.

**Index Terms**—Software repository analysis and mining, source code repositories, semantic search, Human aspects of software evolution

## I. INTRODUCTION

Distributed software development and the resulting need for collaboration among developers have made software code repositories with version control ubiquitous in the software development process. This has resulted in millions of projects being stored in popular source code repositories like GitHub<sup>1</sup>.

For easy retrieval of projects in such large repositories, it is necessary for semantics like the *domain* of a project to be captured. However, conventional source code repositories do not capture such semantics. For example, the query “distributed systems” is interpreted by the search engine of code repositories as a text string rather than as a *domain*. We define *domain* as a collective notion that exists among a statistically significant population of the developer community about a group of projects belonging to a shared concept. The domain is not an implementation technology but a concept to which a number of projects and technologies belong. For example, a technical professional with relevant experience would immediately note “Hadoop”, “Voldemort”, “STORM” among other projects as one of the top projects belonging to the domain “distributed systems”. While developers are aware of the *domain* of the projects to which they contribute, they might not explicitly mention the *domain* in the *wiki* page of the project due to a number of reasons ranging from carelessness to the domain being obvious given the context. This semantic information about the *domain* of a project is not stored in a

code repository. This makes retrieval of projects from a source code repository on the basis of *domain* a challenging problem.

A model to identify the domains to which projects belong is necessary in the development of a semantic search engine for code repositories. A few systems like SourceForge<sup>2</sup> have partially overcome this problem by making it compulsory for users to annotate projects with tags before uploading them. However, since the categories are predefined by humans, updation and maintenance of such categories require additional effort. Another classification approach based on similarity measures calculated from source code artifacts [11] cannot always be used because of restricted access to source code in many corporate repositories. In such an approach, developing tools to extract source code artifacts for every programming language is practically infeasible.

In this paper, we propose a model to capture the notion of *domain* of projects belonging to a source code repository by mining the network formed between developers contributing to same project. We call this network as *latent co-development network*. We use this network to detect clusters of semantically related projects. These identified clusters are mapped to domains with the help of a taxonomy which we constructed using the metadata from an online Q&A website. The proposed model can also be used for other allied applications including categorization of projects, measuring popularity of projects among the developer community, skilled recruitment, recommendations etc.

In section II we present a brief overview of related work. Section III describes a model for identifying domains of projects. Section IV describes the architecture of the semantic search that we built. Section V and VI discuss the experimental setup and the initial results obtained, respectively. We conclude our paper in section VII and provide an outline for future work.

## II. RELATED WORK

A number of attempts [6], [7], [9] have been made to extract topics from projects through the use of source code artifacts. Latent Dirichlet Analysis on source code artifacts has been popularly used in the software engineering community to extract topics from projects. However, in the current work we hold the view that mining the latent co-development

<sup>1</sup>www.github.com

<sup>2</sup>www.sourceforge.net

network reveals sufficient semantics to understand the domain of projects. Mining the developer contribution network from source code repository has been used to solve a wide variety of problems including understanding collaboration and influence dynamics [1], [4], predicting software failures [10] etc. We are not aware of any other model to identify domains in a code repository using developer-project contribution network.

### III. THE MODEL

Given a source code repository consisting of many projects the problem we address is the following: *find the domain(s) to which the projects belong.*

The proposed model can be described briefly in the following sections:

- Cluster detection of similar projects in code repository
- Mapping project clusters to a *domain* taxonomy

#### A. Cluster Detection of Similar Projects in Code Repository

Bird et al. [2] claim that the software components within a project that are developed by the same developer are implicitly related. We extend this claim to software projects in a source code repository and argue that the number of common developers between the projects represent the degree of similarity between them.

Formally, *If projects  $P_1$  and  $P_2$  share 'd' common developers and  $d > k$ , where  $k$  is a minimum threshold then the two projects are implicitly related to each other.*

We further claim that this similarity is because of the same or related domains that the projects belong to and not just because of the implementation technology used. Empirically, we observed that such behavior among developers is more pronounced in popular domains like distributed systems, NOSQL databases, machine learning etc.

An affiliation network (see Fig.1) between developers and projects captures the latent co-development relations in a source code repository. We analyze the latent co-development network in code repository to detect clusters of similar projects and then assign them to the respective domains to which they belong.

An affiliation network is an undirected bi-partite graph with two types of nodes - developers  $D$  and projects  $P$  and can be formally defined as  $A = (D, P, T)$ .  $D$  and  $P$  are the two sets of vertices that represent developers and projects, respectively.  $T$  is the set of edges that represent the contribution of developers towards projects  $T \subseteq \{(d, p) \mid d \in D \ \& \ p \in P\}$ . Using this representation of a source code repository, we construct a project similarity graph  $G = (V, E, w)$  to capture the similarity between projects in a source code repository (see Fig.2).  $V$  is the set of vertices that represents all the projects in a code repository.  $E$  is the set of edges representing *similarity* between projects. The degree of similarity between projects is represented by a count of common developers. The function  $w : E \rightarrow \mathbb{N}$  represents the degree of similarity. The weight of an edge  $E_{ij}$  between two projects  $P_i$  and  $P_j$  is calculated by  $w_{ij} = \{|d \in D \mid (d, P_i) \in T \ \& \ (d, P_j) \in T\}$ .

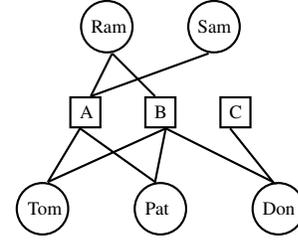


Fig. 1. An example of developer contribution affiliation network. Circles denote developers and rectangles denote projects. An edge indicates the contribution of a developer to a project.

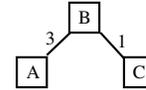


Fig. 2. Project similarity graph constructed from the developer contribution network in Fig. 1. The edge weights indicate the number of common developers between the two projects.

We extract clusters of projects  $C = \{C_1, C_2, \dots, C_n\}$  from the project similarity graph  $G$  using the community detection algorithm proposed by Vincent et al. [3].

#### B. Mapping project clusters to a domain taxonomy

Each of the project clusters obtained has to be mapped to domain(s). The *wiki* page of a project describes its goals and the implementation technology used. Since there is no explicit mention of domain names in a *wiki* page, there exists a need for a taxonomy to encompass all the terms in software engineering with the domain names subsuming the respective technologies used for implementation. For example, *Haskell*, *Scala*, *Lisp* belong to the domain of *functional programming languages*. A taxonomy can be formally represented as a set of concepts and the relationships between the concepts. We adopt the following definition proposed by Kaipeng et al. [8]

**DEFINITION 1 (Taxonomy).** *A taxonomy is a structure  $O = (I, root, \preceq_I)$  consisting of i) a set  $I$  of concept identifiers, ii) a designated root element, i.e. root representing the top element of the iii) upper semi-lattice  $(I \cup \{root\}, \preceq_I)$  called concept hierarchy.*

The nodes in the higher levels of hierarchy represent the domains while the lower levels represent implementation technologies and tools. This taxonomy can be generated by a number of means like manual curation, inducing a taxonomy from a folksonomy dataset etc. Assuming the availability of a taxonomy to satisfy our requirements, we discuss the mapping of a cluster of projects to concept(s) in a taxonomy. The mapping between project clusters and concepts in the taxonomy is given by the relation  $M : C \times I \rightarrow n$ .  $n$  denotes the strength of mapping between a cluster and a concept in the taxonomy. The project descriptions of each of the projects in the cluster are consolidated and this singular description of a cluster is mapped to concept identifier(s) in the taxonomy. This

mapping from a cluster descriptor to concept identifier(s) in a taxonomy can be performed in a number of ways depending upon the application for which the model is used for.

#### IV. APPLICATION OF THE MODEL : SEMANTIC SEARCH

In this section, we demonstrate the application of the model to implement semantic search over large code repositories to browse through the projects belonging to a specified domain. In the following sections, we describe the architecture of the semantic search engine that we built using our proposed model.

##### A. Mapping clusters of projects to a taxonomy

We constructed a taxonomy for software engineering by converting the folksonomy dataset obtained from a Q&A website (eg. StackOverflow<sup>3</sup>) by using the technique outlined by Liu et al. [8]. Most of the programmers contributing to source code repositories also post questions and answer queries on such technical forums. Hence, the list of tags extracted from such a dataset is fairly exhaustive to construct a taxonomy.

From a partial dump of GitHub, we constructed the project similarity graph  $G$ . We used the community detection algorithm proposed by Vincent et al. [3] to obtain clusters of related projects. We used this algorithm because it offers a fair compromise between the accuracy of the estimate of the modularity maximum and the computational complexity [5]. The *wiki* pages of projects in a cluster are aggregated to form a single description of the cluster from which we extracted only the *technical* terms. We restricted our vocabulary of *technical* terms to the tags used to annotate questions in the Q&A website. Depending on the frequency of *technical* terms in a cluster descriptor, each project cluster is assigned to concept identifier(s) in a taxonomy which is constructed as described above.

##### B. Query Expansion

A user enters a query which represents a *domain*. We expand the given query to a set of semantically related terms to facilitate better search results. We use the metadata from a large Q&A website to find semantically related terms to the given query. A Q&A website is a dataset of the form  $F : (Q, T, \alpha)$ , where  $Q$  represents the set of questions,  $T$  the set of tags used to annotate the questions and the binary relation between the tags and questions is represented by  $\alpha \subset Q \times T$ . Using this dataset, we constructed a tag co-occurrence graph  $G = (T, E, w)$  to model the developer community's perception of related tags. Edges  $E$  represent the co-occurrence between the tags  $T$  and  $w$  represents the degree of co-occurrence between tags. Each term in the query  $S$  is mapped to a tag  $T_i$  in the graph  $G$ . Since all the terms in the neighborhood are not equally important they are weighted by a factor  $w_{ij}$ , representing the weight of the edge between the given query tag  $T_i$  and the neighboring tag  $T_j$ . We use the top  $k$  neighboring tags to augment the query. If the query term

<sup>3</sup>www.stackoverflow.com

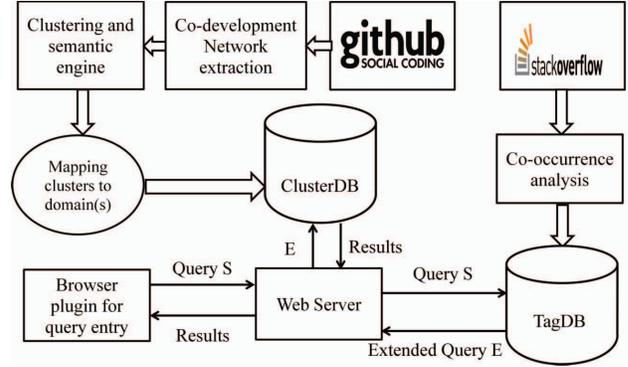


Fig. 3. Block diagram of the system architecture

contains more than a single word representing two or more tags in the graph  $G$ , the intersection of neighbors of the terms are given higher precedence by weighting them appropriately. We listed programming languages as stopwords since they are ubiquitous in a dataset of this nature.

##### C. Ranking clusters of projects

We normalize the mapping between cluster descriptors to concepts in a taxonomy by using the term frequency-inverse document frequency(TF-IDF) measure. The score  $R$  of a cluster  $C_i$  for a query  $S$  is defined as the sum of the product of the weights of each term in a query with the matching domain term.  $R(C_i) = \sum_{s \in S} w(s) \times M(s)$ . The clusters are ordered according to the obtained scores. The projects within a given cluster are displayed according to a parameter which can be changed at the user's discretion. Some parameters include best string match, last updated project, number of forks, number of watchers etc.

#### V. DATASETS AND EXPERIMENTAL SETUP

In this section, we first describe the dataset and the experimental setup used in the experiments.

We used two datasets for conducting experiments: Stack-Overflow dataset for taxonomy construction and a partial repository of GitHub to test our hypothesis. StackOverflow is a collaboratively edited Q&A website for programmers and is one of the most used websites for posting technical questions. We used the latest data dump from StackOverflow which was updated till March, 2013. The raw dataset consists of 4,189,241 questions and 32,051 unique tags to annotate the questions.

We used the GitHub REST API to retrieve a partial list of projects to form our experimental source code repository. Our partial dataset consists of 30,906 Java projects contributed to by 22,322 developers. Using this partial GitHub dump, we constructed the project similarity graph  $G$ . We obtained 150 project clusters after applying the community detection algorithm. We ignored the clusters with less than 5 projects because such clusters are typically formed because of contributions from a single developer and is mostly written for personal or

TABLE I  
EXPERIMENTAL RESULTS FOR SIMILARITY VALUES

Similarity measure	Similarity
Jaccard	0.72
Overlap	0.85

academic purposes. The setup for the query expansion engine and ranking of clusters is hosted on an internal server. The results are displayed to the user through a Google Chrome extension which displays our results along with default search results from GitHub.

## VI. EVALUATION RESULTS

The evaluation of our results is carried out in two stages. In the first stage, we performed a user evaluation to validate the relevance of domain(s) detected for each project cluster using our proposed model. We requested developers to tag each cluster with names of domain(s) that they think the cluster belongs to. In order to avoid the bias of a single developer, we used crowdsourcing techniques to tag the clusters. We selected 15 developers working across various teams at Infosys<sup>4</sup> to tag 10 clusters of projects each in the first round. In the next round, the clusters were exchanged between the developers and they were requested to perform the same exercise again. The tags obtained from the two rounds were consolidated to represent the domain(s) which the volunteers perceived. We measured the similarity of the consolidated set of tags obtained from developers for a cluster with the domains obtained from our model using the Jaccard and overlap similarity measures. Let  $X_1$  represent the domains obtained from our model and  $X_2$  represent the consolidated set of tags for a cluster obtained from developer tagging.

The Jaccard similarity is given by:

$$J(x_1, x_2) = \frac{|X_1 \cap X_2|}{|X_1 \cup X_2|} \quad (1)$$

The overlap similarity is defined as follows:

$$\sigma(x_1, x_2) = \frac{|X_1 \cap X_2|}{\min(|X_1|, |X_2|)} \quad (2)$$

The average value for the similarity measures for all the project clusters is listed in Table I. The high values of similarity measures obtained indicate a strong correlation between the domain(s) obtained by our model and the existence of a notion of domain(s) among the developer community.

In the second stage of evaluation, we observed the results obtained from semantic search qualitatively. A few queries along with their top results are displayed in Table II. Given the limited source repository for testing, it is encouraging to find that most of the projects are closely related to the query domain. We manually compared the search results with GitHub’s native search engine and observed that our results are semantically more relevant to the given query.

<sup>4</sup>www.infosys.com

TABLE II  
SEARCH RESULTS FOR A SAMPLE OF QUERIES

Query	Search Results
social networks	Twitter4J, FourSquareJavaAdaptor, Trie4j Face4j, openGraphJava, prettyTimeSocial
knowledge representation	libSBOLJava, libSBOLRDF, javaOWLSensor JavaOWLSolver, RESTOWLCommon
games	pinBall, Sudoku, MineSweeper Tetris, Pentago, TicTacToe
cloud computing	java-mongoDB-connector, javaAmazonDB jClouds, javaCloudFoundry, heroku, rabbitMQ

## VII. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated a technique to mine latent semantics in a source code repository using the developer contribution network. We proposed a model to detect *domain* of projects in source code repositories. We corroborated our hypothesis with experimental evidence. This work is a part of a larger vision to model the entire software engineering ecosystem comprising of source code repositories, developers, Q&A websites, technical blog posts, wikis, and use the information to arrive at better insights. In the future, we plan to explore other semantics that can be extracted from source code repositories and demonstrate the same through applications.

## ACKNOWLEDGMENT

The authors would like to thank Kshitiz Bakshi for his contribution in developing the browser plugin and the developers who helped in user evaluation. We would like to express our gratitude to Dr. Srinivas Padmanabhuni for his guidance.

## REFERENCES

- [1] X. Ben, S. Beijun, and Y. Weicheng. Mining developer contribution in open source software using visualization techniques. In *ISDEA, 2013*, pages 934–937. IEEE, 2013.
- [2] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *ISSRE*, pages 109–119. IEEE, 2009.
- [3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *JSTA*, (10):P10008, 2008.
- [4] B. Heller, E. Marschner, E. Rosenfeld, and J. Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th WRCE*, pages 223–226. ACM, 2011.
- [5] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, Nov 2009.
- [6] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.
- [7] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. *ASE ’07*, pages 461–464, New York, NY, USA, 2007. ACM.
- [8] K. Liu, B. Fang, and W. Zhang. Ontology emergence from folksonomies. In *CIKM*, pages 1109–1118. ACM, 2010.
- [9] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference*, pages 113–120. ACM, 2008.
- [10] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT FSE*, pages 2–12. ACM, 2008.
- [11] T. Wang, G. Yin, X. Li, and H. Wang. Labeled topic detection of open source software from mining mass textual project profiles. In *LAMDA*, pages 17–24. ACM, 2012.

# Differentiating Roles of Program Elements in Action-Oriented Concerns

Emily Hill

Montclair State University  
Montclair, NJ 07043  
hillem@mail.montclair.edu

David Shepherd

Industrial Software Systems ABB Corporate Research  
Raleigh, NC, 27606  
david.shepherd@us.abb.com

Lori Pollock and K. Vijay-Shanker

University of Delaware  
Newark, DE 19716  
{pollock, vijay}@cis.udel.edu

**Abstract**—Many techniques have been developed to help programmers locate source code that corresponds to specific functionality, i.e., concern or feature location, as it is a frequent software maintenance activity. This paper proposes operational definitions for differentiating the roles that each program element of a concern plays with respect to the concern’s implementation. By identifying the respective roles, we enable evaluations that provide more insight into comparative performance of concern location techniques. To provide definitions that are specific enough to be useful in practice, we focus on the subset of concerns that are action-oriented. We also conducted a case study that compares concern mappings derived from our role definitions with three developers’ mappings across three concerns. The results suggest that our definitions capture the majority of developer-identified elements and that control-flow islands (i.e., groups of elements with little to no control flow connections) can cause developers to omit relevant elements.

**Index Terms**—concerns; evaluation; software maintenance

## I. INTRODUCTION

Programmers who maintain software, such as adding new features, fixing bugs, or other evolution tasks, spend considerable time locating the program elements relevant to the feature (also called a *concern* [1]). Whether due to programming skill level, familiarity with the code, or other reasons [7], programmers do not often agree on what program elements are relevant to a given feature [2]. This causes difficulty in evaluating, comparing, and providing direction for improving feature location techniques.

Existing concern and feature location evaluations consider relevance to be binary; each program element is either judged as part of the concern or not. However, we have observed in our experience that not all program elements of a concern are equal. Some program elements play a major role in implementing the feature’s functionality, while others play a less key role, but are important to understand how the feature is implemented and interacts with its surrounding source code context. Our hypothesis is that more informative analysis of feature location techniques can be achieved if they are evaluated with respect to the roles that each program element is playing in the concern and its surrounding context. Additionally, feature location tools could be designed to provide more informative feedback about the elements in the concern.

In this paper, we propose operational definitions for differentiating the roles that each program element of a concern

plays with respect to the concern’s implementation. Specifically, we describe the characteristics of a program element that takes on the role of an *action*, *trigger*, *result*, or *connector* node in a program structure graph representation of a program. In this initial work, we focus on a specific kind of concern: action-oriented concerns [3]. Action-oriented concerns can be specified using a precise verb phrase (VP), which not only includes a verb and direct object, but also an indirect object. For example, not just “add a song”, but “add a song to a playlist”. Action-oriented concerns may implement user-observable features or be object-oriented, while there are action-oriented concerns that are neither. In the future, we hope to extend this work to a broader set of concerns.

Previous concern mapping studies [2] have investigated the agreement among developers or compared the opinions of a newcomer with the code’s author [4]. Neither study has investigated the nature of a concern’s elements and differentiated among the different roles the program elements might play. If we can identify the roles of program elements in gold sets, we can study how different feature location techniques identify the different kinds of program elements. For example, some feature location techniques may identify only the action nodes, while another technique also identifies and displays trigger nodes. In addition, with the capability to identify the roles automatically, a feature location tool could provide different display options to the programmer based on the programmer’s preferences during maintenance.

This paper makes the following contributions:

- A classification of roles that program elements play in an action-oriented concern with a set of operational definitions that enable precise, objective role-labeling
- A motivating example illustrating the labeling of concern elements with their respective roles
- A preliminary study analyzing human-annotated concerns with respect to roles of each element and annotator agreement

## II. MOTIVATING EXAMPLE

jBidWatcher is an auction bidding, sniping, and tracking tool for online auction sites such as eBay or Yahoo. It includes a unique sniping feature that allows the user to place a bid in the closing seconds of an auction. Before a user can bid on an auction, they must add the auction to the user view and data

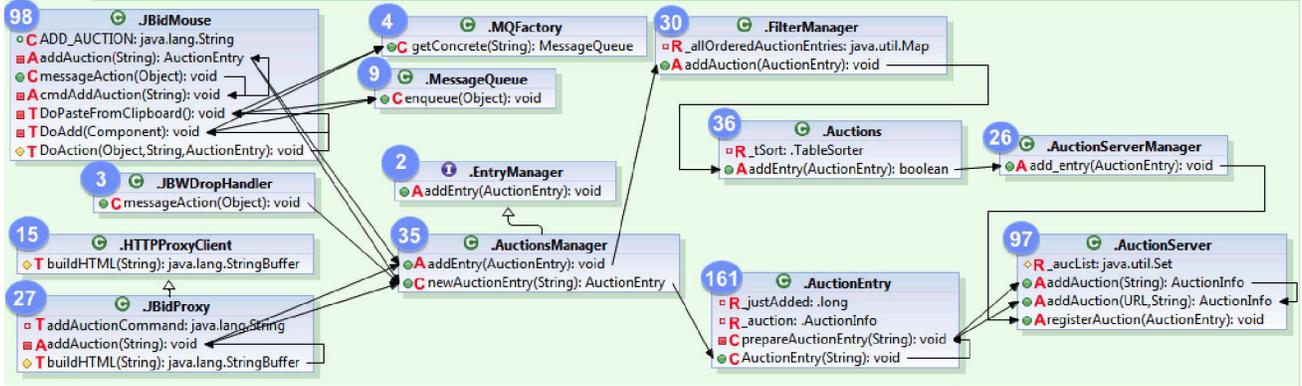


Fig. 1: Program elements and roles for the example “add new auction to local system” concern. Each element (method and field) is annotated with its role in red to the left of the name. The numbers annotating each class in the upper left hand corner are the total number of additional methods and fields in the class, aside from what is displayed in the diagram. Edges indicate structural relationships such as calls (solid-head arrows) and inheritance (open-head arrows).

structures. We define the verb phrase (VP) for this concern to be “add new auction to local system.” Figure 1 shows the code that implements the concern.

Nodes are added to the concern for a number of reasons. Some methods trigger the execution of the concern, such as the “do” methods in `JBidMouse` or `JBWDropHandler.messageAction`. Some methods are relatively generic, used to process all user-initiated actions, such as `MessageQueue` and `MQFactory`. Although generic, these methods communicate (or *connect*) information from the triggers to methods that implement the concern’s actions. The action of adding an auction culminates in updating several internal data structures: the set of auction entries being managed by the system (`FilterManager._allOrderedAuctionEntries`), the table of auctions displayed by the user interface (`Auctions._tSort`), and the list of auctions being managed by the internal auction server (`AuctionServer._aucList`). These are some of the *results* of the concern. Although creating a new `AuctionEntry` object is not obviously part of the add auction concern, since it can be precisely described by its own VP, creating a new auction object culminates in adding that `AuctionInfo` to the system with the `addAuction` methods in class `AuctionServer`.

This example demonstrates that a program element may be included as part of an action-oriented concern for different reasons, and that implementing the main action is only one of the reasons. In Figure 1, each element (method and field) is annotated with its role in red to the left of the name. Some concern elements are included because they trigger, or initiate, the concern action (indicated by ‘T’), whereas other elements execute the action of adding an auction by creating `AuctionEntry` objects and adding them to the internal data structures of the system (indicated by ‘A’). The data structures and fields updated as a result of the action are indicated by ‘R’, with connecting elements that communicate between different parts of the concern labelled as ‘C’.

### III. CONCERN ELEMENT ROLES

An underlying premise is that action-oriented concerns are described by a VP that includes a verb, direct object, and an indirect object. The action, or verb, is key to determining whether program elements (i.e., methods and fields) should be included in the concern. The direct and indirect objects are also important, with the direct object more prominently used in identification because fields rarely refer to actions and verbs. The indirect object helps further differentiate nodes to be included in the concern by indicating concern boundaries and helping to determine when one concern has become another.

One issue could be when the concern’s implementation does not use the verb in the VP to describe the action. For example, *delete* might be implemented as the synonym *remove*, and the concept of *adding* may include *creating* a new item. Plus, an objective definition of action-oriented concerns should not be so fragile as to require exact word matching in the source code. Thus, in the definitions below, we make use of the concept of *similarity*, where two words or phrases are semantically equivalent (i.e., synonyms).

We define four distinct roles of nodes in a structural representation of a concern:

**Action Node (A):** Any method that directly implements the concern’s verb phrase. The name need not explicitly refer to the VP, since method naming can be arbitrary (especially for overridden methods). Action nodes can also serve as concern trigger points (see trigger nodes, below).

**Trigger Node (T):** Any method that triggers the execution of an action node, either directly or through connector nodes, but does not implement the concern’s VP. Trigger nodes usually contain a reference to the concern’s VP, and serve as an entry point into the concern from outside the concern. If a node is a trigger point for the concern but also implements the concern’s VP, then the node is considered to be an action node. This role also applies to fields that are explicitly used to trigger the concern and have a strong relationship with the

concern’s VP. We consider initialization of the trigger code to be its own concern.

**Result Node (R):** Any field that has a similar object to the concern’s VP that is altered by an action (method) node as a result of an action related to the VP. For example, an AuctionEntry object is added to the `_auclist` field as part of the add auction concern in Section II.

**Connector Node (C):** Any method or field that structurally connects two identified concern nodes (actions, triggers, or result nodes) in the program structure graph with a similar, but not identical, VP to the concern’s VP. These nodes do not perform the action, but communicate data and information about the action, enable the action to execute, or support the action. The difference between a connector field and a result field is that the purpose of a connector field is to communicate between different parts of the concern (for example, by connecting a trigger with an action node) and supporting the action’s execution in some way, whereas a result field typically stores the results of the action’s execution or is otherwise a side effect of the action.

Most of the action nodes in Figure 1 are easily determined from their names alone, but some of the other roles are not as obvious. For example, the action node `registerAuction` in `AuctionServer` is responsible for updating the list of auctions managed by the internal auction server (`_auclist`), and very clearly contains the phrase “add auction” in its method body. However, not all methods with a strong relationship to the verb and direct object qualify as action nodes; the indirect object needs to be taken into account as well. For example, the method `addAuction` in the class `MultiSnipe` is not adding an auction to the system, but adding an existing auction as part of a special multi-snipe bid process. However, some method names have little relationship to the concern’s VP. For example, `MQFactor.getConcrete` and `MessageQueue.enqueue` process the generic message queue that connects all user-triggered events to their implementations. Although they are general methods shared by multiple concerns, the add auction concern would not be able to execute without them.

#### IV. PRELIMINARY STUDY: USING ROLES IN PRACTICE

We hypothesize that poor agreement when annotating concerns may be due to annotators favoring certain roles over others. This led us to the following research question:

**Research Question:** *Do individual annotators tend to include or exclude certain roles over others?*

To investigate this question, we used annotated concerns from a prior study to compare against [2]. Each concern was annotated by 3 different developers. We selected three concerns from two different programs that had strong verb phrases in the concern descriptions given to the annotators: “update auction upon user-trigger,” “zoom mind map in and out,” and “toggle folded node state.” Two of the authors independently annotated the concerns and met to agree on the roles of each method and field. Finally, we applied these roles to the concern annotations created by the 3 subjects. In our analysis, we identified two observations, described below.

**Observation 1:** *Action-oriented role definitions capture the majority of program elements that developers annotate.*

When annotating concerns without a clear VP definition, the agreement among annotators has been dismally low, with average agreement only 34% [2]. After annotating three concerns ourselves and comparing our annotations with developer-generated annotations, we determined that our role annotations contained the majority of program elements that the developers included. For instance, in the toggle folded concern, 43 of the 50 elements in our mapping were contained in at least one developer’s mapping—an 86% intersection rate. Similarly, in the update auction concern, 32 of 42 elements intersected (76%). We attribute these unusually high intersection rates to the ability of our concern definitions to comprehensively define the roles relevant to action-oriented concerns.

In the last concern, we saw much less intersection between our annotations and the developers’, with only 21 of 35 (60%) intersecting. This low intersection rate is explained by examining the non-intersecting nodes. For this concern, developers included six nodes that initialize the trigger code. Recall from Section III that we do not consider trigger initialization code as part of the concern, because they only execute during startup and not during the action’s execution. For this concern, developers also included two incorrect triggers, code related to zooming as part of the fit-to-page action, which was intentionally omitted from the concern description. Not considering these initialization nodes and incorrect triggers increases the intersection rate to 82%. In our future work, we will explore whether initialization nodes should be included as part of the role definitions. Although developers included a number of these extra nodes, there were only 3 elements that we included that the annotators missed (91% agreement).

We also found that the role definitions possess inherently good explanatory power. For example, the zoom concern contains two methods, `update` and `updateAll`, which seemingly are outside of the zoom concern. In fact, only one developer identified them as relevant. When producing our annotations, we included both methods because they are connector nodes leading to the action node `setZoom` and result node `zoomFactor`. Thus, roles can help us reason about why a developer may include certain nodes in their annotations.

**Observation 2:** *Differences in annotations are due to control flow islands within the action-oriented concerns.*

When analyzing developers’ concern mappings, we noticed that they were often composed of distinct “islands” where there were no control flow connections or the connections were difficult for a developer to recognize. These islands tended to be organized around trigger nodes or action nodes, with connectors occurring in either type of island, and result nodes predominantly in action-oriented islands (as opposed to trigger-oriented). Some islands contained both trigger and action nodes, if there were strong control flow relationships and method names.

The update auction concern offers a clear example of these islands, with our final annotation comprising three distinct

call graphs. The first island contains user triggers and a flag-setting method, the second contains a timer process to check the flag periodically, and the third the actual UI updating call-chain. These islands have two clear effects on a developers' ability to annotate a concern. First, if developers can find a single element in an island, they typically include multiple elements from that island. For example, in the update auction concern, all subjects found two of the three islands, and all subjects found the second island, which contains 8 elements and includes both action and trigger nodes. However, the annotators found different numbers of elements within those islands: subject P13 found 2/4 and 3/8 elements, P10 found 3/4 and 7/8 elements, and P12 found 7/8 and 3/5 elements. This example illustrates that developers are adept at following clear control flow edges and expanding from a starting point in the code. Second, we observed that even if developers found a significant portion of the overall concern, they were likely to omit *an entire* island, with each developer omitting at least one of the three islands in this concern. For example, P13 found 0/5 in the third island, P10 found 0/5 in the third island, and P12 found 0/4 in the first island. Both the other concerns we studied contain similar islands. For instance, the toggle folded concern contains an island mainly composed of the `UnfoldAll` class and methods that no participants included because it had no direct control flow links to identified elements. Similarly, the zoom concern contained an island with elements in the `MultipleImage` class. The developers that skipped `NodeView`'s `update` and `updateAll` methods missed the control flow links to the `MultipleImage` island.

In terms of navigating to islands, developers seemed less likely to follow data flow between fields connecting islands, or control flow where the textual clues are poor. Developers especially had trouble finding methods that implemented interfaces that were called by other parts of the concern. For example, in the "toggle folded node state" concern, annotators found portions of the `ToggleFoldedAction` and `ControllerAdapter` island, but completely missed the remaining 3 islands. Both the unfound islands `MindMapNode` and `FoldActionType` contain methods called directly by the found island, but developers seemed to have trouble recognizing the relevance of the interface methods, and did not locate the implementing methods. The `UnfoldAll` island is largely composed of triggers and connectors, and would be missed if the call graph were not explored from `ControllerAdapter` in the found island.

## V. RELATED WORK

Koenemann, et al. studied developers comprehending code for maintenance, and differentiated between 3 tiers of relevance: direct (must be modified), intermediate (studied if interaction with relevant code is important), and strategic (guide comprehension process; points to relevant code) [5]. These relevance tiers are orthogonal to concern element roles.

In a study of three hand-annotated concerns related to specific change tasks, Murphy, et al. observed that concern boundaries may be difficult to determine [4]. Based on interviews with annotators, the authors observed that the interface

between concerns can be a concern. This is in contrast to our notion of triggers and connectors, which we consider to be part of a concern when specified by a precise verb phrase, rather than a change task. The authors conclude that concerns have 3 different types of elements: core behavior, a potentially ambiguous interface that may be a concern in its own right, and a set of execution points that hook into where new functionality may be added to the concern during maintenance. Although the notion of core nodes is similar to our action nodes, the notions of interfaces and execution hooks are different. As previously mentioned, interface methods are part of our trigger and connecting nodes. Because the focus of this work is on annotating existing functionality, we would consider execution hooks to belong to their own concern.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a characterization of the roles program elements play in action-oriented concerns. We identified four basic concern element roles: action, trigger, result, and connector nodes. We used these role annotation nodes to study formerly annotated concerns with low agreement, finding that the roles help explain the differences between annotations. We believe the preliminary results show promise for improving recommendation and exploration tools by focusing on retrieving the types of nodes and connections that developers seem to have trouble identifying, and that are not currently well-supported in state of the art IDEs. We plan to use the notion of action and trigger islands to study the agreement in the remaining concerns from the prior study [2], and investigate whether trigger initialization code should be added as a concern role type. We hypothesize that roles may also help explain differences in existing code recommendation and exploration tools.

## REFERENCES

- [1] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *TOSEM*, vol. 16, no. 1, p. 3, 2007.
- [2] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, "An empirical study of the concept assignment problem," School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3, Jun. 2007, <http://www.cs.mcgill.ca/~martin/concerns/>.
- [3] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD*, 2007.
- [4] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong, "Design recommendations for concern elaboration tools," in *Aspect-Oriented Software Development*, T. Elrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, 2004, pp. 507–530.
- [5] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *CHI*, 1991.
- [6] N. Dragan, M. Collard, and J. Maletic, "Reverse engineering method stereotypes," in *ICSM*, 2006.
- [7] M. Reville, T. Broadbent, and D. Coppit, "Understanding concerns in software: Insights gained from two case studies," in *IWPC*, 2005.
- [8] J. Marie Burkhardt, F. D tienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Soft. Eng.*, vol. 7, no. 2, 2002.
- [9] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *IWPC*, 2002.

# Theory and Practice, Do They Match?

## A Case With Spectrum-Based Fault Localization

Tien-Duy B. Le, Ferdian Thung, and David Lo  
 School of Information Systems  
 Singapore Management University, Singapore  
 {btdle.2012,ferdianthung,davidlo}@smu.edu.sg

**Abstract**—Spectrum-based fault localization refers to the process of identifying program units that are buggy from two sets of execution traces: normal traces and faulty traces. These approaches use statistical formulas to measure the suspiciousness of program units based on the execution traces. There have been many spectrum-based fault localization approaches proposing various formulas in the literature. Two of the best performing and well-known ones are Tarantula and Ochiai. Recently, Xie et al. [18] find that *theoretically*, under certain assumptions, two families of spectrum-based fault localization formulas outperform all other formulas including those of Tarantula and Ochiai. In this work, we empirically validate Xie et al.’s findings by comparing the performance of the *theoretically best* formulas against popular approaches on a dataset containing 199 buggy versions of 10 programs. Our empirical study finds that Ochiai and Tarantula statistically significantly outperforms 3 out of 5 theoretically best fault localization techniques. For the remaining two, Ochiai also outperforms them, albeit not statistically significantly. This happens because an assumption in Xie et al.’s work is not satisfied in many fault localization settings.

### I. INTRODUCTION

In software systems, bugs are unavoidable. Many bugs are regularly found and reported to the developers. The amount of bugs to be fixed is often much larger compared to the size of the development team [3]. To tackle this problem, researchers have developed automated approaches to help developers in fixing bugs. These automated approaches include the many fault localization techniques proposed in the literature [15], [9], [1], [19], [13], [14]. The goal of fault localization is to localize a bug to local regions of the source code. Thus, rather than the whole program, developers only need to investigate a much smaller part of the program. This would significantly reduce the amount of time needed to find the buggy program elements and fix the bug.

One large family of fault localization techniques is Spectrum-Based Fault Localization (SBFL) techniques [15], [9], [1], [19], [13]. SBFL techniques analyze program spectra, which are program traces collected during the execution of a program, to correlate failures (i.e., faulty execution traces) with program elements (e.g., lines, basic blocks) that are responsible for them. Various SBFL techniques use various formulas to assign suspiciousness scores to program elements. Program elements are then ranked based on their suspiciousness scores. The resulting ranked list is then given to developers to help them find the root cause of failures. Two well-known SBFL techniques are Tarantula [9] and Ochiai [1].

Recently, Xie et al. [18] have theoretically investigated many SBFL formulas. Their study has shown that SBFL formulas can be grouped into families (or equivalence classes). Within each family, the formulas have the same effectiveness to localize bugs under certain assumptions. Also they have created a partial order which shows which families of SBFL formulas are better than others. At the top of the partial order are 2 families of SBFL formulas named ER1 and ER5 which contain in total 5 SBFL formulas. Xie et al. have *theoretically* proven that the 5 SBFL formulas can outperform Tarantula’s and Ochiai’s SBFL formulas. However, these SBFL formulas have not been *empirically* compared with one another on actual failures and programs.

In this study, we want to inspect the applicability of the *theoretically best* SBFL formulas to localize faults in standard SBFL benchmark dataset. Xie et al. theoretical analysis assumes that the test coverage level is 100%. This assumption is likely not to hold for many fault localization settings. Thus, there is a need for an empirical study to demonstrate whether these *theoretically best* formulas could outperform popular formulas in many fault localization settings.

In this empirical study, we use 199 buggy versions of 10 programs: NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [8]. We want to answer the following research questions:

- RQ 1 How effective are the popular and theoretically best SBFL formulas?
- RQ 2 Could the theoretically best SBFL formulas outperform the popular formulas?
- RQ 3 Is the assumption considered by Xie et al. [18] satisfied in many fault localization settings?

Our empirical study demonstrates that among the 7 SBFL formulas (5 theoretically best and 2 popular formulas), Ochiai’s SBFL formula performs the best. Using it, on average, developers only need to inspect 21.02% of the source code. Among the theoretically best formulas, the best percentage is 21.09%. According to the Wilcoxon signed rank test [16], Ochiai’s SBFL formula statistically significantly outperforms 3 out of the 5 theoretically best SBFL formulas.

The following are our contributions:

- 1) We empirically evaluate the effectiveness of the *theoretically best* SBFL formulas against popular ones (i.e.,

TABLE I  
RAW STATISTICS FOR SBFL

	$e$ Executed	$e$ Not Executed
Test Passed	$n_s(e)$	$n_s(\bar{e})$
Test Failed	$n_f(e)$	$n_f(\bar{e})$

Tarantula’s and Ochiai’s SBFL formulas). We find that Ochiai’s SBFL formula statistically significantly outperforms 3 out of the 5 theoretically best SBFL formulas. For the remaining two, Ochiai’s SBFL formula performs better, although the differences are not statistically significant.

- 2) We highlight that the assumption made by Xie et al. in their theoretical analysis, that the code coverage level is 100%, is not satisfied in many fault localization settings which affects the performance of the *theoretically best* SBFL formulas.

The following is the structure of the paper. In Section II, we introduce SBFL and highlight Ochiai, Tarantula, and the two families of theoretically best SBFL formulas [18]. In Section III, we describe our empirical study methodology. In Section IV, we present the answers to the research questions. We discuss related work in Section V. We conclude and mention future work in Section VI.

## II. BACKGROUND

In this section, we first succinctly introduce SBFL. Next, we describe the formulas used in two popular approaches namely Tarantula [9] and Ochiai [1]. We then highlight the formulas demonstrated by Xie et al. [18] to be theoretically the best.

### A. SBFL in a Nutshell

SBFL is a technique to localize a bug to certain parts of the program by utilizing program spectra collected from software testing and the result of the tests (*pass* or *fail*). Program spectra, which is a record of which program elements are executed for each test case, can be collected at different levels of granularity (e.g., lines, basic blocks, methods, components, etc.). In this paper, we consider the basic block level granularity (i.e., each basic block is a program element). SBFL requires a set of test cases where at least one of the test cases results in a faulty execution (i.e., the test case fails). For each program element  $e$ , SBFL computes the raw statistics shown in Table I.

The notation  $\bar{e}$  means  $e$  is not executed,  $n_s(e)$  denotes the number of successful test cases that execute  $e$ ,  $n_f(e)$  denotes the number of failing test cases that execute  $e$ ,  $n_s(\bar{e})$  denotes the number of successful test cases that do not execute  $e$ , and  $n_f(\bar{e})$  denotes the number of failing test cases that do not execute  $e$ .

Based on the statistics, a suspiciousness score for each program element  $e$  is computed. The higher the score is, the more suspicious the program element is. Thus, a ranked list of program elements sorted by their suspiciousness scores is returned. The list can then be investigated by developers, starting from the most suspicious program element.

### B. Popular Approaches: Tarantula and Ochiai

Many approaches have been proposed to compute the suspiciousness scores of program elements [9], [1], [13], [18]. Tarantula [9] and Ochiai [1] are among the most popular approaches. Using the notations in Table I, Tarantula’s SBFL formula, which assigns a suspiciousness score to a program element  $e$ , is defined as follows:

$$Tarantula(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_s(e)}{n_s} + \frac{n_f(e)}{n_f}}$$

where  $n_f = n_f(e) + n_f(\bar{e})$  and  $n_s = n_s(e) + n_s(\bar{e})$ .

Ochiai’s SBFL formula is defined as follows:

$$Ochiai(e) = \frac{n_f(e)}{\sqrt{n_f n_f(e) + n_s(e)}}$$

### C. Theoretically Best SBFL Formulas

Xie et al. [18] have compared 30 SBFL formulas and theoretically prove that two families of SBFL formulas outperform others, including those of popular approaches like Tarantula and Ochiai. They refer to these two families as  $ER1$  and  $ER5$ .  $ER1$  has two members:  $ER1^a$  and  $ER1^b$ .  $ER5$  has three members:  $ER5^a$ ,  $ER5^b$ , and  $ER5^c$ . Using the notations in Table I, the following are the definitions of those formulas which assign a suspiciousness score to a program element  $e$ :

$$ER1^a(e) = \begin{cases} -1, & \text{if } n_f(e) < n_f \\ n_s - n_s(e), & \text{if } n_f(e) = n_f \end{cases}$$

$$ER1^b(e) = n_f(e) - \frac{n_s(e)}{n_s(e) + n_s(\bar{e}) + 1}$$

$$ER5^a(e) = n_f(e)$$

$$ER5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_s(e) + n_s(\bar{e})}$$

$$ER5^c(e) = \begin{cases} 0, & \text{if } n_f(e) < n_f \\ 1, & \text{if } n_f(e) = n_f \end{cases}$$

## III. METHODOLOGY

In this section, we first describe the dataset that we use to investigate the effectiveness of SBFL approaches. Next, we describe how we collect traces from this dataset. We then describe how we measure effectiveness.

### A. Dataset

Our dataset consists of buggy versions of 10 programs: NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [8]. NanoXML is a Java library for XML parsing. XML-Security is a Java library for encryption and digital signature. Space is an Array Definition Language (ADL) interpreter written in C. Siemens test suite is a suite created by Siemens for research in test coverage adequacy. NanoXML, XML-Security, and Space are downloaded from the Software Infrastructure Repository (SIR) [5]. For NanoXML and XML-Security, we exclude faulty versions that do not have failing

TABLE II  
DATASET DESCRIPTIONS: NAME, LINES OF CODE, PROG. LANGUAGE,  
NUMBER OF FAULTY VERSIONS, AND NUMBER OF TEST CASES.

Dataset	LOC	Language	# Faulty	# Tests
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6,218	C	35	13,585
NanoXML v1	3,497	Java	6	214
NanoXML v2	4,007	Java	7	214
NanoXML v3	4,608	Java	8	216
NanoXML v5	4,782	Java	8	216
XML security v1	21,613	Java	6	92
XML security v2	22,318	Java	6	94
XML security v3	19,895	Java	4	84

test cases or the faulty program elements are not executed by any test case. For Siemens test suite, we exclude versions where the fault is located in the variable declaration. This is done since our instrumentation cannot reach it. Each faulty version contains one bug that may span across multiple program elements (i.e. basic blocks). Table II shows some statistics of our dataset. These programs and buggy versions have been used as a benchmark dataset used to evaluate many past SBFL studies [1], [9], [15], [12], [13], [10].

#### B. Collecting Execution Traces

Execution traces are collected at the basic block level. Each basic block in a program is instrumented by a “print” statement such that we can detect whether a particular basic block is executed when a test is run. For each basic block, a matrix shown in Table I is maintained. Based on the status of a test (i.e., pass or fail) and whether a basic block is executed when the test is run, the corresponding element in the matrix is updated. For example, if the test fails and basic block A is executed, then  $n_f(A)$  is increased by 1. The matrices for the other basic blocks in the program are also updated accordingly. This process is repeated for each test. In the end, each basic block will have a matrix reflecting its execution profile.

#### C. Measuring Effectiveness of SBFL Approaches

In order to evaluate the effectiveness of a SBFL formula, we count the percentage of executable code that needs to be inspected to reach the first faulty program element. In the case that many program elements share the same suspiciousness score with the faulty program element, we assign the worst rank to the faulty program element (i.e. the faulty program element has the largest rank among all program elements with the same score). This measure is referred to as the *EXAM* score [17]. The following is the formula for calculating the *EXAM* score:

$$EXAM \text{ score} = \frac{\text{Rank of the first faulty program element}}{\text{Total number of executable program elements}}$$

The lower the *EXAM* score, the better is the performance of a SBFL formula. To illustrate *EXAM* score computation, consider four program elements  $e_1$ ,  $e_2$ ,  $e_3$ , and  $e_4$  in a

TABLE III  
EFFECTIVENESS OF THE SBFL FORMULAS

Technique	Average % Inspected	Standard Dev.
Tarantula	23.37%	23.44%
Ochiai	21.02%	21.96%
$ER1^a$	33.34%	35.22%
$ER1^b$	21.09%	19.48%
$ER5^a$	43.04%	19.63%
$ER5^b$	43.04%	19.63%
$ER5^c$	54.95%	26.83%

program with suspiciousness scores of 1.0, 0.75, 0.75, and 0.5 respectively. Assuming that  $e_2$  is the faulty program element, in the worst case, developers need to inspect 3 program elements to reach the faulty program element. Thus, the *EXAM* score for this example is  $\frac{3}{4} = 75\%$ .

## IV. EXPERIMENTS & ANALYSIS

In this section, based on the methodology described in Section III, we describe the answers to the 3 research questions that we listed in Section I.

#### A. RQ1: Effectiveness of SBFL Formulas

The effectiveness of the various SBFL formulas are shown in Table III. The average percentage of program elements to be inspected to find the first faulty program element are 23.37%, 21.02%, 33.34%, 21.09%, 43.04%, 43.04%, and 54.95% for Tarantula, Ochiai,  $ER1^a$ ,  $ER1^b$ ,  $ER5^a$ ,  $ER5^b$ , and  $ER5^c$  respectively.

#### B. RQ2: Popular vs. Best Approaches

From Table III, we notice that Ochiai has the lowest *EXAM* score. The *EXAM* score of Tarantula is also lower than 4 out of the 5 *theoretically best* SBFL formulas. We have also performed non-parametric statistical tests, i.e., Wilcoxon signed rank tests [16], with a significance level of 0.05. We find that the *EXAM* scores of Ochiai are *statistically significantly* better than those of  $ER5^a$ ,  $ER5^b$ ,  $ER5^c$ .

#### C. RQ3: Validity of Assumptions

We have also investigated the reason why the theoretically best SBFL formulas cannot outperform popular techniques despite the theoretical analysis given in [18]. We investigate the code coverage of the buggy versions of the 10 programs. We find that out of the 199 buggy versions, for 135 of them, the code coverage is not 100%. The average code coverage for the 199 buggy versions is 84.97%. This highlights the reason why the theoretical findings in [18] does not hold for many fault localization settings.

#### D. Threats to Validity

Threats to internal validity refers to errors or experimental bias. We have double checked our code and implementation of the formulas. Still there could be errors that we do not notice.

Threats to external validity refers to the generalizability of our findings. We have analyzed 199 buggy versions from 10 programs. These programs and buggy versions have been used to evaluate many past SBFL studies [1], [9], [15], [12],

[13], [10]. In the future, we plan to reduce this threat to validity further by investigating more bugs from more software systems.

Threats to construct validity refers to the suitability of our evaluation measure. We have used the *EXAM* score which is used to evaluate many past SBFL studies [17], [1]. The study by Xie et al. [18] also theoretically compares the performance of many SBFL formulas using the *EXAM* score.

## V. RELATED WORK

In the following paragraphs, we first highlight some SBFL studies. Next, we also briefly discuss other fault localization approaches that do not rely on program spectrum. Due to the space constraint, the survey here is by no means complete.

**SBFL.** Many SBFL approaches have been proposed in the literature [15], [9], [1], [19], [13]. All these techniques analyze program spectra which are logs of execution traces generated when a target program is run. Zeller proposes a technique named Delta Debugging which finds the minimum state difference that causes a failure to be generated [2]. Renieris and Reiss compare a faulty execution with the nearest correct execution to find suspicious program elements [15]. Jones and Harrold propose an SBFL technique named Tarantula which uses a formula to compute suspiciousness of program elements based on the assumptions that program elements executed more by faulty executions rather than by correct executions are more likely to be faulty [9]. Abreu et al. propose another SBFL technique named Ochiai that uses another formula to compute suspiciousness of program elements [1]. Lucia et al. investigate the effectiveness of many association measures for fault localization [13]. Gong et al. propose an interactive SBFL approach that takes incremental user input into consideration [7]. Gong et al. also propose another SBFL approach that reduces the number of test cases with oracles [6]. Cheng et al. mine graph-based signatures that highlight suspicious program elements by analyzing program spectra [4]. Duy and Lo propose a classification-based approach that predicts whether an SBFL technique would be effective for a particular fault localization task [11]. Xie et al. theoretically analyze many SBFL formulas including Tarantula and Ochiai and show that two families of SBFL formulas (ER1 and ER5) could outperform the others if a number of assumptions hold [18]. In this work, we compare the effectiveness of the *theoretically best* formulas presented in Xie et al.'s work with Tarantula and Ochiai using a standard SBFL benchmark dataset.

**Other Fault Localization Approaches.** Aside from SBFL, a number of past papers have also proposed model-based fault localization techniques which often use formal models and employ expensive logic reasoning, e.g., [14]. This limits the applicability of this family of fault localization approaches especially on large complicated programs. In this work, we only consider SBFL approaches.

## VI. CONCLUSION AND FUTURE WORK

We have conducted an empirical evaluation of various SBFL techniques on 199 buggy versions of NanoXML, XML-

Security, Space, and the 7 programs from the Siemens test suite. We compare the performance of 5 *theoretically best* SBFL formulas presented by Xie et al. [18] with popular SBFL formulas (Tarantula and Ochiai). We find that Ochiai's SBFL formula outperforms all, while Tarantula's SBFL formula outperforms four *theoretically best* SBFL formulas. For three out of the five *theoretically best* formulas, Ochiai and Tarantula SBFL formulas *statistically significantly* outperform them. We highlight that the assumption made by Xie et al. is not valid for many settings. For many programs, even though with a large number of test cases, the code coverage is not 100%. A relatively small reduction in test coverage can significantly affect the performance of the *theoretically best* SBFL formulas.

As a future work, we plan to perform a more in-depth study on how coverage levels and other factors affect the effectiveness of various SBFL formulas. We are also interested in theoretically analyzing the performance of SBFL formulas under a more relaxed assumption (i.e., less than 100% coverage). Furthermore, we want to reduce the threat to external validity by investigating more programs and buggy versions.

## REFERENCES

- [1] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.
- [2] Andreas Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT FSE*, 1999, pp. 253–267.
- [3] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, 2005, pp. 35–39.
- [4] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009.
- [5] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.
- [6] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *ASE*, 2012.
- [7] —, "Interactive fault localization leveraging simple user feedback," in *ICSM*, 2012.
- [8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of ICSE*, 1994, pp. 191–200.
- [9] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.
- [10] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *PLDI*, 2011, pp. 437–446.
- [11] T.-D. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *ICSM*, 2013.
- [12] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005.
- [13] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
- [14] W. Mayer and M. Stumptner, "Model-Based Debugging – State of the Art And Future Challenges," *ENTCS*, vol. 174, no. 4, 2007.
- [15] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 141–154.
- [16] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
- [17] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 42, no. 3, pp. 378–396, 2012.
- [18] X. Xie, T. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *TOSEM (to appear)*, 2013.
- [19] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.

## An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings

Quinten David Soetens  
*University of Antwerp*  
*Antwerp, Belgium*

Javier Perez  
*University of Antwerp*  
*Antwerp, Belgium*

Serge Demeyer  
*University of Antwerp*  
*Antwerp, Belgium*

**Abstract**—Today, it is widely accepted that if refactoring is applied in practice, it is mainly interweaved with normal software development — so called “floss refactoring”. Unfortunately, the current state-of-the-art is poorly equipped to mine floss refactorings from version histories, mainly because they infer refactorings by comparing two snapshots of a system and making educated guesses about the precise edit operations applied in between. In this paper we propose a solution that reconstructs refactorings not on snapshots of a system but using the actual changes as they are performed in an integrated development environment. We compare our solution against RefFinder and demonstrate that on a small yet representative program (the well-known “VideoRental system”) our approach is more accurate in identifying occurrences of the “MOVEMETHOD” and “RENAMEMETHOD” refactorings.

### I. INTRODUCTION

Refactoring is widely recognized as a crucial technique applied when evolving object-oriented software systems. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [1].

Several researchers set-out to reconstruct refactorings as they occurred in software projects. Initially, this was mainly an act of scientific curiosity (i.e. [2], [3], [4], [5]), however later on actual applications emerged. Weißgerber for instance used them as a means for studying the impact of refactorings on defects [6]. Dig. et al. prototyped a capture-playback tool capable of replaying refactorings when migrating systems dependent on a refactored API [7], [8]. Obviously, several authors tried to correlate the impact of refactorings on the maintainability of a software project [9], [10], [11], [12].

In the meantime, several field studies and surveys indicated that if refactoring is applied in practice, it is mainly interweaved with normal software development [13], [14]. This phenomenon is what is commonly referred to as “floss refactoring”: frequent short bursts of refactorings interspersed with normal editing right before or after a code smell is introduced [15]. Consequently, current refactoring reconstruction techniques will miss a significant portion of the actual refactorings, because they infer refactorings by comparing two snapshots of a system and making educated guesses about the precise edit operations applied in between.

In this paper we investigate whether access to the actual changes as performed in an integrated development environment can improve refactoring reconstruction. In particular, we want to address the following research questions.

**RQ – Feasibility.** *Is it possible to reconstruct refactorings from a stream of fine-grained changes ?*

**RQ – Comparison.** *How does such a change-based approach compare to a snapshot-based approach with regard to floss refactoring ?*

We performed a proof by construction via a tool prototype named ChEOPJS, which sits in the background of Eclipse and records the changes made to a software system while a developer is programming. The changes recorded by ChEOPJS are interconnected with dependencies; for example a change that adds a method is dependent of the change that adds the class in which the method is contained. As such, we get a graph of dependent changes which can be searched for pre-defined patterns representing refactorings. We compare our solution against RefFinder and demonstrate that on a small yet representative program (the well-known VideoRental system [1, Chapter 1]) our approach is more accurate in identifying occurrences of the MOVEMETHOD and RENAMEMETHOD refactorings.

### II. FIRST CLASS CHANGE OBJECTS

We define *Change* as an object representing an action that changes a software system. As such, a change becomes a tangible entity that we can analyze and manipulate. This idea of representing changes as tangible objects has already been implemented and explored in a number of tools and studies. Omori and Maruyama created tools called OperationRecorder and OperationReplayer, which is capable of recording and replaying operations performed in the Eclipse IDE [16]. The approach by Robbes et al. in the Spyware tool [17] and later by Hattori et al. in the Syde tool [18] model changes as operations on the Abstract Syntax Tree.

We opted to adopt the approach made by Ebraert et al., as this change model also includes dependencies between the changes [19]. Where Ebraert et al. made creative use of Smalltalk’s internal change list, our tool ChEOPJS implements a Java version of his model in Eclipse [20], [21]. Our tool sits in the back of Eclipse and captures all changes made

in the main editor while the developer is programming. In this model we defined three kinds of *Atomic Changes*: Add, Modify and Remove. These changes act upon a *Subject* representing an actual source code entity. For these subjects we chose to use the FAMIX model [22], as this is a model to which most object oriented programming languages adhere. It defines entities representing, packages, classes, methods and attributes, as well as more fine grained entities such as method invocations, variable accesses and inheritance relationships. Although we can use any model that is capable of representing source code entities.

We define dependencies between changes as follows. A change object  $c_1$  is said to depend on another change object  $c_2$  if the application of  $c_1$  without  $c_2$  would violate the system invariants. For instance, an addition of a method depends on the addition of a class. As such a set of changes becomes interconnected through a series of dependencies.

We can therefore represent the changes made to a software system as a graph  $G = (V, E)$ . The nodes ( $V$ ) then represent both the first class change entities (additions represented by the  $\oplus$  icon and removals represented by the  $\otimes$  icon) as well as FAMIX entities upon which those changes act.

The set of edges  $E$  consists of two types of edges: an edge between two changes represents a dependency relationship between those changes (in Figure 3 represented as a dashed red arrow); an edge between a change and a FAMIX entity says that this entity is the subject of the change (in Figure 3 represented as a dashed black arrow).

### III. CHANGE-BASED REFACTORING RECONSTRUCTION

Once we have the sequence of changes and their dependencies in place, we use Groove [23], a graph transformation tool, to search the change graph for pre-defined patterns corresponding to a refactoring. A graph transformation rule typically consists of a lefthand side (LHS) and a righthand side (RHS). In Groove the LHS and RHS are combined into one single view and color coding is used to distinguish between the different elements. We use the LHS to define the graph pattern representing a refactoring operation and we use the RHS to introduce a new type of nodes, representing the identified refactoring instances. For this initial investigation, we limited ourselves to the two most frequently occurring floss refactorings [13]: the MOVEMETHOD refactoring and the RENAMEMETHOD refactoring. As an example we detail how the MOVEMETHOD is defined. The RENAMEMETHOD can be defined in a similar way.

Our pattern definition for the MOVEMETHOD refactoring is shown in Figure 1. It consists of two class nodes and two method nodes and four addition nodes and a removal node. The two additions linked to the method nodes are dependent of the additions linked to the class nodes, meaning that these methods are members of those classes. The class nodes should not be equal, indicating that it should be two different classes. The two methods have the same

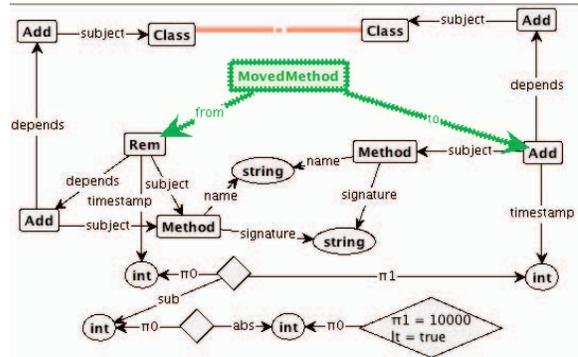


Figure 1: The MOVEMETHOD pattern in Groove

signature and the same name. One of the methods also has a remove change linked to it. We then check that the difference between the timestamps of the remove change and the change that added the other method is less than 10 seconds (10000 milliseconds). We take 10 seconds to take into account manual refactorings using copy and paste operations. In our experience this threshold worked, however further experiments are needed in order to validate this time threshold. If this pattern can be matched on the change graph, we have identified the MoveMethod refactoring. In the RHS of the graph transformation we added a node representing this refactoring by connecting it to the change that removed the original instance of the method and to the change that added the new instance of the method.

### IV. SNAPSHOT-BASED REFACTORING RECONSTRUCTION

The state of the art in refactoring reconstruction consists of analyses of the snapshots maintained in a source code repository. Most approaches use some kind of code similarity measure to identify possible refactoring candidates. Danny Dig et al. as well as Weißgerber et al. used a combination of a signature based analysis and shingles (a form of hashing) [6], [8]. Van Rysselberghe and Demeyer use clone detection on two versions to look for a decrease in the number of clones, which would suggest that a refactoring was performed, since many refactorings are aimed at the elimination of duplicated code [4].

There are some approaches that do not use clone detection or similarity metrics. For instance Demeyer et al. developed a set of heuristics to identify refactorings using source code metrics [2]. Xing and Stroulia search for refactorings at the design level using their UMLDiff algorithm, which can detect some basic structural changes to the system [5].

Ref-Finder, an eclipse plugin by Kim et al., is to date the most comprehensive refactoring reconstruction tool as it supports 63 different types of refactorings [11]. They use the technique proposed by Prete et al. which is stronger than all previous techniques because they not only detect primitive refactorings (which all previous techniques can do to some extent) but also “complex refactorings” (i.e. refactorings which are combinations of primitive refactorings). They use

a fact base with a Tyruba logic query engine [3].

## V. COMPARISON AGAINST REFFINDER

We compare our approach to the RefFinder tool on the VideoRental system as described in [1, Chapter 1]. The class diagram of this system is shown in Figure 2. It consists of three classes: Movie, Rental and Customer. The program’s main functionality is to calculate and print a customer’s charges at a video store. It keeps track of which movies are rented by a customer and for how long. It can then calculate the charges that depends on the type of movie and on how long the movie was rented.

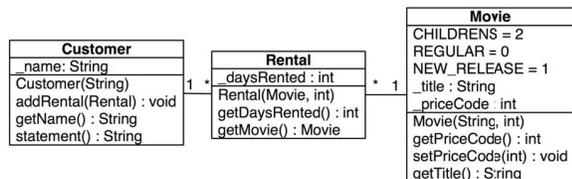


Figure 2: Version 1 of the VideoRental system.

We recreated this system with the change logger of ChEOPJS running in the background, thus obtaining the first class changes representing the VideoRental system. We also stored the different versions of this system on an svn repository. The system as described in Figure 2 is called Version 1. We then proceed in refactoring the system and store a version of the system after each atomic refactoring operation. In short Table I shows which refactorings were performed in between two subsequent versions.

Table I: Refactorings performed on the VideoRental system.

Versions	Refactoring Performed
V1 - V1.1	ExtractMethod
V1.1 - V1.2	MoveMethod
V1.2 - V1.3	RenameMethod
V1.3 - V2	RemoveParameter
V2 - V2.1	ExtractMethod
V2.1 - V2.2	NONE
V2.2 - V2.3	MoveMethod
V2.3 - V3	RemoveParameter

Figure 3 shows the change graph that we obtained before and after running our graph transformation rules. The top section of this figure shows a part of the change graph that we obtained by logging the changes made to the VideoRental system. We only show that part of the graph that is relevant for our refactoring rules. After extracting the change graph and importing it into Groove we could use our graph transformation rules to add the nodes in the bottom section of Figures 3. We successfully managed to reconstruct the MOVEMETHOD and RENAMEMETHOD refactorings that were applied to go from Version 1 of the code to Version 3.

This effectively answers **RQ – Feasibility**, showing that it is indeed possible to reconstruct refactorings from a graph of changes using graph pattern matching to identify specific refactoring patterns. Moreover the use of graph transformation engines for graph pattern matching is a simple and

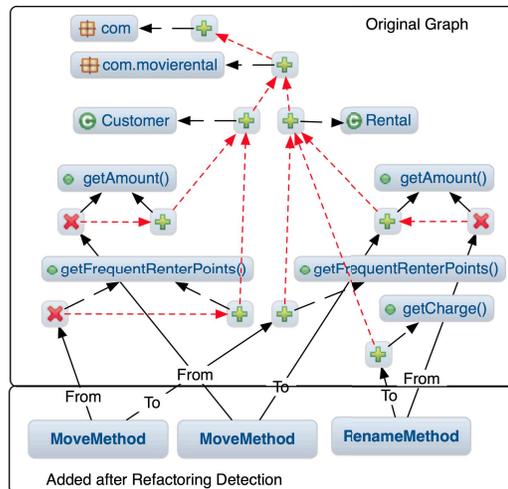


Figure 3: The change graph after logging the development of the VideoRental system (top) and the nodes added after applying the refactoring reconstruction rules (bottom).

natural choice when dealing with a model that can be represented as a graph.

We then also ran RefFinder on each subsequent version of the VideoRental system and find that RefFinder does indeed successfully reconstructs all performed refactorings. However when no intermediate versions would have been stored after each refactoring and we only have Versions V1, V2 and V3 in the repository, then RefFinder is unable to reconstruct the performed refactoring operations. It did however identify a HIDEDELEGATE refactoring.

This answers **RQ – Comparison**, RefFinder can successfully reconstruct the refactorings when they are stored in separate revisions. However in the face of floss refactoring RefFinder is unable to reconstruct the refactorings, as one refactoring can hide behind another, making it impossible to find either one. Our change-based approach on the other hand does not suffer from this problem, since entities that are removed are still represented in the system making it easier to reconstruct the actual refactorings that were applied. We can thus conclude that a change-based approach is more accurate in identifying occurrences of the “MOVEMETHOD” and “RENAMEMETHOD” refactorings than a snapshot-based approach.

## VI. LIMITATIONS AND FUTURE WORK

The main drawback of this technique is that it needs a live-recorded change history. Therefore, developers should install the change-recording plugin and agree to have its source code edits logged. The heuristic nature of the refactoring detection rules could also imply some percentage of false positives and negatives. Experiments with real data have to be run in order to precisely quantify this. We should also investigate further on finding out if there are some refactorings that could not be detected by this means.

As future work, in the short term, we will add new

detection rules for more refactorings and run experiments with real data. We are currently in contact with some companies for testing our refactoring detection proposal in real situations. ChEOPJSJ will be installed in these companies to run experiments in real development teams. The results of this experiments will help us study the precision and recall of our technique and will help us improve the detection rules, *e.g.* for fine tuning time thresholds.

Our work also raises some open questions and points to discuss. Our proposal on heuristic rules based on the refactoring specifications from the literature and on our own expertise. These rules have to be specified, then tested and improved with real data. We would like to explore alternative ways to identify refactorings, for example, by searching for frequent graph patterns in a system's change model.

Snapshot-based refactoring identification tools might detect different refactorings from the same situation, as we have mentioned at the end of Section V. We think it is worth exploring whether these two approaches combined could be used to find equivalent refactoring sequences. Moreover, it should be investigated in which situations finding the exact refactorings is needed and, in what other situations it does not matter to only find an equivalent one. Reapplying equivalent refactorings can be useful in some cases, such as in the presence of a merge conflict.

## VII. CONCLUSIONS

In this paper, we have investigated whether access to the actual changes as performed in an integrated development environment can improve the reconstruction of floss refactorings. First of all, we have demonstrated through a proof by construction that it is feasible to query a stream of changes to distinguish refactorings from ordinary program edits. For this we relied heavily on a graph-based representation of first class changes and the dependencies between them as induced by the FAMIX meta-model. Secondly, we compared our solution against RefFinder—the most comprehensive snapshot based system—and demonstrated that on a small yet representative program (the well-known “VideoRental system”) our approach is more accurate in identifying occurrences of the “MOVEMETHOD” and “RENAMEMETHOD” refactorings.

As a final remark, we point out that this paper presents yet another example of the potential of fine-grained change histories and how this may facilitate certain programming tasks. In the long term, we intend to study other benefits of and explore whether fine-grained change histories should be distributed, together with the source code.

## ACKNOWLEDGMENTS

This work has been sponsored by (i) the Interuniversity Attraction Poles Programme - Belgian State Belgian Science Policy, project MoVES; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

under project number 120028 entitled “Change-centric Quality Assurance (CHAQ)”

## REFERENCES

- [1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.
- [2] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” in *Proc. of OOPSLA'00*, 2000.
- [3] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, “Template-based reconstruction of complex refactorings,” in *Proc. ICSM'10*, 2010.
- [4] F. Van Rysselberghe and S. Demeyer, “Reconstruction of successful software evolution using clone detection,” in *Proc. IWJSE'03*, 2003.
- [5] C. Schofield, B. Tansey, Z. Xing, and E. Stroulia, “Digging the development dust for refactorings,” in *Proc. ICPC '06*, 2006.
- [6] P. Weißgerber and S. Diehl, “Are Refactorings less error-prone than other changes?” in *Proc. MSR'06*, 2006.
- [7] J. Henkel and A. Diwan, “Catchup!: capturing and replaying refactorings to support api evolution,” in *Proc. ICSE'05*, 2005.
- [8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *Proc. ECOOP'06*, 2006.
- [9] Q. D. Soetens and S. Demeyer, “Studying the effect of refactorings: a complexity metrics perspective,” in *Proc. QUATIC'10*, 2010.
- [10] A. Murgia, M. Marchesi, G. Concas, R. Tonelli, and S. Counsell, “Parameter-based refactoring and the relationship with fan-in/fan-out coupling,” in *Proc. ICSTW '11*, 2011.
- [11] N. Rachatasumrit and M. Kim, “An empirical investigation into the impact of refactoring on regression testing,” in *Proc. ICSM'12*, 2012.
- [12] K. Stroggylos and D. Spinellis, “Refactoring—does it improve software quality?” in *Proc. WoSQ '07*, 2007.
- [13] E. Murphy-Hill, C. Parmin, and A. P. Black, “How we refactor, and how we know it,” *IEEE TSE*, vol. 38, no. 1, pp. 5–18, 2012.
- [14] H. Liu, Y. Gao, and Z. Niu, “An initial study on refactoring tactics,” in *Proc. COMPSAC'12*, 2012.
- [15] E. Murphy-Hill and A. Black, “Refactoring tools: Fitness for purpose,” *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.
- [16] T. Omori and K. Maruyama, “A change-aware development environment by recording editing operations of source code,” in *Proc. MSR'08*, 2008.
- [17] R. Robbes and M. Lanza, “Spyware: A change-aware development toolset,” in *Proc. ICSE'08*, 2008.
- [18] L. Hattori and M. Lanza, “Syde: A tool for collaborative software development,” in *Proc. ICSE'10*, 2010.
- [19] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt, “Change-oriented software engineering,” in *Proc. ICDL '07*, 2007.
- [20] Q. D. Soetens, S. Demeyer, and A. Zaidman, “Change-based test selection in the presence of developer tests,” in *Proc. CSMR'13*, 2013.
- [21] Q. D. Soetens and S. Demeyer, “ChEOPJSJ: Change-based test optimization,” in *Proc. CSMR'12*, 2012.
- [22] S. Demeyer, S. Tichelaar, and P. Steyaert, “FAMIX 2.0 - the FAMOOS information exchange model,” University of Berne, Tech. Rep., 1999.
- [23] A. Rensink, “The groove simulator: A tool for state space generation,” in *AGTIVE*, ser. Lecture Notes in Computer Science, vol. 3062, 2004, pp. 479–485.

# Automatically Extracting Instances of Code Change Patterns with AST Analysis

Matias Martinez, Laurence Duchien and Martin Monperrus  
 INRIA and University of Lille  
 Email: *firstname.lastname@inria.fr*

**Abstract**—A code change pattern represents a kind of recurrent modification in software. For instance, a known code change pattern consists of the change of the conditional expression of an *if* statement. Previous work has identified different change patterns. Complementary to the identification and definition of change patterns, the automatic extraction of pattern instances is essential to measure their empirical importance. For example, it enables one to count and compare the number of conditional expression changes in the history of different projects. In this paper we present a novel approach for search patterns instances from software history. Our technique is based on the analysis of Abstract Syntax Trees (AST) files within a given commit. We validate our approach by counting instances of 18 change patterns in 6 open-source Java projects.

## I. INTRODUCTION

Studying recurrent source code modifications in software is an essential step to understand how software evolves. *Change patterns* describe these kinds of modification. For instance, Pan et al. [1] identified 27 code change patterns related to bug fixing modifications. One of these pattern is “Addition of precondition check”. It represents a bug fix that adds an “if” statement to ensure that a precondition is met before an object is accessed or an operation is performed.

Research on change patterns focus on *definition* and *quantification*. The definition of code changes patterns consists of producing interesting *change pattern catalogs* (a.k.a *change taxonomies*). The change pattern quantification means measuring the number of code changes pattern instances. For instance, Pan et al. [1] counted 148 instances of pattern “Addition of precondition check” in the history of open-source project Columba.

Our motivation is to provide a generic way for specifying change patterns. The specification should be precise enough so as to automatically measuring the recurrence of change patterns, i.e. the number of instances. This would facilitate the replication of change pattern quantification experiments of the literature. One could also extract instances of known pattern from projects not considered in previous experiments. Furthermore, it would enable one to specify new patterns and assess their importance.

In this paper we propose an automated process to define source code change patterns and quantify them from software versioning history. Our technique is based on the automated analysis of differences between the abstract syntax trees (AST). We use the AST change taxonomy introduced by Fluri et al. [2]. We define a structure to describe a change pattern

using the mentioned AST change taxonomy. The identification of instances for a change pattern consists of selecting those revisions that contain the AST changes described by the pattern. This is done by calculating the AST differences between every file pairs of all commits (the new version and its ancestor). To our knowledge, this way of defining and quantifying change patterns is novel.

To sum up, this paper makes the following contributions:

- An approach to specify source code change patterns with an abstraction over AST differencing.
- An approach to automatically recognize concrete pattern instances based on the analysis of abstract syntax trees.
- An analysis of 18 change patterns from 6 Java open source project totaling 23,597 Java revisions (Java file pairs).

The remainder of this paper is as follows. Section II presents an approach to analyze software versioning history at the abstract syntax tree level. Section III is a first evaluation of our approach. Section IV discusses the related work. Section V concludes the paper.

## II. AST ANALYSIS OF SOFTWARE VERSIONING HISTORY

In this section, we present a method to detect whether a revision contains an instance of a change pattern. This method uses AST analysis and a tree differencing algorithm. In subsection II-A we present a representation of changes at the AST level. In subsection II-B we use this representation to codify code change patterns. We then define the notion of “hunk” for AST changes in subsection II-C. Finally, in subsection II-D we present our algorithm to extract instances of code change patterns.

### A. Representing Versioning Changes at the AST Level

We represent source code versions as changes at the AST level. For a given pair of consecutive versions of a source file, we compute the AST of both versions. We then apply a AST differencing algorithm to extract the essence of the change. Let us take as example the change presented in Figure 1. It shows a hunk pair and the syntactic differences between those revisions. It consists of a removal of code. At the AST level, our AST differencing algorithm finds two AST changes: one representing the removal of an *else* branch and another for the removal of a method invocation statement surrounded by the *else* block.

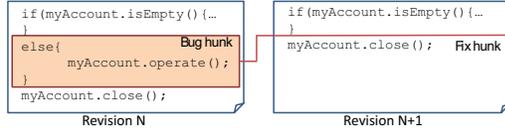


Fig. 1. A lined-based difference of two consecutive revisions. The bug hunk in revision N (the left one) contains an “else” branch. The fix hunk in revision N+1 is empty. The corresponding AST hunk (introduced in Section II-C) consists of two nodes removal i.e. the ‘else’ node and the method invocation.

To compute the set of AST changes between two source code files, we use the AST differencing algorithm ChangeDistiller [2]. We chose ChangeDistiller due to its fine-granularity change taxonomy and the availability of an open-source stable and reusable implementation of their algorithm for analyzing AST changes of Java code. ChangeDistiller provides detailed information on the AST differences between source files at the level of statements. It defines 41 source changes types, such as “Statement Insertion” or “Condition Expression Change” [3]. ChangeDistiller handles changes that are specific to object-oriented elements such as “field addition”. Formally, ChangeDistiller produces a list of “source code changes”. Each AST source code change is a 3-value tuple:  $scc = (ct, et, pt)$  where  $ct$  is one of the 41 change types,  $et$  (for entity type) refers to the source code entity related to the change (e.g. a statement update may change a method call or an assignment), and  $pt$  (for parent type) indicates the parent code entity where the change takes place<sup>1</sup> (such as a the top-level method body or inside an *if* block). For example, the removal of an assignment statement located inside a *For* block is represented as:  $scc = (\text{“Statement delete”, “Assignment”, “For”})$ . In the rest of the paper we also use a textual representation formed by the concatenation of  $ct$ +“of”+ $et$ +“in”+ $pt$ . For the previous example, it would be “Statement delete of Assignment in For”.

### B. Representing Change Patterns at the AST Level

We represent a change pattern with a structure formed of three elements: a list of micro-patterns  $L$ , a relation map  $R$ , and a list of undesired changes  $U$ .

$$\text{pattern} = \{L, R, U\}$$

A micro-pattern is an abstraction over ChangeDistiller AST changes. A micro-pattern is a tuple  $(ct, et, pt)$  where only the  $ct$  field is mandatory. The fields  $et$  and  $pt$  can take a *wildcard* character “\*”, meaning they can take any value. For example, a micro-pattern (“Statement Insert”, “\*”, “\*”) means that an insertion of any type of statements (e.g. assignment) inside any kind of source code entity, e.g. “Method” (top-level method statement) or “If” block. Moreover, the list of micro-patterns  $L$  is ordered according to their position inside the source code file. This means that a pattern formed by  $scc1$  and  $scc2$  is not equivalent to another formed by  $scc2$  and  $scc1$ . The former means that  $scc1$  occurs before  $scc2$ , while the latter the opposite.

Let us present the AST representation of pattern “Addition of Precondition Check with Jump” [1]. This pattern represents

<sup>1</sup>For change type “Statement Parent Change”, which represents a move,  $pt$  points to the new parent element.

the addition of an *if* statement that encloses a jump statement like *return*. It is represented by two AST changes<sup>2</sup>:  $scc1 = (\text{“Statement Insert”, “If”, “*”})$  and  $scc2 = (\text{“Statement Insert”, “Return”, “If”})$ .

The *relation map*  $R$  is a set of relations between the changes of the entities ( $et$ ) involved in the micro-patterns of  $L$ . The relation map also describes the link between them. For example, “Addition of Precondition Check with Jump” requires the entity *return* (affected by the change  $scc2$ ) to be enclosed by an *If* entity, which in turn is affected by the change  $scc1$ . In other words,  $scc2.pt = scc1.et$ .

The list of *undesired changes*  $U$  represents AST changes that must not be present in the pattern instances. For example, the pattern “Removal of an Else Branch” [1] requires only the *else* branch being removed, keeping its related *if* branch in the source code. In other words, it is necessary to ensure that some micro-patterns *do not* occur. For the pattern “Removal of an Else Branch”, there is one *undesired change* telling that the *if* element, parent of the removed *else* branch, must not be removed.

### C. Defining “Hunk” at the AST level

Previous work has set up the “localized change assumption” [1]. This states that the pattern instances lie in the same source file and even within a single hunk i.e., within a sequence of consecutive changed lines. From our experience, the “localized change assumption” is indeed relevant, especially to remove noise in the mining and matching process. Hence we define the notion of “hunk” at the AST level.

AST hunks are *co-localized source code changes*, i.e. changes that are near one from each other inside the source code. For us, an “AST hunk” is composed of those AST changes that meet one the following conditions: 1) they refer to the same syntactic line-based hunk. 2) they are moves within the same parent element. For instance, the two AST changes from the example of Figure 1 are in the same AST hunk (both changes occur in the same syntactic hunk). Note that there are the same number of AST hunks than line-based hunks or less. The reason is that AST hunks sometimes merge line-based hunks and also that AST hunks only consider AST changes. By construction, there is no AST hunks for changes related to comments or formatting, while, at the syntactic, line based level, those hunks show up.

### D. Searching Instances of AST change Patterns

This section presents an AST change classifier that decides whether a given pattern is present or not inside an AST hunk. The classification procedure has three phases: change mapping, identification of change relations and exclusion of AST hunks containing undesired changes. It takes as input one AST change pattern definition and an AST change hunk.

First, in the change mapping phase, we carry out a mapping between the AST changes of the hunk and those from the micro-patterns of the change pattern under consideration. Each

<sup>2</sup>to simplify the example, we exclude jump statements ‘break’ and ‘continue’

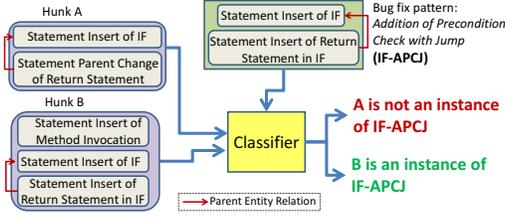


Fig. 2. The process of classifying AST hunks: At the top, a AST representation of pattern “Addition of Precondition Check with Jump” (IF-APCJ). At the left, two AST hunks (A and B), only B is an instance of the pattern.

mapping of AST changes means equality of their change type, entity type and parent types (unless wildcards are specified). The procedure ensures that all micro patterns of the change pattern actually appear in the AST hunk.

Then, in the change relation validation phase, we verify that all the relations from the pattern’s *relation map* are satisfied within the hunk changes. Finally, in the undesired changes validation phase, we verify that no change of the *undesired changes* list is present in the hunk change list. A pattern instance is present in the hunk if the validations made in the three phases are successful.

Figure 2 presents an example to illustrate the AST hunk classification procedure. On the left hand side, it shows two AST hunks: A is formed by two AST changes (“Statement Insert of If” and “Statement Parent Change of Return Statement”) and B formed by three AST changes (“Statement Insert of Method Invocation”, “Statement Insert of If” and “Statement Insert of Return Statement in If”). On the top, the figure shows the AST representation of Pan’s pattern “Addition of Precondition Check with Jump” (IF-APCJ).

First, let us classify hunk A. The first AST change of the pattern matches with the first-one of hunk A: both AST change type (“Statement Insert”) and entity type (“If”) match. The classifier continues by comparing the remaining AST changes. However, the second AST change of the pattern does not match with the second of A because the change types are different (“Statement Parent Change” vs. “Statement Insert”). As hunk A does not have more changes, the algorithm stops and says that hunk A is not an instance of the pattern.

Now, we proceed to classify hunk B. In the first comparison there is not matching between the first AST change of B and the first change of the pattern. However, as B contains more changes, the classifier continues comparing the remaining AST changes. Then, the classifier successfully matches the two micro-pattern with the second AST change and third AST changes of B, respectively. As the pattern has no more changes to map, the classifier then verifies the parent relation constraints. The pattern has one parent relation (“If” entity is parent of “return” entity), that is satisfied in AST hunk B. Consequently, the classifier says that pattern IF-ACPJ is present in B i.e. B is an instance of the pattern.

In this Section we have presented an approach to search for instance of change patterns. This approach is based on the analysis of AST differences, which is insensitive to formatting changes.

TABLE I  
VERSIONING DATA USED IN OUR EXPERIMENT. SINCE WE FOCUS ON BUG FIX PATTERNS, WE ANALYZE THE 23,597 JAVA REVISIONS WHOSE COMMIT MESSAGE CONTAINS “BUG”, “FIX” OR “PATCH”.

	#Commits	#Revisions	#Java Revisions
All	24,042	173,012	110,151
BFP	6,233	33,365	23,597

### III. EVALUATION

We now evaluate our approach to specify code change patterns at the level of ASTs. Our research questions are: Does our approach allow specifying existing change patterns of the literature? Does our approach scale to the analysis of long versioning history of large open-source projects?

#### A. Representing Known Code Change Patterns

Pan et al. [1] contributed with a catalog of 27 code change patterns related with bug fixing. They call them “bug fix patterns”. For instance, changing the condition of an *if* statement is one of Pan’s patterns, it is a kind of change that often fixes bugs.

We define AST change pattern representations for 18 bug fix patterns belonging to the categories If, Loops, Try, Switch, Method Declaration and Assignment. In summary, 18 patterns can be represented in this work. We have already discussed in much details, pattern “Addition of Precondition Check with Jump”. All 18 patterns and their AST changes are presented in appendix [4]. In Section III-C we discuss the limitations of our approach to express the remaining 9 patterns from the catalog.

To sum up, our approach enables us to specify existing change patterns of the literature. In the remaining of this section, we use these pattern representations to search for change instances in six Java open source projects.

#### B. Automatically Extracting And Counting Pattern Instances

We have searched for instances of the 18 patterns mentioned in III-A in the history of six Java open source projects: ArgoUML, Lucene, MegaMek, Scarab, jEdit and Columba. The complete descriptive statistics are given in appendix [4]. In Table I we present the total number of commits (versioning transactions) and revisions (file paiers) present in the history of these projects. In the rest of this section, we analyze the 23,597 Java revisions whose commit message contains “bug”, “fix” or “patch”, in a case insensitive manner (row “BFP” in Table I).

Table II proves that our approach based on AST analysis scales to the 23,597 Java revisions from the history of 6 open source projects. This table enables us to identify the importance of each bug fix pattern. For instance, adding new methods (MD-ADD) and changing a condition expression (IF-CC) are the most frequent patterns while adding a try statement (TY-ARTC) is a low frequency action for fixing bugs. Overall, the distribution of the pattern instances is very skewed, and it shows that some of Pan’s patterns are really rare in practice. Interestingly, we have also computed the results on all revisions – with no filter on the commit message – and the distribution of patterns is rather similar. It seems that the

TABLE II  
CONTEXT-INDEPENDENT BUG FIX PATTERNS: ABSOLUTE NUMBER OF  
PATTERN INSTANCES IN 23,597 JAVA REVISIONS.

Pattern name	Abs
Change of If Condition Expression-IF-CC	4444
Addition of a Method Declaration-MD-ADD	4443
Addition of a Class Field-CF-ADD	2427
Addition of an Else Branch-IF-ABR	2053
Change of Method Declaration-MD-CHG	1940
Removal of a Method Declaration-MD-RMV	1762
Removal of a Class Field-CF-RMV	983
Addition of Precond. Check with Jump-IF-APCJ	667
Addition of a Catch Block-TY-ARCB	497
Addition of Precondition Check-IF-APC	431
Addition of Switch Branch-SW-ARSB	348
Removal of a Catch Block-TY-ARCB	343
Removal of an If Predicate-IF-RMV	283
Change of Loop Predicate-LP-CC	233
Removal of an Else Branch-IF-RBR	190
Removal of Switch Branch-SW-ARSB	146
Removal of Try Statement-TY-ARTC	26
Addition of Try Statement-TY-ARTC	18
Total	21,234

bug-fix-patch heuristics does not yield a significantly different set of commits.

Knowing this distribution is important in some contexts. For instance, from the viewpoint of automated software repair approaches: their patch generation algorithms can concentrate on likely bug fix patterns first in order to maximize the probability of success.

This experiment allows us to answer our research questions. With our AST-based approach, we can specify existing change patterns of the literature and we can search instances from large version history of open-source projects.

### C. Limitations

There are patterns that can not be expressed with our approach. There are two reasons for this. First, some of them are context dependent, meaning that an instance is found only if the change is of a certain kind, and the context of the change is of the certain kind as well. For instance, there is one pattern representing removal of a method call in a sequence of method calls. To observe an instance of removal of a method call in a sequence of method calls: 1) the change itself has to be a removal of a method call 2) the context of the removal has to be a sequence of method calls on the same object. Expressing the context at the AST-level is future work.

The second reason is that some patterns involve an analysis grain that is not handled by ChangeDistiller. For instance, an update operation in a class field declaration is not detected by ChangeDistiller. This limitation prevents us to represent pattern “Change of Class Field Declaration” (CF-CHG) from Pan et al. catalog using AST changes.

### IV. RELATED WORK

Pan et al. [1] present a catalog of 27 bug fix pattern and a tool to extract instances of them from source code. Nath et al. [5] use the patterns to evaluate another Java open source project. However, they mined the pattern instances by hand.

Kim et al. [6] have introduced a taxonomy of signature change kinds. In contrast with our work, their experiment

focuses on calculate the frequency of these signature changes from eight open source project histories. Additionally, Kim et al. [7] present an approach called BugMem to detect potential bugs and suggest corresponding fixes. BugMem stores the bug fix instances information extracted from a particular project. In contrast with our work, they do not use explicit bug fix patterns definition for the instance identification.

Fluri et al. [8] use hierarchical clustering of source code changes to discover change patterns. As in our work, they use ChangeDistiller to obtain fine-grained source code changes. They concentrate on coarse grain change patterns (such as development change, maintenance change), while we focus on fine-grain, AST level bug fix patterns only.

Livshits and Zimmermann [9] propose an approach to detect error patterns of application-specific coding rules. The authors propose an automatic way to extract likely error patterns by mining software revision histories and checking them dynamically. This work is concentrated on method calls (i.e. patterns formed by added or removed method calls) while our work focus on pattern formed by any type of AST level changes from ChangeDistiller.

### V. CONCLUSION

In this paper, we have presented a methodology to automatically extract instances of source code change patterns based on the analysis of AST differences. We have applied on it 18 patterns of the literature and analyzed 23,597 Java revisions of 6 open-source Java projects. We are now setting up a comparative quantitative experiment to assess whether our AST-based analysis works better than token-based change pattern detection [1]. Also, it is future work to take into account the context of AST changes in order to be able to express more change patterns.

### REFERENCES

- [1] K. Pan, S. Kim, and E. J. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, pp. 286–315, Aug. 2008.
- [2] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, nov. 2007.
- [3] B. Fluri and H. Gall, “Classifying change types for qualifying change couplings,” in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pp. 35–45, 0-0 2006.
- [4] M. Martinez, L. Duchien, and M. Monperrus, “Appendix of “Studying Bug Fix Patterns with Automatic AST Analysis of Versioning History,”” tech. rep., INRIA, 2013. Available at <http://goo.gl/y4f1r>.
- [5] S. K. Nath, R. Merkel, and M. F. Lau, “On the improvement of a fault classification scheme with implications for white-box testing,” in *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12*, (New York, NY, USA), pp. 1123–1130, ACM, 2012.
- [6] S. Kim, E. J. Whitehead, and J. Bevan, “Analysis of signature change patterns,” *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005.
- [7] S. Kim, K. Pan, and E. J. Whitehead, “Memories of bug fixes,” in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.
- [8] B. Fluri, E. Giger, and H. C. Gall, “Discovering patterns of change types,” in *Proceedings of the International Conference on Automated Software Engineering*, 2008.
- [9] B. Livshits and T. Zimmermann, “Dynamine: finding common error patterns by mining software revision histories,” in *Proceedings of the European software engineering conference held jointly with International Symposium on Foundations of Software Engineering*, 2005.

# Identification of Refused Bequest Code Smells

Elvis Ligu, Alexander Chatzigeorgiou, Theodore Chaikalis, Nikolaos Ygeionomakis

Department of Applied Informatics

University of Macedonia

Thessaloniki, Greece

{mai1315, achat, chaikalis}@uom.edu.gr, nygeion@gmail.com

**Abstract**—Accumulated technical debt can be alleviated by means of refactoring application aiming at architectural improvement. A prerequisite for wide scale refactoring application is the automated identification of the corresponding refactoring opportunities, or code smells. One of the major architectural problems that has received limited attention is the so called 'Refused Bequest' which refers to inappropriate use of inheritance in object-oriented systems. This code smell occurs when subclasses do not take advantage of the inherited behavior, implying that replacement by delegation should be used instead. In this paper we propose a technique for the identification of Refused Bequest code smells whose major novelty lies in the intentional introduction of errors in the inherited methods. The essence of inheritance is evaluated by exercising the system's functionality through the corresponding unit tests in order to reveal whether inherited methods are actually employed by clients. Based on the results of this approach and other structural information, an indication of the smell strength on a 'thermometer' is obtained. The proposed approach has been implemented as an Eclipse plugin.

**Keywords**—software maintenance; refactoring; code smell; Refused Bequest

## I. INTRODUCTION

One of the most challenging activities, in terms of cost and effort, in the lifecycle of contemporary software systems is the process of maintenance, an inevitable consequence of software evolution. From the perspective of quality, as software systems evolve, their original architecture usually deteriorates due to the fact that design and implementation decisions are taken under time pressure. To confront software degradation over time, code and design refactorings can offer significant aid by improving the internal structure of a software system without changing its external behavior [2]. The application of a refactoring can eliminate specific architectural anomalies or principle violations, widely known as “code smells”, and restore the code structure that exhibited a smell, to an acceptable level of quality. However, while the mechanics for the application of each refactoring have been defined in detail [2], the identification of code smells that should be refactored is a non-trivial, time-consuming and challenging activity. To this end, a number of automated tools for the identification of code smells and the facilitation of software maintainers have been developed [6], [10].

In the context of object-oriented systems, the notion of inheritance has been recognized as a key feature claimed to

reduce the amount of software maintenance. However, inheritance is not a panacea, especially if it is applied incorrectly in cases where other forms of relationships would be more appropriate. The Refused Bequest code smell concerns an inheritance hierarchy where a subclass does not support the interface inherited from its parent class [2]. More precisely, this smell is present if the functionality inherited by the subclass is not utilized by its clients nor specialized by means of overriding. In other words, the relation between the superclass and the subclass does not constitute an “*is-a*” relationship. The appropriate refactoring is the “Replace Inheritance with Delegation” [2] which dictates to transform an inheritance relationship into composition where the subclass contains a reference to an object of the superclass and uses only the desired functionality. This refactoring is in agreement to the GoF suggestion “*Favor Composition over Inheritance*” [3]. It can be deduced that the Refused Bequest bad smell cannot emerge in abstract classes or interfaces.

This paper proposes a methodology for the identification of the Refused Bequest smell that employs static source code analysis for the identification of suspicious hierarchies and dynamic unit test execution for the determination of subclasses that actually exhibit the smell. Identified smells are sorted according to their intensity based on criteria such as the number of overridden methods, the invocation of superclass methods and the results from test execution. Smell interpretation is facilitated by a “Smell Thermometer” which depicts graphically the intensity of the smell. The approach has been implemented as an extension on the JDeodorant Eclipse plug-in [4] and is evaluated on an open-source project.

## II. KEY CONCEPT

The key idea behind the proposed identification technique lies in the detection of whether a subclass in a given hierarchy actually “*wants to support the interface of the superclass*” [2]. Refusing an inherited interface, in the sense that clients of the subclass do not invoke any of the inherited functionality (but rather access only new functionality) is a relatively clear sign of Refused Bequest. There are numerous factors that come into play and indicate whether the use of inheritance is justified or not, but the notion of “refusing” the inherited behavior implies that the particular generalization does not have the properties of an “*is-a*” relationship.

This is a property that in general is hard to assess without relying on human expertise and thus is difficult to automate.

However, we could rely on the potential invocations of subclass' methods from the rest of the classes to detect whether inherited methods (and consequently the interface of the superclass) are actually exploited by the subclass. This concept is illustrated with the help of Fig. 1 where inherited and additional methods of `Beta` can be accessed by the Client.

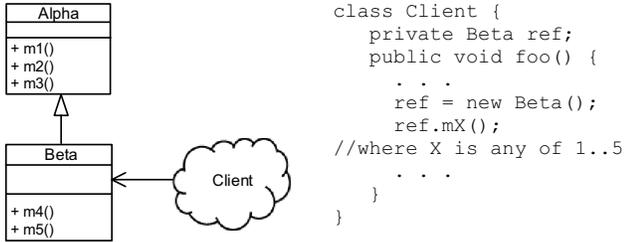


Fig. 1. Client accessing subclass methods.

According to the Dependency Inversion Principle [7] the proper scenario should be a client holding a reference to the superclass to exploit the benefits of polymorphism. However, in case the subclass contains additional methods, these can only be assessed by a client holding a reference of type `Beta`.

If the Client is to invoke only the additional methods of class `Beta` (`m4` and `m5`) without never calling the inherited superclass methods (`m1`, `m2` or `m3`), and the same holds for all clients of class `Beta`, it appears that the subclass somehow "denies" the inherited interface implying that generalization might not be appropriate (instances of `Beta` are not used as specializations of `Alpha` entities). The role of other parameters such as overriding and invocations of superclass methods through `super` will be discussed in the next section.

One way to detect whether superclass methods are actually invoked on subclass instances, is to override these methods in the subclasses and intentionally introduce an error in the corresponding implementation (such as a division by zero). If the corresponding method is invoked anywhere in the code base on instances of the subclasses, then, in case of overriding, the overridden methods will be invoked instead, causing an easily observable failure. If, despite the introduction of errors, the execution of all system scenarios does not lead to any failures, it can be concluded that the inherited superclass methods would not be actually used on any of the subclasses, providing a strong hint for the presence of Refused Bequest.

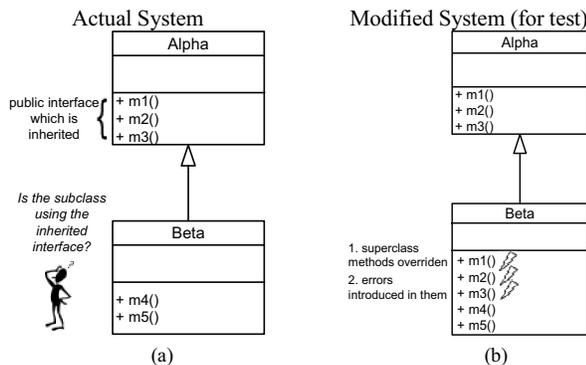


Fig. 2. (a) Inheritance relation under investigation, (b) Error insertion.

The proposed approach is illustrated in Fig. 2. Assuming that a subclass inherits behavior (Fig.2(a)), the question is whether the subclass actually uses the inherited interface. In Fig. 2(b) the non-overridden methods are now implemented in the subclass with errors deliberately introduced in them. If the execution of system functionality exhibits failures, it means that the overridden and faulty methods are invoked. In this case it can be concluded that in the initial system the inherited interface is not "Refused" and thus the smell is not present.

To exercise the system's functionality in order to reveal whether the inherited methods are invoked on instances of a subclass one could either rely on a) manually invoking all system functionality, b) executing specific methods that aim at demonstrating a large portion of the system's functionality and c) executing test cases within a project, assuming that a sufficient level of coverage is provided. We have relied on the third alternative since the execution of test cases can be automated and because for several open-source projects unit tests cover a large portion of the corresponding code base.

In contrast to other detection techniques which rely only on static analyses, the proposed approach can reveal dynamic information which is crucial for determining whether an inheritance relationship is appropriate or not. For example, polymorphic method invocations which totally justify the use of generalization, can only be detected by dynamic analysis.

### III. SMELL THERMOMETER

As already mentioned, the proposed approach takes several factors into account to assess the smell intensity. Based on the findings the following classification is obtained:

#### a) Abstract Superclass or Interface

When a designer employs a generalization relationship and places an abstract class or an interface at the root of the hierarchy, his/her intention is rather clear: The goal is to apply the Dependency Inversion Principle and essentially to allow polymorphic behavior where the public interface of the base abstract class (or interface) is implemented by a corresponding subclass. In these cases it is theoretically impossible to encounter a Refused Bequest symptom, since the same benefit cannot be achieved by other means. In other words, it is clearly evident that the employed generalization is on purpose, well-designed and constitutes an "is-a" relationship. When the superclass is neither abstract nor an interface, the following cases can be distinguished.

#### b) Overriding and Failures

In this case one or more superclass methods have been overridden. Moreover, when exercising the system functionality failures emerged because of the introduced errors. Here, we have two clear indications that the presence of Refused Bequest is highly improbable. First, since the designer re-implemented methods which have been inherited to provide functionality that is specific to the subclass, it can be deduced that the goal is to enable polymorphism. Second, the presence of errors indicates that the inherited methods are invoked on instances of the subclass, i.e. the inherited functionality is actually employed. Thus, we can conclude that generalization is appropriately applied and cannot be replaced. The picture in the following cases is less clear.

c) *Some overriding and No failures*

A first indication that Refused Bequest might be present is the lack of failures in the executed test cases. This means that (assuming that the test cases provide complete coverage) no method in the entire system has invoked inherited methods on subclass instances. In other words, subclasses exhibit signs of refusing the inherited interface. However, if at the same time one or more of the superclass methods are overridden in the subclass, the symptom is alleviated. Overriding allows polymorphism, something which would not be possible in case inheritance is replaced by delegation. Therefore, we consider this situation as a mixed case where only limited signs of Refused Bequest can be diagnosed.

d) *No overriding, some failures and invocation of super*

When the subclasses in a generalization do not override superclass' methods, the inherited interface is no longer specialized by the descendants in the hierarchy. As a result the designer's intention deviates from the goal of enabling polymorphic behavior or conforming to the requirements of a design pattern such as State, Strategy or Template Method. On the other hand, the presence of failures which implies that the inherited methods are invoked on subclass instances, is a sign towards the opposite direction. Since the subclass does not override superclass methods, the only alternative left (for an hierarchy to be meaningful) is to introduce additional methods to the subclass. This particular case can be further categorized, depending on whether the introduced subclass methods invoke superclass methods through the `super` keyword. Although method invocations through `super` could be refactored in case inheritance is replaced by delegation [5], the presence of the `super` keyword is an indication of a certain degree of reuse. Consequently, we consider the presence of superclass method invocations as a (relatively weak) indication that inheritance might be appropriate, at least in comparison to the next case.

e) *No overriding, some failures and No invocation of super*

Here, the only difference to the previous case is the lack of superclass method invocations in the additional methods of the subclass. In combination with the lack of overridden methods, these characteristics imply an even stronger probability that a Refused Bequest symptom actually exists. In fact, only the occurrence of some failures due to the introduced errors points to the opposite direction.

f) *No overriding, No failures*

This case constitutes the stronger indication that the Refused Bequest smell exists in the examined hierarchy. Here, no superclass method is overridden, none of the inherited methods is actually used on instances of the subclass and none of the additional subclass methods contains a superclass method invocation. In other words it appears as if the subclass refuses any connection to its superclass and the corresponding generalization can hardly be characterized as an "is-a" relationship. No argument in favor of inheritance can be made and the hierarchy can be safely refactored to delegation [5].

All of the aforementioned symptoms according to which the strength of the Refused Bequest smell can be deduced, are summarized visually in the Smell Thermometer of Fig. 3. The higher the "temperature" gets, the stronger the smell is.

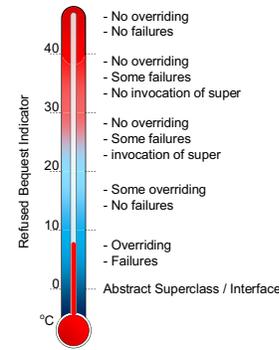


Fig. 3. Refused Bequest Thermometer.

To facilitate the identification of Refused Bequest smells we have extended the JDeodorant Eclipse plugin [4]. The plugin<sup>1</sup> enables the user to select either an entire Java project or a particular package, execute the identification and observe the findings, ordered by smell severity. The identification relies on the representation of the project under study as an Abstract Syntax Tree (AST) provided by the Eclipse JDT API.

#### IV. CASE STUDY AND THREATS TO VALIDITY

The developed plugin has been applied on SweetHome3D (v.4.0) which is an open-source Java interior design application. Size properties are shown in Table I. The approach revealed one characteristic example shown in Fig. 5.

TABLE I. OVERVIEW OF SWEETHOME3D SIZE METRICS

LOC	NUMBER OF PACKAGES	NUMBER OF CLASSES	NUMBER OF OPERATIONS	NUMBER OF INHERITANCE HIERARCHIES	NUMBER OF TEST CASES*
76730	9	460	3360	69	42

\*These tests act as integration tests exercising a large portion of software features.

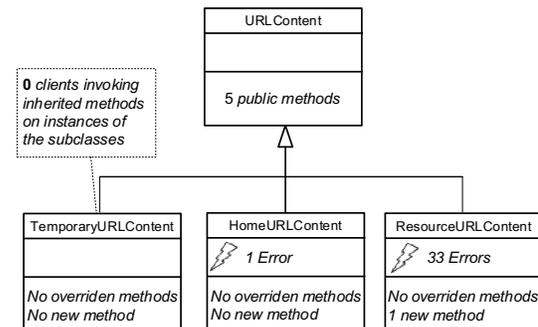


Fig. 4. No overriding, No failures case in SweetHome3D (Refused Bequest).

A `URLContent` enables the retrieval of resources from a Uniform Resource Locator. `TemporaryURLContent` is supposed to extend the `URLContent` parent class. However it provides no additional methods and consists of a single constructor and a single static method. No methods of the superclass are invoked through `super`. Overriding all 5 inherited methods and introducing errors into them has not led to any failure in the execution of JUnit test cases. Since there is no opportunity for polymorphic behavior and no use of the

<sup>1</sup> The plugin can be downloaded from [http://java.uom.gr/ref\\_bequest/](http://java.uom.gr/ref_bequest/)

inherited methods, it appears that this particular case exhibits the symptoms of a clear Refused Bequest. Even refactoring by means of delegation seems to make no sense for this case.

A similar, however slightly different observation can be made for the `HomeURLContent` subclass. Once again, no additional method is introduced. However when executing the test cases, one error was generated because of the flawed overriding of the superclass methods, implying that one of these methods has been invoked on a subclass instance. The symptom of Refused Bequest still exists since the rest of the superclass interface is refused by the subclass. In any case the intensity of the smell is lower than in the previous case.

Concerning the `ResourceURLContent` class, one additional method is introduced. More important is the fact that the introduction of errors in the overridden methods generated 33 failures, implying that the inherited interface is heavily used by clients of the subclass. As a result, Refused Bequest is hardly present. The only evidence that generalization is not exploited as much it could be, is the lack of any overridden methods (prohibiting essential polymorphism) and the lack of superclass method invocations.

The proposed detection process is based on the assumption that unit tests exercise thoroughly the system's functionality to reveal whether the inherited methods are actually invoked on instances of a subclass. This could impose a threat to construct validity which deals with how well the selected measures or tests can stand in for the concepts of interest. In particular, the exercised unit tests might not invoke the inherited methods on instances of the subclasses of interest and false positives might emerge. In other words, the introduced errors might not lead to test failures just because the unit tests have not been designed to cause the invocation of the corresponding methods and not because they are not actually utilized in the system. To mitigate this threat it is advised to perform the identification on projects with extensive test coverage, something which becomes typical for contemporary software projects.

## V. RELATED WORK

Although the frequency of the Refused Bequest smell is relatively low, it has been investigated by several researchers. Some works explicitly target the Refused Bequest smell, while others treat more general inheritance related problems. Zhang et al. [11] performed a literature review regarding approaches in the field of code smell detection and refactoring. According to their findings, 11 out of 39 relevant papers (28%), deal either with the identification or the removal of Refused Bequest. The need to analyze an hierarchy's clients to find out the original intention of a generalization has also been emphasized in [8]. Stefan Slinger [9] developed the CodeNose Eclipse plug-in which is capable of identifying the Refused Bequest smell by examining subclasses and the methods that they may override. Refused Bequest can also be identified by the detection strategy proposed by Marinescu [6] relying on a combination of selected code metrics and the definition of appropriate thresholds. Tourwe and Mens [10] employed logic meta-programming for the identification of *Inappropriate Interfaces*, which are unclear or incomplete interfaces hindering the evolution of hierarchies in a way that favors

polymorphism. Arevalo et al. [1] in their smell identification approach, which is based on Formal Concept Analysis, refer to this type of smell as "Unanticipated Dependency Schemas". Kegel and Steimann [5] defined pre and post conditions for the application of the "Replace Inheritance with Delegation" refactoring to alleviate the Refused Bequest smell.

## VI. CONCLUSIONS

Code smell detection is an extremely challenging maintenance task because it involves both static analysis for parsing structural code properties as well as dynamic examination of the context in which a particular code structure is being used. In this paper we have attempted to confront the problem of diagnosing Refused Bequest smells, employing a combination of static and dynamic analyses. The key contribution lies in the introduction of intentional errors in the non-overridden inherited methods, enabling designers to determine whether a subclass refuses the inherited interface or not. Measuring symptom severity on a smell thermometer can highlight suspect hierarchies that warrant further attention. We believe that the concept of intentionally introduced errors and the inspection of the resulting software behavior by means of test case execution can be generalized for the detection of other architectural problems.

## ACKNOWLEDGMENT

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thalys – Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

## REFERENCES

- [1] G. Arévalo, S. Ducasse, and O. Nierstrasz, "Discovering unanticipated dependency schemas in class hierarchies," *9th European Conference on Software Maintenance and Reengineering*, UK, March 2005, pp. 62-71.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [4] JDeodorant, <http://www.jdeodorant.com>, June 2013.
- [5] H. Kegel, and F. Steimann, "Systematically refactoring inheritance to delegation in java," *30th IEEE/ACM Int. Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 431-440.
- [6] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," *20th IEEE Int. Conference on Software Maintenance*, Chicago Illinois, USA, September 2004, pp. 350-359.
- [7] R.C. Martin, *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall, 2003.
- [8] P. F. Mihancea, "Towards a Client Driven Characterization of Class Hierarchies", *14th IEEE Int. Conference on Program Comprehension*. Athens, Greece, June 2006, pp. 285-294.
- [9] S. Slinger, "Code Smell Detection in Eclipse," Ph.D. Thesis, Delft University of Technology, March 2005.
- [10] T. Tourwe and T. Mens, "Identifying Refactoring Opportunities using Logic meta programming," *7th European Conference on Software Maintenance and Reengineering*, Italy, March 2003, pp. 91-100
- [11] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179-202, 2011.

# Code Smell Detection: Towards a Machine Learning-based Approach

Francesca Arcelli Fontana, Marco Zanoni and Alessandro Marino  
 Department of Informatics, Systems and Communication  
 University of Milano-Bicocca  
 Milano, Italy  
 Email: {arcelli, marco.zanoni}@disco.unimib.it,  
 a.marino4@campus.unimib.it

Mika V. Mäntylä  
 Aalto University  
 Helsinki, Finland  
 Email: mika.mantyla@aalto.fi

**Abstract**—Several code smells detection tools have been developed providing different results, because smells can be subjectively interpreted and hence detected in different ways. Usually the detection techniques are based on the computation of different kinds of metrics, and other aspects related to the domain of the system under analysis, its size and other design features are not taken into account. In this paper we propose an approach we are studying based on machine learning techniques. We outline some common problems faced for smells detection and we describe the different steps of our approach and the algorithms we use for the classification.

**Keywords**—code smells detection, machine learning techniques

## I. INTRODUCTION

Many tools for code smells detection have been developed, some of them are available to be used, for other tools one can find the paper on the tool but not the tool and other are commercial tools. The results provided by the detectors are usually different, as we observed in a previous paper [1], code smell detectors do not agree in their answers. Usually detection rules are based only on the computation of a set of metrics, well known object-oriented metrics or metrics defined ad hoc for the detection of a particular smell. The metrics used for the detection of a smell can be different, for example for the Large Class smell a tool can use different metrics of cohesion and complexity, while another one only the LOC metric; moreover, also if the metrics are the same, the thresholds of the metrics can be different. By changing this value, the number of smells increases or decreases accordingly. Some tools allow to change and set this value. Another problem regards the accuracy of the results, many false positive smells can be detected, not representing real problems and hence smells to be refactored, because information related to the context, the domain, the size and design of the analyzed system are not usually considered. Detection rules of smells not based on metrics computation are provided by the JDeodorant tool [2], which is able to detect four smells through the opportunities of applying refactoring steps to remove the smells.

Detection rules based on machine learning techniques have not been largely explored, and this is the reason why we decided to exploit them. In the paper we describe the process we follow, the empirical analysis we carried out and the machine supervised learning techniques which we experimented on the

Qualitas Corpus repository [3]. For all the reasons reported above and the intrinsic informal and subjective definition of smells, a benchmark platform has not been yet developed. Through our experimental analysis on a set of 76 systems of the Qualitas Corpus, we aim to provide also a significative dataset to be used as a basis for future benchmarks of code smells detectors.

The paper is organized through the following Sections: in II we introduce some related works; In III we briefly outline some common problems encountered through our experiences in code smells detection; in IV we describe our approach based on machine learning techniques and finally in V we provide some concluding remarks.

## II. RELATED WORK

We found few works in the literature exploiting machine learning techniques for code smell detection. Maiga et al. [4] introduce SVMDetect, an approach to detect anti-patterns, based on support vector machines. The subject of their study are Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife antipatterns on three open-source programs: ArgoUML, Azureus, and Xerces. Khomh et al. [5] present BDTEX (Bayesian Detection Expert), a Goal Question Metric approach to build Bayesian Belief Networks from the definitions of antipatterns and validate BDTEX with Blob, Functional Decomposition, and Spaghetti Code antipatterns on two open-source programs. Maiga and Ali [6] introduce SMURF, an approach to detect antipatterns, based on a machine learning technique with support vector machines and taking into account practitioners' feedback. Yang et al. [7] study the judgment of individual users by applying machine learning algorithms on code clones.

As we can see the principal differences of the previous works respect to our approach is that the above papers are principally focused on the detection of antipatterns and not of code smells of Fowler [8], they did their experimentations by considering only 2 or 3 systems and they usually experiment only one machine learning algorithm. While in our approach we focus our attention on 4 code smells, we consider 76 systems for the analysis and our validation and we experiment 6 different machine learning algorithms.

### III. CONSIDERATIONS ON CODE SMELL DETECTORS RESULTS

In a previous paper [1] we presented a comparison of four code smell detection tools and an assessment of the agreement, consistency and relevance of the answers produced. We investigated whether different tools for code smell detection, based on different algorithms, agree on their results or not. We considered smells analyzed by at least two different tools: Duplicated Code (analyzed by Checkstyle<sup>1</sup> and inFusion [9]), Feature Envy and God Class (analyzed by JDeodorant [2] and inFusion), Large Class and Long Parameter List (analyzed by Checkstyle and PMD<sup>2</sup>), Long Method (analyzed by Checkstyle, PMD and JDeodorant). We computed the kappa statistics of the tools, which is basically an attempt to balance the amount of accordance between the tools and the amount of accordance due to randomness (tools returned the same results, but not for the same reason). Our experiments outlined that different detectors for the same smell do not meaningfully agree in their answers. Nevertheless, they can detect problematic regions of code which are relevant for the future evolution of software. We observed that we have the best agreement in the results for the God Class detection and then for Large Class and Long Parameter List smells.

In another analysis we have done [10], we outline another problem regarding the reliability of the results of the detectors, in particular related to the recall values. With this aim, we have manually checked all the results provided by the tool iPlasma<sup>3</sup> on the detection of two smell, God and Data Class, on twelve systems of the Qualitas Corpus. The results of this analysis revealed that there is a high number of false positive results, detected smells that are not real smells and they have not to be refactored. These smells have been excluded because they represent domain or design dependent smells. We found many false positives with a total percentages of real God classes and Data classes of 23.38% and 13.67% respectively.

Starting from these observations on the low agreement on the results provided by different code smells detectors and on the high number of false positive smells detected by one tool (we have obviously to check if the same holds with other detectors), in the next Section we describe our proposal for code smells detection based on machine learning techniques.

#### IV. TOWARDS A MACHINE LEARNING BASED APPROACH

The application of machine learning to the code smell detection problem needs a formalization of the input and output of the learning algorithm and a selection of the data to be used and the algorithms to use in the experimentation. The following points summarize the principal steps of our approach, while the remainder of the section describes them.

- Collection of a large repository of heterogeneous software systems.
- Choice of a set of code smells and tools, or rules, for their detection.
- Application of the chosen tools/rules on the systems, recording the results for each class and method.

<sup>1</sup><http://checkstyle.sourceforge.net/>

<sup>2</sup><http://pmd.sourceforge.net/>

<sup>3</sup><http://loose.upt.ro/iplasma/index.htm>

Table I. ADVISORS

Code smell	Reference Tool/Detection rules
God Class	iPlasma (God Class, Brain Class), PMD
Data Class	iPlasma, Fluid Tool [13], Anti-Pattern Scanner [12]
Long Method	iPlasma (Brain Method), PMD, Marinescu detection rule [14]
Feature Envy	iPlasma, Fluid Tool

- Labeling: following the output of the code smell detection tools, the reported code smell candidates are manually evaluated, and they are assigned different degrees of gravity.
- Experimentation: The manual labeling is used to train a supervised classifier, whose performance (precision, recall, etc...) will be compared with the other tools.

For our analysis, we considered the Qualitas Corpus of systems collected by Tempero et al. [3]. The corpus, and in particular the collection we use, 20120401r, is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. We selected 76 systems of different size and computed a large set of object-oriented metrics that are considered as independent variables in our machine learning approach. The selected metrics are at class, method, package and project level; the set of metrics is composed of metrics needed by the exploited detection rules, plus standard metrics covering different aspects of the code, i.e., complexity, cohesion, size, coupling. The metrics were then detected on each system and recorded. To be able to correctly compute the metric values, most of the 76 systems were completed (e.g., adding missing libraries) in order to make them compile.

The first step towards the application of machine learning has been to identify a set of code smells defined in the literature, by choosing the ones having the high frequency [11], that may have the greatest negative impact [12] on the software quality, and which can be recognized by some of the available detection tools [1]. In this study, code smells represent the dependent variable of a machine learning task. At class level, we decided to detect God/Large Class and Data Class, while at method level, we detect Long/God/Brain Method and Feature Envy. For each code smell we identified in the literature a set of available detection tools and rules, which are able to detect them. We selected as many heterogeneous detection rules as possible, favoring the detection rules implemented by tools, because they should be more reliable and have a larger user base. The detection rules selected for each code smell are reported in I. Other tools have not been considered because it is not possible to run them in batch without a manual configuration of the projects to analyze.

Our proposal is to detect code smells using the selected tools and rules on the chosen 76 systems. The detection output will be used as a hint to select a class (or method) to be manually evaluated, producing a label for each class (or method). The labelling process involves the inspection of the code of the selected class or method, supported also by graphical code representations, like dependencies, call, or hierarchy graphs. This information is used to support the decision of which label to assign.

The labelling is done with a severity classification of code smells. The idea of severity classification has been applied for software defects for years and its industrial adaption is widespread. Recently, several authors have worked in predicting defect severity with machine learning techniques [15], [16], [17]. Our initial idea is that code smell severity classification can have one of the four possible values:

- 0 *No smell*: the class (or method) is not affected by the smell;
- 1 *Non-severe smell*: the class (or method) is only partially affected by the smell;
- 2 *Smell*: the smell characteristics are all present in the class (or method);
- 3 *Severe smell*: the smell is present, and has particularly high values of size, complexity or coupling.

The code smell severity classification can have two possible benefits. First, ranking of classes (or methods) by the severity of their code smells can aid software developers in prioritizing their work on the most severe code smells. Often developers have to work under time-constraints and they might not have time to fix all the code smells we can automatically detect. Second, the usage of the code smell severity classification for the labelling provides more information than a more traditional binary classification, as for machine learning purposes the code smell severity classification will be interpreted as an ordinal scale. This information can be exploited during the learning phase, or collapsed back to a binary classification by grouping together the values, e.g.,  $\{0\} \rightarrow \text{INCORRECT}$ ,  $\{1, 2, 3\} \rightarrow \text{CORRECT}$ . It is important to understand that different machine learning algorithms have different capabilities, and choosing an appropriate labeling helps to achieve better results. In fact, selecting an algorithm which does not exploit the ordinal information in the class attribute is similar to perform binary classification over the (four in our case) possible values; as a consequence, each class value defines a reduced training set, with respect to the binary classification.

The manual evaluation will be performed also on a random sample of the remaining classes and methods, to avoid the introduction of a bias in the evaluation.

After this phase the training set will be analyzed using some machine learning methods we already selected. We chose the following classifiers, in the version available in the Weka tool [18]: Support Vector Machines (SMO, LibSVM), Decision Trees (J48), Random Forest, Naïve Bayes, JRip, applied directly and through boosting approaches. The choice of the algorithms was made by selecting different algorithms representing different approaches, and possibly already exploited in the literature for similar purposes; a preference was made for algorithms able to give a human-readable explanation of their learning.

Machine learning algorithms need a dataset input format. We decided to create two datasets, one for class-level smells, and another one for method-level smells. In each dataset, each row represents a class or method, and has one feature for each metric gathered on the subject. In addition, a boolean feature represents the label telling if the subject is a code smell instance or not.

Before the application of machine learning, it is necessary to process the values of the metrics to avoid possible distortions

Table II. PERFORMANCE RESULTS FOR DATA CLASS CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.976	0.976	0.981
Random Forest	0.978	0.979	0.998
Naïve Bayes	0.819	0.824	0.955
JRip	0.966	0.967	0.970
SMO	0.969	0.969	0.961
LibSVM	0.947	0.947	0.927

Table III. PERFORMANCE RESULTS FOR GOD CLASS CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.964	0.964	0.947
Random Forest	0.973	0.974	0.989
Naïve Bayes	0.954	0.955	0.981
JRip	0.973	0.974	0.973
SMO	0.964	0.964	0.954
LibSVM	0.766	0.726	0.655

due to different scales of values; to this end, normalization and standardization will be applied.

We are experimenting each classifier through k-fold cross validation, in different configurations, to find the best one, in terms of the standard performance measures, e.g., precision and recall. For each classifier, we will produce a learning curve, to determine which algorithm learns more quickly, in addition to the absolute performances. Prior work has found that the criteria of code smells vary somewhat between people [19]. Thus, algorithms with a short learning curve are needed as they are quickly able to match the individual's personal criteria. Furthermore, the learning curve lets us to understand if adding new examples to the training set would be useful. The manual validation of code smells, in fact, is costly, and the effort should better be measured.

Some algorithms, like J48, Random Forest and JRip, will provide also human-readable rules; we are analyzing them to understand which (combination of) metrics have more influence on the detection.

The best classifier's results will be compared with the state of the art tools and rules, to measure the effectiveness of the classification approach. Finally, it might be that no single algorithm is superior in code smell detection. From website <http://www.kaggle.com/>, that hosts online data analysis competitions, we can find that winners often use several machine learning methods in combination [20]. Thus, we will also experiment with combining several machine learning algorithms if no clear winner arises.

#### A. Preliminary Results

In Tables II, III, IV, V we show the results of the machine learning algorithms used with their default parameters for each type of code smell. We use 10-fold cross-validation to assess the performance of predictive models. Each code smell data set contains 140 positive instances and 280 negative instances (420 instances), having a ratio of 33% of positive instances. The labeling was performed by three MsC students specifically trained for the task. The instances were taken from all the 76 analyzed systems.

Although the algorithms do not use optimized parameters, the results are very interesting. J48, Random Forest, JRip and

Table IV. PERFORMANCE RESULTS FOR FEATURE ENVY CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.940	0.941	0.943
Random Forest	0.952	0.952	0.991
Naïve Bayes	0.861	0.861	0.945
JRip	0.964	0.964	0.969
SMO	0.930	0.930	0.911
LibSVM	0.690	0.586	0.536

Table V. PERFORMANCE RESULTS FOR LONG METHOD CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.983	0.983	0.984
Random Forest	0.990	0.999	0.999
Naïve Bayes	0.930	0.932	0.968
JRip	0.992	0.993	0.993
SMO	0.966	0.967	0.964
LibSVM	0.692	0.590	0.539

SMO have accuracy values greater than 90% for all datasets, and on average they have the best performances. Naïve Bayes has slightly lower performances on Data Class and Feature Envy than on the other two smells. LibSVM performances are lower than the others (by far lower in three cases out of four). LibSVM belongs to the same family of predictive models of SMO, so the main reason of its lower performances is to be found in the default setting of the parameters. We expect better results with this class of models when we will experiment with different parameters.

## V. CONCLUDING REMARKS

In this paper we outlined some common problems of code smell detectors and we described the approach we are following based on machine learning-techniques. We aim through our experimental analysis and the results we obtained to:

- provide a large dataset containing: 1) the source code, 2) gathered source code metrics, 3) results of the used detection tools and rules, and 4) manually validated information of the code smells using code smell severity classification
- the dataset can be used to: 1) make statistically significant comparisons between the tools and algorithms, 2) host online competitions on the dataset to find the best possible code smell predictors. The competition results can be used as a baseline for a community effort in finding the best possible code smell predictor, and then can be extended with other tools (e.g., JDeodorant).
- provide a new code smell detector, able to evolve with the availability of new data; the same detector can be easily used to detect other code smells, when training data will be available.

## REFERENCES

- [1] F. Arcelli Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 5:1–38, aug 2012. [Online]. Available: [http://www.jot.fm/contents/issue\\_2012\\_08/article5.html](http://www.jot.fm/contents/issue_2012_08/article5.html)
- [2] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [3] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proceedings of the 17th Asia Pacific Software Engineering Conference*. IEEE Computer Society, December 2010, pp. 336–345.
- [4] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. Essen, Germany: ACM, 2012, pp. 278–281.
- [5] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtx: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011, the Ninth International Conference on Quality Software.
- [6] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *19th Working Conference on Reverse Engineering (WCRE 2012)*. Kingston, Ontario, Canada: IEE, October 2012, pp. 466–475.
- [7] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Filtering clones for individual user based on machine learning analysis," in *Proceedings 6th International Workshop on Software Clones (IWSC 2012)*. Zurich, Switzerland: IEEE Computer Society, June 2012, pp. 76–77.
- [8] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1999, <http://www.refactoring.com/>.
- [9] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, 2012.
- [10] V. Ferme, A. Marino, and F. Arcelli Fontana, "Is it a real smell to be removed or not," in *Presented at the RefTest 2013 Workshop, co-located event with XP 2013 Conference*, June 2013.
- [11] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.
- [12] S. Olbrich, D. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010, p. 10.
- [13] K. Nongpong, "Integrating "code smells" detection with refactoring tool support," Ph.D. dissertation, University of Wisconsin Milwaukee, August 2012. [Online]. Available: <http://dc.uwm.edu/etd/13>
- [14] R. Marinescu, "Measurement and quality in objectoriented design," Ph.D. dissertation, Department of Computer Science, "Politehnica" University of Timisoara, 2002.
- [15] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.
- [16] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.
- [17] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 215–224.
- [18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.
- [19] M. Mantyla, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, pp. 10 pp.–.
- [20] P. Aldhous, "Specialist knowledge is useless and unhelpful," *New Scientist*, no. 2893, pp. 28–29, 2012.

# Variations on Using Propagation Cost to Measure Architecture Modifiability Properties

Robert L. Nord<sup>1</sup>, Ipek Ozkaya<sup>1</sup>, Raghvinder S. Sangwan<sup>1,2</sup>, Julien Delange<sup>1</sup>, Marco González<sup>3</sup>, Philippe Kruchten<sup>3</sup>  
 Software Engineering Institute<sup>1</sup>      Pennsylvania State University<sup>2</sup>      Electrical & Computer Engineering<sup>3</sup>  
 Carnegie Mellon University      Malvern, PA, USA      University of British Columbia  
 Pittsburgh PA, USA      rsangwan@psu.edu      Vancouver, Canada  
 {rn, ozkaya, jdelange}@sei.cmu.edu      {marcog, pbk}@ece.ubc.ca

**Abstract**—Tools available for measuring the modifiability or impact of change of a system through its architecture typically use structural metrics. These metrics take into account dependencies among the different elements of a system. However, they fail to capture the semantics of an architectural transformation necessary to control the complexity and cost of making changes. To highlight such limitations, this paper presents a study where we applied a representative structural metric, called ‘propagation cost’, to archetypical architectural transformations known to affect system modifiability such as rearchitecting a tightly coupled system to a layered pattern. We observe that in its original form the propagation cost metric does not provide consistent indications of architecture health. Enhancing this metric based on the semantics of the architectural pattern and tactics used in the transformation show improvements. Our results demonstrate that these enhancements detect modifiability properties that are not detectable by the propagation cost metric.

**Keywords**—software architecture; modifiability; propagation cost; stability; change propagation; dependency analysis

## I. INTRODUCTION

Modifiability is the quality attribute of an architecture that relates to the cost of change and refers to the ease with which a software system can accommodate change [1]. Techniques that provide guidance on the extent of the modifiability or extensibility qualities of a system often measure code-level metrics such as coupling, cohesion, cycles and code clones. Preventing an architectural change that compromises modifiability, however, is not simply a matter of controlling the quality of a system through the use of these code-level metrics [2]. A recent exploratory study by Bertan *et al.* revealed that many code-related issues detected were not correlated with architectural problems, and many issues that were not detected had architectural implications [3].

More recently, structural metrics based on the use of dependency structure matrices (DSM), also called design structure matrices, have been studied to assist with architecture-level analysis, such as the value of modular designs [4][5][6]. Stability and propagation cost metrics [7] extracted from a DSM view of the architecture indicate the likelihood of change propagation and, consequently, its impact on future maintenance costs and ease of making modifications. Several studies highlight the need to analyze dependencies in detecting design violations such as those in modifiability [8] and provide advice on using other dependency sources to augment the

information in a dependency matrix [6]. Increasingly such forms of analysis are being supported by commercial tools such as Sonar, Lattix and Structure 101 [9].

In this paper, we study the DSM-based *Propagation Cost* metric (PC) and we explore the following research questions:

1. Does PC measure modifiability properties to understand the impact of making a change to the architecture?
2. What are the limitations of PC, and how can it be enhanced and calibrated to provide practical guidance in understanding the impact of making a change to the architecture of a given system?

For this purpose, we use a commonly known transformation to improve modifiability: *strictly layered*, where a system is organized into layers with a higher-level layer only allowed to use an adjacent layer immediately below. We apply PC to gauge the state of the system before and after this architectural transformation, thereby highlighting its limitations. We then propose variations of this metric by adding dependency strengths to the DSM view of the architecture. We enhance PC to take into account the information about dependency strength and how it propagates along dependency paths.

While dependency strengths have been used in metrics on code and module structure, they treat dependencies alike. Our objective is to see if improvements to the metrics, using dependency strengths representative of the semantics inherent in the transformation, provide better insights that are in line with the modifiability properties of an architectural design while preserving their usefulness by ensuring such metric improvements can be supported by tools.

## II. STRUCTURAL METRICS

Consider a dependency graph showing elements of a system and their dependencies in Fig. 1(a). This dependency graph can be represented as a DSM, shown in Fig. 1(c), where the rows and columns are the elements from the graph and the cells are the dependencies among these elements.

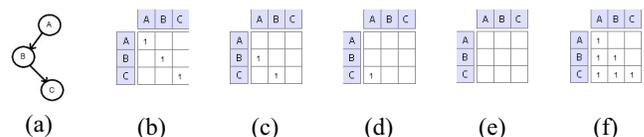


Fig. 1. (a) Dependency Graph (b) Identity DSM  $D^0$ , (c) direct dependency DSM  $D^1$ , (d) DSM  $D^2$ , (e) DSM  $D^3$ , and (f) visibility matrix V

The PC metric characterizes “the degree to which a change in a single element causes a potential change to other elements

in the system, directly or indirectly [7].” To determine the impact of elements in a system, a visibility matrix (V) is computed as follows:

$$V = \sum_{i=0}^n D^i \quad (1)$$

Matrices D and V are shown in Fig. 1(b) – (f) for the system shown in Fig. 1(a) where  $D^0$  represents dependencies of elements on themselves,  $D^1$  represents all direct dependencies, and  $D^p$  represents all indirect dependencies of length  $p$  where  $2 \leq p \leq n$  and  $n$  is the number of elements in a system.

PC, therefore, represents the density (or the proportion of cells marked 1) and is computed using V as follows:

$$PC = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n V_{ij} \quad (2)$$

In the case of Fig. 1, PC is 67% implying a change to an element might affect 67% of the system. This type of analysis forms the basis for the structural metric called stability that tools such as Lattix use to assist with software architecture management [5]. In the rest of the paper, we will focus on PC and use it as a representative structural metric.

### III. ARCHITECTURE DEPENDENCY ANALYSIS

We apply PC for analyzing the scenario shown in Fig. 2 where the architecture of a system is transformed using a strictly layered pattern. The *strictly layered pattern* is a division of software into units called layers. The layers are related to each other by the strictly ordered relation, *allowed-to-use*. Transforming a design to be strictly layered consists of removing connections that bypass one or several layers.

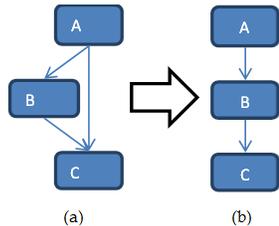


Fig. 2. System (a) before, and (b) after applying strictly layered pattern

In the example in Fig. 2, the connection from A to C bypasses element B. Applying the strictly layered transformation consists of identifying B as the intermediary between A and C and removing the dependency between A and C so that connection from A to C has to pass through B. One would, therefore, expect the architecture of the system to have changed with respect to the modifiability quality attribute depending on the role chosen for B in weakening the dependencies. The result of applying PC (modified to omit the diagonal or identity matrix  $D^0$ ), however, does not show this. PC is 33% for both before and after the transformation.

To adequately reflect the semantics of the architecture transformation we introduce variations to PC and discuss next.

### IV. VARIATIONS IN CALCULATING PC

We improve the shortcomings associated with the calculation of PC by focusing on the strength of the dependencies and how they propagate along paths in the dependency graph in series and in parallel.

Consider the dependency graph in Fig. 3.

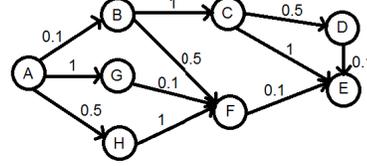


Fig. 3. Dependency graph showing the strength of dependencies

We use values between 0 and 1 for the strength based on measures taken to shield a dependent element from the ripple effect of change made to an element it depends on.

Suppose we have a path of length  $n \geq 2$  from element X to Z that goes through element Y such that the path from X to Y is of length  $n - 1$  and from Y to Z is of length 1. Then the change propagation cost of dependencies in series,  $Cost(X \rightarrow Z)_n$ , is the minimum of the PC of path from X to Y,  $Cost(X \rightarrow Y)_{n-1}$ , and PC of path from Y to Z,  $Cost(Y \rightarrow Z)_1$ :

$$Cost(X \rightarrow Z)_n = \min(Cost(X \rightarrow Y)_{n-1}, Cost(Y \rightarrow Z)_1) \quad (3)$$

For a path of unit length the change propagation cost is the strength of the dependency associated with that path. Using this convention, the change propagation cost for the path from element A to element D in Fig. 3 would be 0.1. In other words, the only change that propagates from element D to element A is one that is able to ripple through tactics employed along the way in element B to prevent a ripple effect of such a change.

Consider  $p$  paths of length  $n$  ( $n \geq 2$ ) from X to Z. We use three variants to calculate the change propagation in parallel:

**SUM:** The SUM of parallel paths from a component X to component Z is computed as follows:

$$Cost_{SUM}(X \rightarrow Z) = \sum_{i=1}^p Cost(X \rightarrow Z)_n \quad (4)$$

$Cost(X \rightarrow Z)_n$  for each path is computed as in equation (3). Using this convention, the change propagation cost from element A to F is 0.7. What this means is the changes to F are equally likely to propagate along each path from F back to A ( $A \rightarrow B \rightarrow F$ ,  $A \rightarrow G \rightarrow F$ , and  $A \rightarrow H \rightarrow F$ ) and, therefore, the impact of change on A is the aggregate of change propagation along each path ( $0.1 + 0.1 + 0.5$ ).

**AVG:** We SUM the path as discussed above, but divide the result by the number of paths.

$$Cost_{AVG}(X \rightarrow Z) = \frac{Cost_{SUM}(X \rightarrow Z)}{p} \quad (5)$$

Using this convention, the change propagation cost from element A to F is 0.23. This means that the changes to F are, on average, only likely to propagate along one of the paths from F back to A ( $A \rightarrow B \rightarrow F$ ,  $A \rightarrow G \rightarrow F$ , and  $A \rightarrow H \rightarrow F$ ). Therefore, the impact of change on A is the average of change propagation along each path ( $[0.1 + 0.1 + 0.5] / 3$ ).

**MAX:** We take the value of the most costly path from component X to component Z. Considering we have  $p$  paths between X and Z, the following is used to compute the cost:

$$\text{Cost}_{\text{MAX}}(X \rightarrow Z) = \max(\text{Cost}_1(X \rightarrow Z)_n, \dots, \text{Cost}_p(X \rightarrow Z)_n) \quad (6)$$

$\text{Cost}_p(X \rightarrow Z)_n$  is the cost of a given path  $p$  of length  $n$  computed using equation (3). Using this convention, the change propagation cost from A to F is 0.5. This means the changes to F are more likely to propagate along a path from F back to A that makes the least effort to prevent such changes from rippling through. Therefore, the impact of change on A is the maximum of 0.1, 0.1 and 0.5, the change propagation along each path.

## V. USING VARIATIONS ON STRICT LAYERING PATTERN

To improve the proposed change propagation cost formulas we will examine how architectural patterns support the modifiability of a system using architectural tactics [10] [11][12]. Modifiability tactics (such as *encapsulate*, *use an intermediary* and *restrict dependencies* that reduce coupling, *semantic coherence* that increases cohesion, and *split module* that reduces the size of a module) are architectural design decisions that control the time and cost of making changes.

The initial and expected strengths of the dependencies and the role of element B in Fig. 2 determine whether the transformation improves or deteriorates the design. We use three values for the strength – small (0.1), medium (0.5) and large (1). A small valued dependency indicates a use of one or more architectural tactics to prevent the ripple effect of a change, whereas a large valued dependency indicates absence of any architectural tactic that could prevent the ripple effect. A medium valued dependency indicates use of some tactics, but they are not sufficient in preventing ripple effects.

We demonstrate three scenarios:

1. Scenario 1, a baseline scenario where all dependencies have equal strength since the role of B is not clear
2. Scenario 2 where B acts as an interface and uses modifiability tactics such as *encapsulate*, *use an intermediary*, and *restrict dependencies* to reduce coupling
3. Scenario 3 where B acts as a conduit and uses modifiability tactics such as *use an intermediary* and *restrict dependencies* to reduce coupling

Scenario 1 simulates the original PC outlined in section II and gives us a reference point to compare the variations. Fig. 4 shows DSMs of the system before and after applying the strictly layered pattern and change propagation cost.

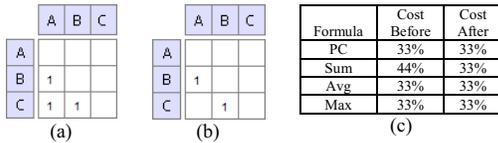


Fig. 4. Scenario 1 DSM (a) before and (b) after applying strictly layered pattern; (c) change propagation cost

The original PC and the formula variations we suggest compute the same result with the exception of *Sum* for the before case. *Sum* takes into account the changes that may

propagate from both of the parallel paths which may be cumulative. Therefore, *Sum* shows an improvement since the parallel paths between A and C are broken. Using the original PC of this scenario associates the same strength with each dependency and does not always accurately reflect the semantics of the change in the design.

Fig. 5 shows DSMs before and after applying the strictly layered pattern in scenario 2. In this scenario, A uses the interface B, but also has a direct connection to C. After the transformation, the dependency weights reflect the strictly layered pattern where B acts an intermediary and weakens the dependencies between A and C.

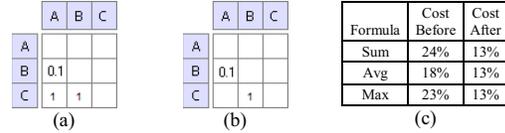


Fig. 5. Scenario 2 DSM (a) before and (b) after applying strictly layered pattern; (c) change propagation cost

The transformation uses the tactics *use an intermediary* and *restrict dependencies* to break the dependency between A and C. Furthermore, the tactic *encapsulate* is applied so B doesn't just pass on all information from C to A. The semantics of the transformations indicate that the change propagation of the structure has improved since the strong dependency between A and C is broken. All of the variations indicate an improvement.

Fig. 6 shows DSMs before and after applying the strictly layered pattern in scenario 3. In this scenario, B is loosely coupled with A and C and does not act as an interface between these two components. After the transformation, B has to know more about C to act as an intermediary for A. Thus, removing the direct connection between A and C increases the dependency strengths between A and B and B and C. Indeed, all information passing between A and C (that are strongly dependent) would go through B.

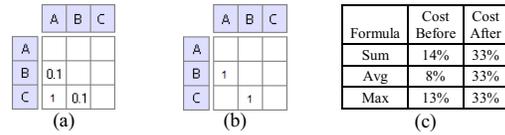


Fig. 6. Scenario 3 DSM (a) before and (b) after applying strictly layered pattern; (c) change propagation cost

The transformation uses the tactics *use an intermediary* and *restrict dependencies*, but without enforcing information hiding imposed by the *encapsulation* tactic. The semantics of the transformations indicate that the structure has deteriorated since previously A and B and B and C were loosely coupled. After the transformation they are strongly coupled since B is acting as a conduit to provide all information from C to A. The variation formulas indicate a change for the worse.

## VI. DISCUSSION AND CONCLUSIONS

Our study focused on two research questions. First, we investigated if the existing DSM-based structural metric, PC, reflects the impact of making a change to the architecture of a given system. Our example pattern of architectural transformations shows that in its current form PC does not reflect the semantics of architectural transformations. This

finding is in line with recent studies that look at the code metrics [3], and extend that to structural metrics [13].

Second, we explored different ways of enhancing and calibrating PC to provide practical guidance in understanding the impact of making a change to the architecture of a given system. Our enhancements were motivated by several limitations in the way change propagation cost is calculated:

- *The current propagation cost metric takes into account the presence or absence of a dependency.* We added dependency strength to the model and included formulas for handling change propagation in series and in parallel.
- *It assumes that all components are equally likely to be changed when the system evolves.* Dependency strength and certain formulas for change propagation in parallel (e.g., *Avg*, *Max*) weaken the assumption that all components contribute equally.
- *In case of smaller number of components, the contribution of self-dependencies is too high.* Archetypical transformations have very small matrices. This makes the metric very sensitive to a change when self-dependencies are considered, a phenomenon not observed in large systems (e.g., when the dimension is greater than 12). We omit the diagonal (the identity matrix  $D^0$ ) in the calculation to reduce the influence of a smaller number of components.
- *The transitive effect of indirect dependencies is too high.* Our use of dependency strength and formula for change propagation in series take into account the muting effect of any tactics used to prevent a change from rippling.

Our initial results hold promise for improving existing structural metrics, such as stability and PC, with common architectural patterns semantics, illustrating the range of design parameters for reducing the size of a module, increasing cohesion, and reducing coupling, and the tactics for manipulating them. We applied the same approach to two other architectural transformations, *module split* where a large element is split into two or more so each has fewer dependencies and *short circuit* where connections are bypassed by direct communication. We computed similar results and in the interest of space reported on the *strictly layered* pattern.

A validity concern for this study is whether the variation formulas will still perform consistently when applied in large systems. In addition, in some of our scenarios *Sum* and *Max* show greater ranges of improvement compared to *Avg*. To investigate whether this range is meaningful and whether any one of the variations among *Sum*, *Avg*, or *Max* perform better consistently, we are applying our variation metrics on an open source health IT project, CONNECT, where we have insights into its architecture of version 3.x through an architecture evaluation study we were asked to conduct [14].

The results of our study present an incremental, but significant improvement over the current state of practice showcasing how existing structural metrics such as PC and stability may yield results that can be misleading. Our approach emphasizes the significance of design decisions (architectural patterns and tactics) in moderating change propagation. While this is not surprising, the ability to measure the influence of these decisions in a way that can inform the architects and instill confidence in the choices they make is significant.

## ACKNOWLEDGMENT

We thank Neeraj Sangal and Frank Waldman for their feedback in automating the analysis. DSM figures are drawn using Lattix.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0000486.

## REFERENCES

- [1] ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.
- [2] R. Marinescu and D. Ratiu. Quantifying the Quality of Object-Oriented Design: The Factor-Strategy Model, in Proc 11th Working Conference on Reverse Engineering, Delft University of Technology, The Netherlands, November, 2004, pp. 192-201.
- [3] M.I. Bertan, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, A. von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. AOSD 2012: 167-178
- [4] C. V. Lopes and S. Bajracharya. 2005. An Analysis of Modularity in Aspect-Oriented Design. In Proceedings of Aspect-Oriented Software Development. Chicago, IL, March 14-18, 2005. ACM.
- [5] C. Hinsman, N. Sangal, J. Stafford. Achieving agility through architecture visibility. in Proc QoSA 2009, pp. 116–129, 2009.
- [6] T. Callo Arias, P. van der Spek, & P. Avgeriou, (2011). A practice-driven systematic review of dependency analysis solutions. Empirical Software Engineering 1-43.
- [7] A. MacCormack, J. Rusnak, and C. Y. Baldwin. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code." Management Science 52, no. 7 (July 2006).
- [8] S. Wong, Y. Cai, M. Kim, M. Dalton: Detecting software modularity violations. ICSE 2011: 411-420
- [9] A. Telea, L. Voinea, H. Sassenburg. "Visual Tools for Software Architecture Understanding: A Stakeholder Perspective," Software, IEEE , vol.27, no.6, pp.46-53, 2010.
- [10] F. Bachman, L. Bass, and R. Nord. 2007. Modifiability Tactics. Technical Report CMU/SEI-2007-TR-002 September 2007. Carnegie Mellon Software Engineering Institute. <http://www.sei.cmu.edu/library/abstracts/reports/07tr002.cfm>
- [11] C. Hofmeister, P. Kruchten, R.L. Nord, H. Obbink, A. Ran, P. America. A General Model of Software Architecture Design Derived from Five Industrial Approaches, Software Architecture Section, Journal of Systems and Software, Elsevier, 2007.
- [12] N. Harrison, P. Avgeriou, How do Architecture Patterns and Tactics Interact? A Model and Annotation, J. of Systems and Software, Elsevier, vol. 83, no. 10, October 2010, pp. 1735-1758.
- [13] J. Brondum and L. Zhu, "Visualising architectural dependencies," Managing Technical Debt (MTD), 2012 Third International Workshop on , vol., no., pp.7,14, 5-5 June 2012.
- [14] N. A. Ernst, I. Ozkaya, R. L. Nord, J. Delange, S. Bellomo, I. Gorton, "Understanding the Role of Constraints on Architecturally Significant Requirements" 3<sup>rd</sup> Int. Workshop on the Twin Peaks of Requirements and Architecture, July 2013.

# Multi-Objective Optimal Test Suite Computation for Software Product Line Pairwise Testing

Roberto E. Lopez-Herrejon\*, Francisco Chicano<sup>†</sup>, Javier Ferrer<sup>†</sup>,  
Alexander Egyed\* and Enrique Alba<sup>†</sup>

\* Systems Engineering and Automation  
Johannes Kepler University Linz, Austria  
Email: {roberto.lopez, alexander.egyed}@jku.at

<sup>†</sup>University of Malaga, Spain  
Email: {chicano, ferrer, eat}@lcc.uma.es

**Abstract**—Software Product Lines (SPLs) are families of related software products, which usually provide a large number of feature combinations, a fact that poses a unique set of challenges for software testing. Recently, many SPL testing approaches have been proposed, among them pairwise combinatorial techniques that aim at selecting products to test based on the pairs of feature combinations such products provide. These approaches regard SPL testing as an optimization problem where either coverage (maximize) or test suite size (minimize) are considered as the main optimization objective. Instead, we take a multi-objective view where the two objectives are equally important. In this exploratory paper we propose a zero-one mathematical linear program for solving the multi-objective problem and present an algorithm to compute the true Pareto front, hence an optimal solution, from the feature model of a SPL. The evaluation with 118 feature models revealed an interesting trade-off between reducing the number of constraints in the linear program and the runtime which opens up several venues for future research.

## I. INTRODUCTION

*Software Product Lines (SPLs)* are families of related software products, where each product provides a unique combination of *features* (i.e. increments in program functionality [1]). Some of the benefits of SPLs are increased software reuse, faster product customization, and reduced time to market [2]. A *feature model (FM)* represents all the possible feature combinations (typically a large number) available in an SPL. The number of combinations poses a unique set of challenges because testing each individual product may not be technically or economically feasible.

Recent surveys and mapping studies on SPL testing [3], [4], attest the increasing relevance of the topic within the SPL community. Salient among the SPL testing approaches are those based on *Combinatorial Interaction Testing (CIT)*, whose premise is to select a group of products where faults due to feature interactions are more likely to occur [5]. Here most of the focus has been on *pairwise* interactions, meaning that these techniques consider the four possible combinations between any two features<sup>1</sup>. The combination of features in a product of an SPL determines the set of pairwise feature combinations that the product *covers*. Pairwise SPL testing aims to select a set of products such that their feature combinations cover the possible combinations of *all* interactions between two

<sup>1</sup>For A and B features: both selected, both not selected, A selected and B not, A not selected and B selected.

features according to the feature model of the SPL. This set of products is called a *test suite*. Pairwise SPL testing approaches have used different techniques such as simulated annealing [6], evolutionary algorithms [7], and constraint programming [8]. These approaches regard SPL pairwise testing as an optimization problem where either coverage (maximize) or test suite size (minimize) are considered as the main optimization objective. Instead, we regard SPL pairwise testing as a multi-objective optimization problem where the two objectives, coverage and test suite size, are equally important. In the bi-objective formulation of the problem we say that one solution *dominates* another if the first is not worse than the second one in any objective and it is better in at least one objective. A set of solutions is said to be *non-dominated* if none dominates another. A *Pareto optimal set* is a set of non-dominated solutions each of which is not dominated by any other solution in the search space. The *Pareto front* is the projection of this set in the objective space, a plot containing the values of the objective functions for each solution. For more details on multi-objective optimization please refer to [9]. We present a zero-one mathematical linear program for solving the multi-objective problem and an algorithm that computes the true Pareto front of a feature model using SAT solvers. This front is the optimal solution for both objectives. We applied our approach to 118 publicly available feature models and were able to obtain their Pareto front. Our evaluation found a correlation between runtime and number of products in the feature model and revealed a trade-off between reducing the number of constraints in the mathematical linear program and runtime that we plan to explore as future work.

## II. FEATURE MODELS AND RUNNING EXAMPLE

Feature models have become the *de facto* standard for modelling the common and variable features of an SPL and their relationships collectively forming a tree-like structure. The nodes of the tree are the features which are depicted as labelled boxes, and the edges represent the relationships among them. Feature models denote the set of feature combinations that the products of an SPL can have [10]. Figure 1 shows the feature model of our running example, the *Graph Product Line (GPL)* [11], a standard SPL of basic graph algorithms that has been extensively used as a case study in the product line community. In GPL, a product is a collection of algorithms applied to directed or undirected graphs.

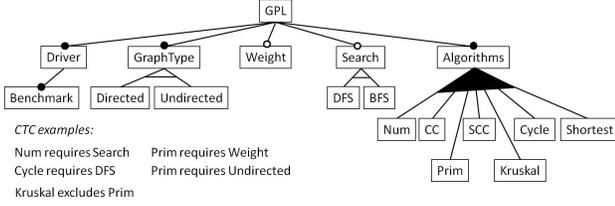


Fig. 1. Graph Product Line Feature Model

In a feature model, each feature (except the root) has one parent feature and can have a set of child features. Notice here that a child feature can only be included in a feature combination of a valid product if its parent is included as well. The root feature is always included. There are four kinds of feature relationships: *i) Mandatory features* are depicted with a filled circle. A mandatory feature is selected whenever its respective parent feature is selected. For example, features *Driver* and *GraphType*, *ii) Optional features* are depicted with an empty circle. An optional feature may or may not be selected if its respective parent feature is selected. An example is feature *Weight*, *iii) Exclusive-or relations* are depicted as empty arcs crossing over a set of lines connecting a parent feature with its child features. They indicate that exactly one of the features in the exclusive-or group must be selected whenever the parent feature is selected. For example, if feature *Search* is selected, then either feature *DFS* or feature *BFS* must be selected, *iv) Inclusive-or relations* are depicted as filled arcs crossing over a set of lines connecting a parent feature with its child features. They indicate that at least one of the features in the inclusive-or group must be selected if the parent is selected. If for instance, feature *Algorithms* is selected then at least one of the features *Num*, *CC*, *SCC*, *Cycle*, *Shortest*, *Prim*, and *Kruskal* must be selected. Besides the parent-child relations, features can also relate across different branches of the feature model with the so called *Cross-Tree Constraints (CTC)*. Figure 1 shows some of the CTCs of our feature model<sup>2</sup>. For instance, *Cycle* requires *DFS* means that whenever feature *Cycle* is selected, feature *DFS* must also be selected. These constraints as well as those implied by the hierarchical relations between features are usually expressed and checked using propositional logic, for further details refer to [12].

Let us illustrate pairwise coverage in GPL. This example has 73 distinct products each with its unique feature combination. Consider for instance the product that computes numbering in DFS order on directed graphs without weight. For this product the features selected are: *GPL*, *Driver*, *Benchmark*, *GraphType*, *Directed*, *Search*, *DFS*, *Algorithms*, and *Num*. Some examples of combinations of pairs of feature interactions are: *GPL* and *Search* selected, *Weight* and *Undirected* not selected, *CC* not selected and *Driver* selected. An example of invalid pair, i.e. not denoted by the feature model, is features *Directed* and *Undirected* both selected. Notice that this pair is not valid because they are part of an exclusive-or relation. In total, GPL has 418 valid pairs, so a test suite for GPL must have these pairs covered by at least one product feature combination.

<sup>2</sup>In total, the feature model has 13 CTCs for further details refer to [11].

### III. MATHEMATICAL LINEAR PROGRAM

We are interested in minimizing the number of test products and maximizing the pairwise coverage. Since we want to compute the Pareto front of the multi-objective optimization problem we proceed by fixing the number of test products and defining a zero-one mathematical program that maximizes coverage. The approach presented here relates to the work by Arito et al. [13] for solving a multi-objective test suite minimization problem in regression testing.

A zero-one program is an integer program in which the variables can only take values 0 or 1 [14]. The details of the algorithm applied are explained in Section IV. In this section we describe the zero-one program. Let us call  $n$  to the number of test products (that is fixed) and  $f$  to the number of features of the FM. We will use the set of decision variables  $x_{i,j} \in \{0, 1\}$  where  $i \in \{1, 2, \dots, n\}$  and  $j \in \{1, 2, \dots, f\}$ . Variable  $x_{i,j}$  is 1 if product  $i$  has feature  $j$  and 0 otherwise. Not all the combinations of features form valid products. Following [12], we can express the validity of any product in an FM as a boolean formula. These boolean formulas can be expressed in Conjunctive Normal Form (CNF) as a conjunction of clauses, which in turn can be expressed as constraints in a zero-one program. The way to do it is by adding one constraint for each clause in the CNF. Let us focus on one clause and let us define the Boolean vectors  $v$  and  $u$  as follows [15]:

$$v_j = \begin{cases} 1 & \text{if feature } j \text{ appears in the clause,} \\ 0 & \text{otherwise,} \end{cases}$$

$$u_j = \begin{cases} 1 & \text{if feature } j \text{ appears negated in the clause,} \\ 0 & \text{otherwise.} \end{cases}$$

With the help of  $u$  and  $v$  we can write the constraint that corresponds to one CNF clause for the  $i$ -th product as:

$$\sum_{j=1}^f v_j (u_j (1 - x_{i,j}) + (1 - u_j) x_{i,j}) \geq 1 \quad (1)$$

As an illustration, in the GPL model let us suppose that *Search* is the 8-th feature and *Num* is the 12-th one. The cross-tree constraint “*Num* requires *Search*” can be written in CNF with the clause  $\neg \text{Num} \vee \text{Search}$  and translated to a zero-one constraint as:  $1 - x_{i,12} + x_{i,8} \geq 1$ .

Our focus is pairwise coverage. This means that we want for each pair of features to cover 4 cases: both unselected, both selected, first selected and second unselected and vice versa. We introduce one variable in our program for each product, each pair of features and each of these four possibilities. The variables, called  $c_{i,j,k,l}$ , take value 1 if product  $i$  covers the pair of features  $j$  and  $k$  with the combination  $l$ . The combination  $l$  is a number between 0 and 3 representing the selection configuration of the features according to the next mapping:  $l = 0$ , both unselected;  $l = 1$ , second selected and first unselected;  $l = 2$ , first selected and second unselected; and  $l = 3$  both selected. The values of the variables  $c_{i,j,k,l}$  depend on the values of  $x_{i,j}$ . In order to reflect this dependence in the mathematical program we need to add the following

constraints for all  $i \in \{1, \dots, n\}$  and all  $1 \leq j < k \leq f$ :

$$2c_{i,j,k,0} \leq (1 - x_{i,j}) + (1 - x_{i,k}) \leq 1 + c_{i,j,k,0} \quad (2)$$

$$2c_{i,j,k,1} \leq (1 - x_{i,j}) + x_{i,k} \leq 1 + c_{i,j,k,1} \quad (3)$$

$$2c_{i,j,k,2} \leq x_{i,j} + (1 - x_{i,k}) \leq 1 + c_{i,j,k,2} \quad (4)$$

$$2c_{i,j,k,3} \leq x_{i,j} + x_{i,k} \leq 1 + c_{i,j,k,3} \quad (5)$$

Variables  $c_{i,j,k,l}$  inform about the coverage in one product. We need new variables to count the pairs covered when all the products are considered. These variables are called  $d_{j,k,l}$ , and take value 1 when the pair of features  $j$  and  $k$  with combination  $l$  is covered by some product and 0 otherwise. This dependence between the  $c_{i,j,k,l}$  variables and the  $d_{j,k,l}$  variables is represented by the following set of inequalities for all  $1 \leq j < k \leq f$  and  $0 \leq l \leq 3$ :

$$d_{j,k,l} \leq \sum_{i=1}^n c_{i,j,k,l} \leq n \cdot d_{j,k,l} \quad (6)$$

Finally, the goal of our program is to maximize the pairwise coverage, which is given by the number of variables  $d_{j,k,l}$  that are 1. We can write this as:

$$\max \sum_{j=1}^{f-1} \sum_{k=j+1}^f \sum_{l=0}^3 d_{j,k,l} \quad (7)$$

The mathematical program is composed of the goal (7) subject to the  $4(n+1)f(f-1)$  constraints given by (2) to (6) plus the constraints of the FM expressed with the inequalities (1) for each product. The number of variables of the program is  $nf + 2(n+1)f(f-1)$ . The solution to this zero-one linear program is a test suite with the maximum coverage that can be obtained with  $n$  products.

#### IV. ALGORITHM

The algorithm we use for obtaining the optimal Pareto set is given in Algorithm 1. This algorithm takes as input the FM and provides the optimal Pareto set. It starts by adding to the set two solutions that are always in the set: the empty solution (with zero coverage) and one arbitrary solution (with coverage  $C_2^f$ , number 2-combinations of the set of features). After that it enters a loop in which successive zero-one linear programs are generated for an increasing number of products starting at 2. Each mathematical model is solved using an extended SAT solver: *MiniSat+*<sup>3</sup>. This solver provides a test suite with the maximum coverage. This solution is stored in the optimal Pareto set. The algorithm stops when adding a new product to the test suite does not increase the coverage. The result is the optimal Pareto set.

#### V. EXPERIMENTS

This section describes how the evaluation was carried out and its scalability analysis. The experimental corpus of our evaluation is composed by a benchmark of 118 feature models, whose number of products ranges from 16 to 640 products, that are publicly available from the SPL Conqueror [16] and the SPLOT [17] repositories. The objectives to optimize are the

---

#### Algorithm 1 Algorithm for obtaining the optimal Pareto set.

---

```

optimal_set ← {∅};
cov[0] ← 0;
cov[1] ←  $C_2^f$ ;
sol ← arbitraryValidSolution(fm);
i ← 1;
while cov[i] ≠ cov[i − 1] do
  optimal_set ← optimal_set ∪ {sol};
  i ← i + 1;
  m ← prepareMathModel(fm,i);
  sol ← solveMathModel(m);
  cov[i] ← |sol|;
end while

```

---

number of products required to test the SPL and the achieved coverage. It is desirable to obtain a high value of coverage in a low number of products to test the SPL, so they are conflicting objectives. Additionally, as performance measure we have also analyzed the time required to run the algorithm, since we want the algorithm to be as fast as possible. For comparison these experiments were run in a cluster of 16 machines with Intel Core2 Quad processors Q9400 at 2.66 GHz and 4 GB running Ubuntu 12.04.1 LTS managed by the HT Condor 7.8.4 manager. Each experiment was executed in one core.

We computed the Pareto optimal front for each model. Figure 2 shows this front for our running example GPL, where the total coverage is obtained with 12 products, and for every test suite size the obtained coverage is also optimal. As our approach is able to compute the Pareto optimal front for every feature model in our corpus, it makes no sense to analyze the quality of the solutions. Instead, we consider more interesting to study the scalability of our approach. For that, we analyzed the execution time of the algorithm as a function of the number of products represented by the feature model as shown in Figure 3. In this figure we can observe a tendency: the higher the number of products, the higher the execution time. Although it cannot be clearly appreciated in the figure, the execution time does not grow linearly with the number of products, the growth is faster than linear.

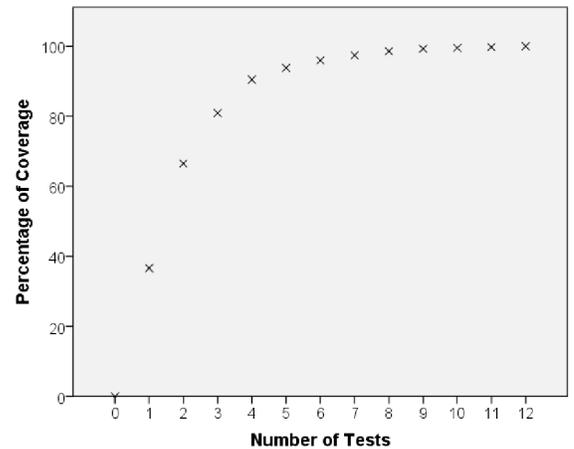


Fig. 2. Pareto optimal front for our running example (GPL).

<sup>3</sup>Available at URL: <http://minisat.se/MiniSat+.html>

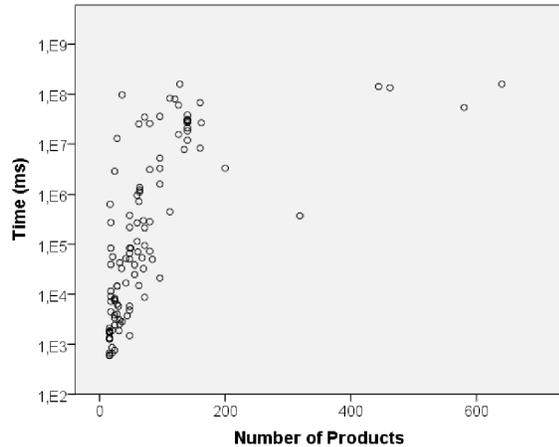


Fig. 3. Time (log scale) required to find optimal Pareto set against the number of products of the feature models.

In order to check our intuition, we have performed a Spearman's rank correlation test. This test's coefficient  $\rho$  takes into account the rank of the samples instead of the samples themselves. The correlation coefficient between the execution time and the number of products denoted by a feature model is 0.831. This is a very high value that confirms our expectations, the higher the number of products, the higher the execution time of the algorithm. We also computed the Spearman's rank correlation for the execution time against the number of features of the feature models which was quite lower (0.407). This is because two feature models with the same number of features could denote significantly different number of products depending on the constraints derived from the relationships between the features. In summary, the best indicator of the execution time of our approach is the number of products denoted by a feature model.

## VI. CONCLUSIONS AND FUTURE WORK

We have proposed an approach to exactly obtain the optimal Pareto set of the multi-objective SPL pairwise testing problem. We defined a zero-one linear mathematical program and an algorithm based on SAT solvers for obtaining the optimal Pareto set. By construction the solution obtained using this approach is optimal and could serve as reference for measuring the quality of the solutions proposed by approximated methods.

The evaluation revealed a generally large runtime for our feature models. This fact prompted us to analyze the impact of the number of products and number of features in runtime. We found a high correlation in the first case and a low correlation in the second case. As a result of this finding our future work is twofold. First, we want to streamline the mathematical program representation in order to reduce the runtime of the algorithm. We observed that some of the constraints can be redundant. For instance, features that are selected in all the products of the product line do not need a variable since they are valid for any product. Similarly, there are pairs of feature combinations, that is  $C_{i,j,k,l}$  variables, that are not valid according to the feature model and hence can be eliminated [18]. We also noticed that removing some

of the redundant constraints can increase the runtime, while adding more constraints could help the SAT solver search for a solution. We plan to study the right balance of both reducing and augmenting constraints. Second, we will look at larger feature models to further study the scalability of our approach.

## ACKNOWLEDGEMENTS

Funded by Austrian Science Fund (FWF) project P21321-N15 and Lise Meitner Fellowship M1421-N15, the Spanish Ministry of Economy and Competitiveness and FEDER under contract TIN2011-28194 and fellowship BES-2012-055967.

## REFERENCES

- [1] P. Zave, "Faq sheet on feature interaction," <http://www.research.att.com/pamela/faq.html>.
- [2] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [3] E. Engström and P. Runeson, "Software product line testing - a systematic mapping study," *Information & Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.
- [4] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information & Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.
- [5] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011. [Online]. Available: <http://doi.acm.org/10.1145/1883612.1883618>
- [6] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.
- [7] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines," *CoRR*, vol. abs/1211.5451, 2012.
- [8] A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *ISSRE*, T. Dohi and B. Cucik, Eds. IEEE, 2011, pp. 120–129.
- [9] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*, 1st ed. Wiley, June 2001.
- [10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
- [11] R. E. Lopez-Herrejon and D. S. Batory, "A standard problem for evaluating product-line methodologies," in *GCSE*, ser. Lecture Notes in Computer Science, J. Bosch, Ed., vol. 2186. Springer, 2001, pp. 10–24.
- [12] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.
- [13] F. Arito, F. Chicano, and E. Alba, "On the application of sat solvers to the test suite minimization problem," in *SSBSE*, ser. Lecture Notes in Computer Science, G. Fraser and J. T. de Souza, Eds., vol. 7515. Springer, 2012, pp. 45–59.
- [14] L. A. Wolsey, *Integer Programming*. Wiley, 1998.
- [15] A. M. Sutton, L. D. Whitley, and A. E. Howe, "A polynomial time computation of the exact correlation structure of k-satisfiability landscapes," in *Proceedings of GECCO*, 2009, pp. 365–372.
- [16] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption," *Information & Software Technology*, vol. 55, no. 3, pp. 491–507, 2013.
- [17] "Software Product Line Online Tools (SPLOT)," 2013, <http://www.splot-research.org/>.
- [18] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Using feature model knowledge to speed up the generation of covering arrays," in *VaMoS*, S. Gnesi, P. Collet, and K. Schmid, Eds. ACM, 2013, p. 16.

# Which Feature Location Technique is Better?

Emily Hill<sup>1</sup>, Alberto Bacchelli<sup>2</sup>, Dave Binkley<sup>3</sup>, Bogdan Dit<sup>4</sup>, Dawn Lawrie<sup>3</sup>, Rocco Oliveto<sup>5</sup>

<sup>1</sup>Montclair State University, Montclair, NJ, USA

<sup>2</sup>University of Lugano, Switzerland & Delft University of Technology, The Netherlands

<sup>3</sup>Loyola University Maryland, Baltimore, MD, USA

<sup>4</sup>The College of William and Mary, Williamsburg, VA, USA

<sup>5</sup>University of Molise, Pesche (IS), Italy

hillem@mail.montclair.edu, a.bacchelli@tudelft.nl, binkley@cs.loyola.edu,

bdit@cs.wm.edu, lawrie@cs.loyola.edu, rocco.oliveto@unimol.it

**Abstract**—Feature location is a fundamental step in software evolution tasks such as debugging, understanding, and reuse. Numerous automated and semi-automated feature location techniques (FLT) have been proposed, but the question remains: How do we objectively determine which FLT is most effective? Existing evaluations frequently use bug fix data, which includes the location of the fix, but not what other code needs to be understood to make the fix. Existing evaluation measures such as precision, recall, effectiveness, mean average precision (MAP), and mean reciprocal rank (MRR) will not differentiate between a FLT that ranks higher these related elements over completely irrelevant ones. We propose an alternative measure of relevance based on the likelihood of a developer finding the bug fix locations from a ranked list of results. Our initial evaluation shows that by modeling user behavior, our proposed evaluation methodology can compare and evaluate FLT's fairly.

**Index Terms**—Feature location, Concern location, Relevance measures, Empirical studies

## I. INTRODUCTION

Feature location in software, also called concept or concern location, is the process of identifying the source code elements, such as methods or files, that implement a user-observable feature [1], [2]. Feature location is a fundamental step in software evolution tasks such as debugging, understanding, and reuse. Researchers have proposed numerous automated and semi-automated feature location techniques (FLT) [3], which usually recommend a ranked list of elements to be examined by a developer. Existing approaches to evaluating feature location techniques have predominantly used bug fixes automatically mined from source code repositories [3], [4]. The locations of the fix are likely relevant, but what about the code that must be understood to implement the fix?

Let us make the parallel that navigating a software system's source code is like traveling the globe. You are a developer in New York, trying to reach San Francisco, where you need to fix a bug. Recommender *A* gives you directions to St. Petersburg, Russia, which you immediately recognize as bad directions to an unrelated continent. Recommender *B* gives you directions to Los Angeles, which you follow, eventually realizing that you are still not that close to San Francisco. Finally, Recommender *C* gives you directions to Berkeley, and from there you can successfully navigate to your final destination. If Recommender *C* had been your first choice, you

would have saved navigation time. This parallel illustrates the plight of a developer using FLT's in trying to locate source code during a software maintenance task in unfamiliar software. In this paper, we propose a way to differentiate between FLT's that send developers to Russia over Berkeley.

Going back to code, consider the following scenario with two FLT's, *A* and *B*. In the top 5 recommended code elements, FLT *A* recommends 4 methods that would need to be understood to implement the fix, and one fix location at rank 5. FLT *B* also recommends the fix location at rank 5, but the methods at ranks 1 through 4 are irrelevant (*i.e.*, are part of unrelated areas of the software system). Although *A* is intuitively better than *B*, the current measures commonly used to compare FLT's, *e.g.*, precision, recall, effectiveness, mean average precision (MAP), or mean reciprocal rank (MRR) [5], would mark them as equally effective.

An alternative to bug fix data is to have developers manually locate features in code [6], [7]. This process can be dependent on the annotators, and may change depending on the expertise, experience, and primary evolution task being undertaken. Prior attempts to create such gold sets have found very little agreement between annotators [7], which is also supported by anecdotal evidence from the authors' experiences in gathering gold sets. The main advantage of bug fix data is its objectivity: given a bug report, the bug fix confirms code locations that were actually changed to implement a modification to the feature as described by the bug report. Nevertheless, by using bug fix data we are faced with the challenge of fairly evaluating our hypothetical FLT's *A* and *B*. We support the use of bug fix data to objectively evaluate FLT's, but we propose an alternative measure of relevance based on the *likelihood* of a developer finding the fix locations.

In this paper, we introduce *rank topology*, a metric for comparing FLT's. We evaluate our proposed measure by running a FLT based on a state of the art Information Retrieval (IR) technique with different randomly generated lists of results that have the same ranks for the relevant code elements but contain other important code elements at different ranks. Our goal is to determine whether our rank-topology metric can distinguish between different lists where the relevant documents (*i.e.*, bug fixes) have exactly the same ranks.

## II. THE RANK TOPOLOGY APPROACH

Our goal is to distinguish between FLTs that rank higher documents closely related to the bug fix than completely unrelated ones. Ideally we would like to mimic the behavior of a typical developer navigating the source code using the search results as a starting point [8], [9]. However, the accuracy of our estimates must be balanced with ease of use: if an evaluation metric is too complicated or difficult to use, it will not be widely adopted by the research community.

Thus, we propose a simple model of developer behavior. We assume the developer starts at the top of the ranked list of documents and investigates them in sequential order. At each rank, the developer has a choice of whether or not to explore the structural topology of the current document. As the developer explores the structural topology, she is constantly faced with the decision to continue forward or go back to the ranked list. In the following subsections, we formalize this decision making process and provide our current solution.

### A. Proposed Rank Topology Framework

To help describe and compare FLTs, we introduce a two-part framework that provides a template for evaluating a FLT by dividing the search into two phases. In the first phase a developer considers, in rank order, the entries of the ranked list returned as the result of a search. When considering an entry, the developer may have some impression of the following  $k$  entries. For example, if when considering  $e_2$ , the next entry  $e_3$ , has a much lower score, then the developer may have the impression that it is worth considering  $e_2$  longer than average, since the next entry appears irrelevant.

Thus, phase 2, which consists of the investigation of an entry  $e_i$  from the ranked list, begins with  $e_i$  and some impression of the subsequent entries (e.g.,  $e_{i+1}$ ) in the ranked list. Phase 2 considers  $e_i$  and the code artifacts connected to it. The connections are captured as a set of relations (e.g., callees, callers, textual similarity, within the same file, (data) depends on, etc.). Each relation can have a weight associated with it. For example, callers might be valued more than callees. We make the simplifying assumption that developers do not revisit a code artifact; thus, they explore a subtree of a spanning tree of the graph defined by the structural relations.

At each code artifact, one of the following two decisions is made: bail or next. If the developer opts to bail on the current structural search, they return immediately to the next entry in the ranked list. Alternatively, the next code artifact can be chosen from the set of structurally connected code artifacts to the structural path being traversed. We assume the developer only chooses to investigate code artifacts that appear “interesting” (i.e., that have a high enough value). In either case the impression (of the value of bailing) is updated. This value can be increased if nothing useful is being encountered or it can be decremented if phase 2 is going well.

### B. Modeling an Imperfect User

One of the challenges in using a rank topology measure is in determining how *smart* the developer is. An *omniscient*

developer would make no wrong choices and explore every profitable structural edge no matter how tenuously related to the bug description. For example, consider JabRef<sup>1</sup>, an open source Java bibliography reference manager, and its bug #1297576 with query “*Printing of entry preview*”. All 20 of the fix locations are just 4 or 5 structural “hops” (i.e., edges) away from the very first result, the `actionPerformed` method in the `PrintAction` class. While the `PrintAction` class seems relevant to the bug, and it is understandable that some of the relevant documents are located nearby in the program structure, some of the links are not obvious. For example, navigating these structural paths requires navigating through a generic `openWindow` method that launches a number of actions within JabRef, and then to the generic `get(String)` method in the `JabRefPreferences` class. The presence of these generic, highly-connected methods makes it trivial for an omniscient user to navigate to most bug fix locations within 4-6 structural hops of the top ranked document. Although most developers are not omniscient, the perfect omniscient user provides an upper bound on the limits of structural topology in finding fix locations.

Developers who are not omniscient will not make generic structural hops unless there is strong evidence to do so. Prior work applying information foraging theory to developer behavior has shown that simple textual information, such as the cosine similarity between a method and a query, closely models actual developer behavior when debugging [8]. Thus, we propose a semi-intelligent user that only follows a structural link if the next method exhibits textual clues.

### C. The Proposed Rank Topology Metric

For each bug fix location, we determine the shortest number of hops required to find it in terms of structural topology and the ranked list. The result set is the minimum cost of navigating to each fix location. For simplicity, in the current work the cost of each rank jump and each structural edge is one. If a developer navigates the structural topology from a method to another method in the same class, we assign a cost of two (one to navigate to the class and one to get to the desired method). Textual information can be used to approximate whether a developer would recognize a structural link and follow it. If the textual relevance to the query is below a certain threshold (i.e., if the developer is unlikely to explore it), the structural edge is ignored.

In this initial work, we make the simplifying assumption of exploring every ranked document, and not considering all the wrong explorations before arriving at a fix location. For example, a fix location may be 5 structural hops away from the third ranked document, giving a total cost of 8. This distance does not consider all the elements explored with rank 1 or 2 before arriving at rank 3 when calculating the cost; it only considers the number of elements to reach the current rank in the list (3, in this case) and the direct structural cost of the path to the fix. The inverse of this cost forms our metric.

<sup>1</sup><http://jabref.sourceforge.net/>

Our proposed *rank topology metric* is inspired by average precision (AP), which is commonly used to calculate MAP scores in evaluating IR systems [5]. Like AP, our metric involves the average of the inverse ranks for each relevant document. However, unlike the AP score, the ranks used are based on the costs using structural topology and rank.

#### D. Current Implementation

To implement the structural topology, we experimented with using method call and type hierarchy information as undirected graphs. Call graphs were extracted using Eclipse’s statically available call hierarchy functionality. Next, we used JGraphT’s implementation (<http://jgrapht.org/>) of the Floyd-Warshall algorithm to find all shortest paths from the bug fix locations to every other method in the program. We use a very simple form of textual information, cosine similarity, to determine textual relevance of each method to the query. The cosine similarity is computed using the Vector Space Model (VSM) where term weights are calculated using term frequency-inverse document frequency (tf-idf). All of these components to the rank-topology metric are available by third party libraries, and need not be implemented from scratch.

### III. PRELIMINARY EVALUATION

Our premise is that existing IR measures such as precision, recall, and MAP cannot differentiate between techniques that rank related versus irrelevant documents above the fix locations. To evaluate if our proposed rank-topology measure captures this distinction, we compare a state of the art IR technique with a randomly ordered list of methods in the program with the same relevant fix locations at the exact same ranks as the IR technique. Thus, we control that the only source of variation is in the ordering of the non-relevant methods (*i.e.*, the methods not involved in the bug fix). By crafting the randomly ordered results in this way, we guarantee the results are indistinguishable by current evaluation measures.

We selected a small set of bugs from the JabRef program in the SEMERU dataset [3], [10]. To determine if our rank topology metric depends on how relevant results are initially ranked, we selected bugs from three different types of ranked lists: one where relevant documents appeared in the top five, one where relevant documents appeared from five to ten, and one where relevant documents first appeared around rank 100. We selected two queries (*i.e.*, bugs) of each type. None of the selected bugs ranked a relevant document in the first position.

The retrieval method used to produce these ranked lists was the Query Likelihood Model (QLM). The model was selected because it does not use VSM, which underlies our rank-topology technique of modeling the imperfect user. Instead, QLM relies on language models to estimate the probability that a document generates a query. Documents with higher probabilities of generating the query are ranked higher. Prior work showed that this model performs well in FLTs [4].

#### A. Effect of Structural Topology on an Omniscient User

One of the parameters to the rank topology approach is what structural information to explore to find fixes in close

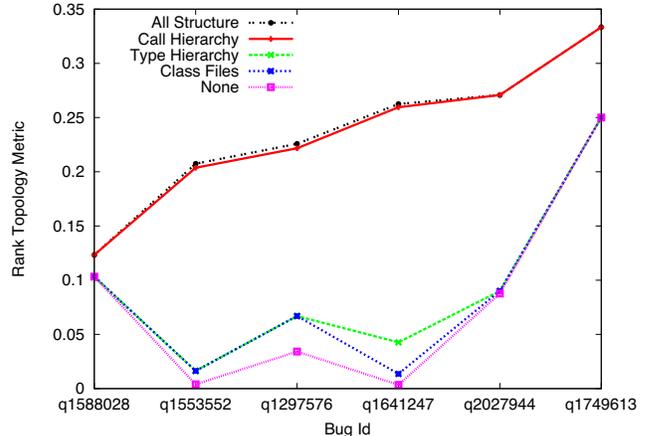


Figure 1. Effect of program structure on the rank topology metric for each JabRef bug used in the case study. Caller information has the biggest impact on finding bug fixes structurally close to the retrieved results.

proximity to the ranked entries. When navigating source code, developers have access to caller and callee information in the call hierarchy, can jump to other classes in the type hierarchy, or can jump through its class to other methods within the same class file. Before investigating whether our rank topology metric can distinguish between random and QLM, we first explore the effect that different structural topology can have on the results.

Figure 1 shows the rank topology (RT) measure for the six JabRef bugs in our case study, sorted by increasing RT scores when using all structural information. The pink line at the bottom uses no structural information, and represents the RT scores for the original raw document ranks. Caller and callee information is combined into the call hierarchy. The class files structure adds an edge to the topology from every class to every method, allowing a method to jump to any other method in the same class using 2 hops. Finally, type hierarchy information adds to the edges introduced by class files with additional edges between super and subclasses.

As highlighted by Figure 1, call information is the dominant connector among ranked documents and bug fix locations. Adding class and type hierarchy information results in a very small improvement overall. Although in the remaining experiments we use all the structure information, the results would be similar if we used only call information.

#### B. Modeling the Imperfect User

The purpose of our case study is to determine if a rank-topology measure can distinguish between QLM and a randomly ordered list of results with exactly the same relevant documents at the same ranks. Figure 2 shows the rank topology metric for each bug in the study. The top line in the figure represents an omniscient user that finds the shortest possible path through the structural topology, irrespective of textual clues. The solid red line below this represents the state of the art IR technique, QLM, with a textual threshold based on

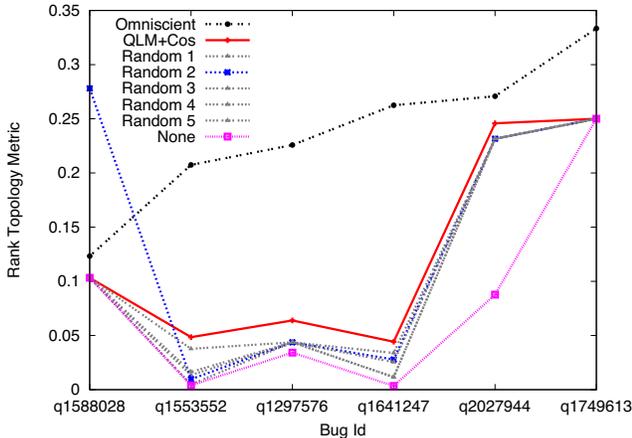


Figure 2. For each JabRef bug used in the case study, the proposed rank topology metric can differentiate between a state of the art IR technique (QLM+Cos) and 4 randomly ordered rank lists (1–5). QLM and random would be indistinguishable using existing IR measures, and would appear as the “None” line, which only includes ranks with no topology.

the VSM cosine similarity. A structural edge is only added to the topology if the similarity of both methods is greater than the median of all the scores returned for that query (*i.e.*, bug). We selected the median, rather than a raw threshold such as .5, because the actual range of cosine values is highly dependent on the nature of the query. We assume that if a method’s relevance score is over the threshold, the edges to related classes (in the Files and Type Hierarchy structures) also exist.

The next 5 dotted lines in gray and blue are randomly generated lists, which also use the median cosine score to prune the structural topology. The pink bottom line uses the raw ranks alone with no structural topology, labelled “None”. As expected, Figure 2 illustrates that our proposed rank topology metric differentiates between a randomly generated list of results and QLM. One of the randomly generated lists, Random 2 (in blue), significantly outperforms even an omniscient user exploring QLM’s rankings for one bug. These results demonstrate that traditional IR measures such as precision, recall, MRR, and MAP can give a misleading picture of the effectiveness of a feature location technique, when a technique can produce a ranked list that allows the developer to navigate to a fix location quicker. For future work, we would like to study actual user navigation habits, to see if our proposed rank topology metric accurately reflects differences between ranked lists for software maintenance.

#### IV. RELATED WORK

Existing FLT’s can be classified by the underlying information they use, such as static, dynamic, textual, or historical information [3]. FLT’s based on static analysis examine structural information such as control or data flow dependencies, and may incorporate textual information in comments and identifiers in the code. One such technique uses information foraging theory to analyze how developers navigate source

code when fixing bugs [8]. Although our rank topology metric is inspired by the same theory, our aim is to quantify the effort required by developers when analyzing the suggestions of a FLT by defining a new measure of relevance that can better analyze and compare the performance of FLT’s.

Another closely related work by Petrenko and Rajlich [11] is similar to the evaluation metric proposed in this paper. Specifically, the authors propose using an IR technique to identify bug fix locations. If the IR technique fails to identify a fix location, the developer can decide either to reformulate the query, or to identify a “focus method” (*i.e.*, a method related to the fix). In the latter case, the developer navigates program dependencies starting from the “focus method” to identify fix locations. In our current work, we use a similar infrastructure but from a different perspective. Whereas Petrenko and Rajlich [11] use dependencies to speed up the process of feature location and avoid query reformulation, we analyze dependencies to better compare and assess FLT’s in a semi-automatic usage scenario.

#### V. CONCLUSION AND FUTURE WORK

In this paper we presented a *rank topology metric* to fairly compare feature location techniques (FLT’s). In a small case study, we demonstrated that our rank topology metric can differentiate between a state-of-the-art IR technique and a randomly ordered list of results with the same relevant results at the exact same ranks. These two techniques would be indistinguishable using existing IR measures commonly used to evaluate FLT’s. For future work we will study how closely the proposed rank topology measure mimics an actual developer in practice when using FLT’s during corrective maintenance. We also plan to investigate the feasibility of more closely modeling user navigation behavior from source code search results.

#### REFERENCES

- [1] T. J. Biggerstaff, B. G. Mitbender, and D. Webster, “The concept assignment problem in program understanding,” in *ICSE*, 1993.
- [2] V. Rajlich and N. Wilde, “The role of concepts in program comprehension,” in *IWPC*, 2002.
- [3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: A taxonomy and survey,” *J. Soft. Maint. Evo.: Research & Practice*, vol. 25, no. 1, pp. 53–95, 2013.
- [4] S. Rao and A. Kak, “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models,” in *MSR*, 2011.
- [5] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY: Cambridge University Press, 2008.
- [6] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using natural language program analysis to locate and understand action-oriented concerns,” in *AOSD*, 2007.
- [7] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, “An empirical study of the concept assignment problem,” School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3, Jun. 2007, <http://www.cs.mcgill.ca/~martin/concerns/>.
- [8] J. Lawrence, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming, “How programmers debug, revisited: An information foraging theory perspective,” *IEEE TSE*, vol. 39, no. 2, 2013.
- [9] A. von Mayrhauser and A. M. Vans, “Program understanding behavior during debugging of large scale software,” in *ESP Workshop*, 1997.
- [10] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. H. Kagdi, “A dataset from change history to support evaluation of software maintenance tasks,” in *MSR*, 2013.
- [11] M. Petrenko and V. Rajlich, “Concept location using program dependencies and information retrieval (DepIR),” *Inf. Softw. Tech.*, vol. 55, no. 4, 2013.

# Automatic Means of Identifying Evolutionary Events in Software Development

Siim Karus

Institute of Computer Science  
University of Tartu  
Estonia  
siim.karus@ut.ee

**Abstract**—The software development process patterns in open source software projects are not well known. Consequently, the longevity of new open source software projects is left up to subjective experiences of the development team. In this study, we are investigating a data mining approach for identifying relevant patterns in software development process. We demonstrate the capabilities of wavelet analysis on 27 open source software projects for identifying similar evolutionary patterns or events in different projects. The analysis identified close to 1000 evolutionary patterns common to multiple projects. The analysis of some of the patterns shows that the end of source code evolution of a project is determined early in the project. In addition, strong fluctuations of activity in sequential periods are identified as good indicators of problems in projects. In conclusion, the analysis reveals that wavelet analysis can be a powerful and objective tool for identifying evolutionary events that can be used as estimation basis or management guide in software projects.

**Keywords**—software evolution; wavelets; analysis; data mining; estimation; management

## I. INTRODUCTION

Open source software (OSS) repositories are a popular source of data for studying software projects. The bulk of mining software repositories research is concerned with defect detection and analysis of object-oriented code, which are generally easy to generalise to closed source projects. Mining processes and evolution of these projects has proven to be far more challenging.

In closed source commercial projects, many evolutionary milestones and events are dictated by the management. These events can be used to identify project scale and progress, which allows comparing or co-analysing different projects. Unfortunately, the evolutionary events in open-source software projects have been mostly theoretical with little empirical validation. This also means that different projects can not be compared in the same scale. In this study, we are proposing means for detecting evolutionary events in OSS projects to offer comparable data on their evolution.

The problem of temporal comparability and scaling is also common in economics and signal processing [1,2,3,4,5]. A means to solve the problem in those fields is applying a wavelet transform on the data. The idea behind a wavelet transform is that we can sample the same signal at different intervals giving us a natural means for scaling it. In case of evolution, the signal can be any measurable property of an evolving entity. This allows the wavelet transform to be applied directly for mining software repositories.

In order to understand OSS project evolution, we need to look at old projects, which have ceased evolving. We note that we are not looking at the vitality of OSS [6,7] as the projects' ability to provide support and grow in the number of releases. In this study, we only look at the development process ignoring documentation and community support.

We also need to gather data that is integrally sound and available across projects. This means the repository systems that have not changed for a long time are preferred. Therefore, data from source code repositories is chosen for the study for their historical availability and reliability.

Having reliable, mostly uniformly interpretable data allows us to use formal methods to identify objective and verifiable indicators of projects' progress and factors supporting long-living projects. We aim to offer these objective insights by trying to answer the following question:

RQ. Can we use wavelet analysis to find objective warning signs in software projects leading to the end of code evolution?

Knowing the warning signs could help in:

- Choosing the open source project to implement in business scenario – evolving source code is deemed to signify good health of a software project.
- Making timely preparations for decommissioning an OSS project – knowing that a project is about to reach maturity or demise, effort can be channelled to other project related activities like promotion or developing a successor alternative.
- Choosing development process that best suites the aims of a software project for new OSS projects – events contributing to early maturity or demise can be used to guide a project to earlier or later completion.

The paper starts by giving an overview of the wavelet analysis method used in the study in Section II. The dataset is explained in Section III and the analysis pipeline in Section IV. In Section V we present the related work. Finally, the results are highlighted and ideas for future are presented in Section VI.

## II. WAVELET ANALYSIS

Wavelet analysis is analysis of signals (time series) by decomposing the signal into wavelet coefficients (also known as shift coefficients) and scaling coefficients based on wavelet functions (aka filters). The decomposition can be repeated on the scaling coefficients until the number of resulting wavelet coefficients is smaller than the filter length.

In this study we chose to use a Daubechies filter of length 2 (also known as Haar wavelet) [5] due to its simplicity and simple interpretation. We apply discrete wavelet transform meaning we use discrete shift when matching the series with the wavelet. The scaling coefficients of discrete Haar wavelet transform can be interpreted as a smoothed curve of the series. The wavelet coefficients show temporal variations.

Wavelet transform has proven important in signal processing thanks to its inherent properties which allow comparisons at different scales and shifts. Compared to other time series analysis techniques, the main advantages are:

- Scaling coefficients allow fuzzy matching as differences in details are “smoothed out”.
- Filter coefficients allow detection of small anomalies in series.
- Transform levels make series of different lengths or scale comparable.

These advantages have been beneficial in financial analytics for identification of anomalies and correlations to identify opportunities [1,2,4]. The fuzzy matching and scale comparisons have proven useful for clone detection in image processing [3,8]. The advantages of wavelet analysis techniques are also useful for frequent pattern analysis of time series data like used in this study.

### III. DATA

In this study, we used extracted 2 time series and 16 data series from 27 OSS projects.

#### A. Projects

The analysis was performed on 27 OSS projects. 18 of these projects were from a dataset of software project chosen randomly using Google Code Search from various repositories containing team projects employing different source code languages, team sizes, and project types. 15 of these projects are on-going and 3 have had no development activity since January 2009 (verified in January 2013). The alive projects have stayed alive for the minimum of 4 years. This dataset was verified to have source code and activity structure similar to the dataset of more than 400,000 OSS projects tracked by ohloh.net<sup>1</sup>. Thus, this dataset should represent the overall state of OSS development fairly well.

The dataset of 18 projects was complemented by 9 dead projects from sourceforge.com in order to balance the number of dead and alive projects in the study. These 9 projects were also used as an independent dataset for verifying some of the patterns found in the 18 project dataset.

We used specially built software to implement ETL (extract-transform-load). The software downloaded the projects' CVS and SVN repository data into a SQL Server database<sup>2</sup> and processed the history data to count lines of code (LOC) and code churn metrics. A commit log of the projects was exported from the SQL server for wavelet analysis.

---

<sup>1</sup> <http://www.ohloh.net/>

<sup>2</sup> <http://www.microsoft.com/en-us/sqlserver/>

#### B. Metrics

We conducted wavelet analysis in respect to two different time series dimensions: days since the first commit and cumulative code churn. Code churn is the sum of code added, modified and removed [9]. Those two time series were chosen due to their popularity in project process measurement frameworks. Additionally, future cumulative code churn can be estimated with reasonable accuracy based on project snapshots [10]. Thus, cumulative code churn as development progress measure combines some of the benefits of measuring progress in time spent and LOC of final code produced.

Even though, some solutions use lines of code in project snapshot to measure progress in software development, we consider it a bad practice as LOC is not monotonously growing in time. This measure can still be used comparing progress to estimated final size of the software code base.

Even though number of commits is also used as an alternative time series in some contemporary systems, it does not fit our requirements due to its inconsistent interpretation. Our analysis of code repositories revealed that some repositories counted each file and folder change as a separate commit while others combined them. Consequently, number of commits as time series would have led to inconsistent views.

The analysed data series are relating to the developers participating in the projects, code churn, and project size.

The metrics relating to developers are:

- Average number of active developers
- Cumulative number of developers
- Number of commits
- Relative team size (only dead projects)

The metrics relating to code churn are:

- Mean LOC added, modified, deleted, and churned per commit (4 metrics)
- Cumulative LOC added, modified, deleted, and churned per commit (4 metrics)
- Relative cumulative LOC churn (only dead projects)

The metrics relating to project size are:

- Mean LOC
- Mean number of files
- Relative progress by date (only dead projects)

In our study, lines of code is measured by counting all text lines including source code, comments, configuration settings, readme, and build files. This takes into account our previous findings showing that on average 4 different types of code are used in OSS project and plaintext or configuration files are a significant portion of that code [11].

### IV. METHOD

The analysis and data preparation was conducted in several steps: data aggregation, discrete waveform transform, similar region detection, similar region grouping.

In the first step, data series were aggregated along the two time series dimensions. For days since first commit, the data was aggregated in 7 day frames (corresponding to a week). For cumulative code churn, a frame of 1000 LOC was used instead.

In the second step, discrete wavelet transform with Daubechies filter with length 2 (also known as Haar filter) was applied on the data series. This gives us two coefficient vectors (wavelet and scaling coefficient) for each transform (compression) level. An example of such decomposition can be seen in Figure 1 where the lowest graph shows the original series and upper ones show the decompositions (scaling coefficients as line and filter coefficients as columns).

Linearly positively similar (maximum deviation 0.5%) maximal sub-sequences of the coefficient vectors were identified in the third step. That is, sequences  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$  were considered similar if there exist  $a \in \mathbb{R}$  and  $b \in \mathbb{R}$  where  $a \geq 0$  so that for every  $i \in \{1, 2, \dots, n\}$   $\frac{|ax_i + b - y_i|}{|y_i|} \leq 0.005$ . We were looking at sub-sequences of the minimum length of 3 as any two 2-value sequences are linearly similar (but not necessarily positively). We only looked for similarities in the same dimension and the same type of coefficient (for example, we did not look for similarities between cumulative LOC added filter coefficient vectors and number of commits scaling coefficient vectors).

The analysis and data aggregation was performed using R Statistics Suite<sup>3</sup> with “wavelets”, “zoo”, and “chron” packages. Package “wavelets” provides discrete wavelet transform, package “zoo” time series methods, and package “chron” extends support for date and time manipulations.

## V. RELATED WORK

Success in formal closed source in-firm software development projects is clearly defined by their financial success. However, this measure does not apply to many of the open source projects, which are community led and distributed freely. Success in OSS projects has been considered to be high level of end-user adoption [12], high software quality [13], or in other aspects [14]. Consequently, the research on the success of OSS projects is dispersed by the various definitions of success of OSS projects [15].

The economics field has inspired researchers to apply economic estimation models employing autoregression to make evolutionary estimations on software evolution [16,17]. These studies have been focusing on defect prediction and are generally looking at short term estimations.

Hindle et al used Fourier analysis and multifractal to discover presence of seasonal and multifractal events in software evolution [18,19]. In this paper, we are using wavelet analysis, which allows us to also capture non-periodic events and variable-scale periodic events, which Fourier analysis can not detect. We also identifying specific occurrences of evolution patterns not just their presence.

We used wavelet analysis to identify several repeating and self-encompassing patterns in OSS projects in a related study [20]. In this study are additionally looking at patterns that occur only once and look at the relative timing and predictive capabilities of pattern occurrences.

<sup>3</sup> <http://www.r-project.org/>

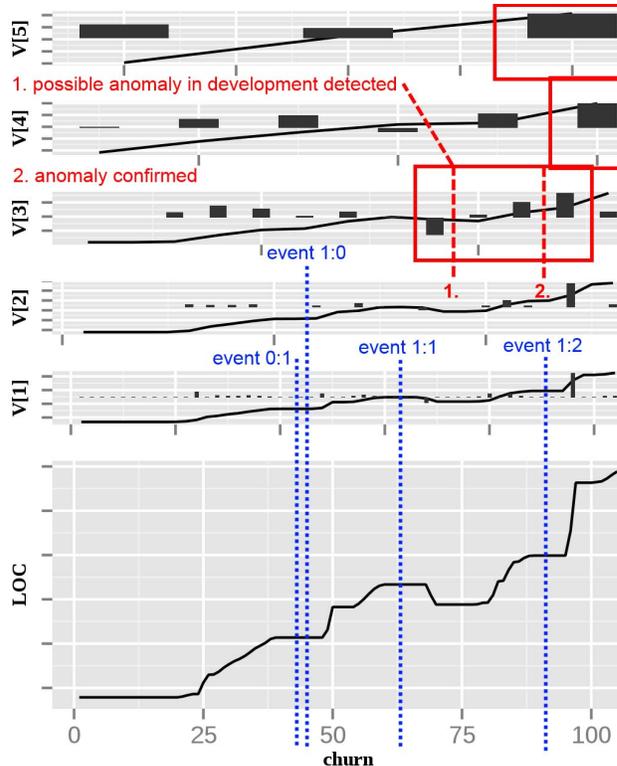


Figure 1. Example of wavelet transform levels (1 to 5), detected iteration events, and an anomaly in a project’s LOC series in churn timeseries.

## VI. RESULTS

The wavelet analysis revealed 998 common patterns across different projects. A quick analysis of these confirms that evolution patterns in software projects do express various warning signs leading to or indicating the impending end of code evolution in the project (RQ). Figure 1 shows an example of a project that contained iterative events. The dotted lines show when a specific event was detected and at what transform level it was detected. This and similar patterns confirm the iterative nature of projects.

The pattern segments also show that active projects have more frequent iterations with less code changed in a burst. In general, sudden large changes in project team or code are a sign of a dying project. Nevertheless, the projects can continue for several months (in several cases the project continued for more than 4 times the period it had lasted so far) after these bad patterns are fully formed. This offers limited evidence that open source projects which implement even activity development patterns and continuous engagement of new developers will keep the project alive. One should also keep in mind that an active engagement of new developers requires the software to be understandable to newcomers, highlighting the importance of documentation and clear design of the software. The ease of joining a project has been shown to be a factor contributing to higher end-user adoption as well [7].

The study also demonstrated the usefulness of wavelet analysis techniques to identify anomalies in software

projects. An example of anomaly detection can be seen on Figure 1 with dashed lines marking points when the algorithm detected the development of an anomaly (1.), confirmed it (2.). In the case shown on Figure 1, the project recovered from the anomaly (developers seeing different goals for the software) in time. This means that wavelet analysis can be successfully used to detect anomalies before they turn into a long-term issue allowing action to be taken before it is too late.

## VII. CONCLUSION

In conclusion, we have demonstrated the usefulness for both tracing software evolution and for detecting anomalies in software evolution. The events and issues detected by wavelets are useful for tracking development progress and answering questions like “Are the developers moving towards the same goal?” or “Is the project at risk of abandonment?” Thus, the wavelets give an automatically calculable and objective alternative for evaluating a project or setting up alarms for managers to address.

The patterns highlighted in this article are only a small subset of all the patterns identified during the study – many of the patterns and their interrelationships are still a subject to on-going studies. The patterns served in this paper are merely a validation of the wavelet technique used.

This study confirms wavelet analysis as an effective means for identifying evolution patterns in software project data. As wavelet analysis has been useful for data compression and clone identification even in case of inexact clones [3,8], it could prove useful for identification of bad “smells” and code clones in software projects. In addition, wavelet analysis could be applied to additional metrics in OSS projects like code complexity and class coupling.

## ACKNOWLEDGMENT

This research is partly funded by ERDF via the Estonian Centre of Excellence in Computer Science and is compiled with the assistance of the Tiger University Program of the Information Technology Foundation for Education.

## REFERENCES

- [1] Francis In and Sangbae Kim, "The Hedge Ratio and the Empirical Relationship between the Stock and Futures Markets: A New Approach Using Wavelet Analysis," *The Journal of Business*, vol. 79, no. 2, pp. 799-820, 2006.
- [2] António Rua and Luís C. Nunes, "International comovement of stock market returns: A wavelet analysis," *Journal of Empirical Finance*, vol. 16, no. 4, pp. 632-639, 2009.
- [3] Saiqa Khan and Arun Kulkarni, "Reduced Time Complexity for Detection of Copy-Move Forgery Using Discrete Wavelet Transform," *International Journal of Computer Applications*, vol. 6, no. 7, pp. 31-36, September 2010.
- [4] Jianhui Yang and Peng Lin, "Dynamic risk measurement of futures based on wavelet theory," in *Seventh International Conference on Computational Intelligence and Security (CIS)*, 2011, pp. 1484-1487.
- [5] Radomir S. Stanković and Bogdan J. Falkowski, "The Haar wavelet transform: its status and achievements," *Computers & Electrical Engineering*, vol. 29, no. 1, pp. 25-44, 2003.
- [6] U. Raja and M.J. Tretter, "Defining and Evaluating a Measure of Open Source Project Survivability," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 163-174, 2012.
- [7] Chandrasekar Subramaniam, Ravi Sen, and Matthew L. Nelson, "Determinants of open source software project success: A longitudinal study," *Decision Support Systems*, vol. 46, no. 2, pp. 576-585, 2009.
- [8] Yang Wang, K. Gurule, J. Wise, and Jun Zheng, "Wavelet Based Region Duplication Forgery Detection," in *Ninth International Conference on Information Technology: New Generations (ITNG)*, 2012, pp. 30-35.
- [9] J.C. Munson and S.G. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance, 1998.*, 1998, pp. 24-31.
- [10] Siim Karus and Marlon Dumas, "Code Churn Estimation Using Organisational and Code Metrics: An Experimental Comparison," *Information and Software Technology*, vol. 54, no. 2, pp. 203-211, February 2012.
- [11] Siim Karus and Harald Gall, "A study of language usage evolution in open source software," in *Proceedings of the 8th International Working Conference*, Honolulu, HI, USA, 2011, pp. 13-22.
- [12] Justin M. Beaver, Xiaohui Cui, Jesse L. St Charles, and Thomas E. Potok, "Modeling success in FLOSS project groups," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE09)*, Vancouver, British Columbia, Canada, 2009, p. 8.
- [13] C.A. Conley and L. Sproull, "Easier Said than Done: An Empirical Investigation of Software Design and Quality in Open Source Software Development," in *42nd Hawaii International Conference on System Sciences, 2009. HICSS '09.*, 2009, pp. 1-10.
- [14] Amir Hossein Ghapanchi, Aybuke Aurum, and Graham Low, "A taxonomy for measuring the success of open source software projects," *First Monday*, vol. 16, no. 8, 2011.
- [15] Kevin Crowston, James Howison, and Hala Annabi, "Information systems success in free and open source software development: theory and measures," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 123-148, 2006.
- [16] M. Goulão, N. Fonte, M. Wermelinger, and F. Brito e Abreu, "Software Evolution Prediction Using Seasonal Time Analysis: A Comparative Study," in *Proceedings in 16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 213-222.
- [17] A. Hmood, M. Erfani, I. Keivanloo, and J. Rilling, "Applying technical stock market indicators to analyze and predict the evolvability of open source projects," in *Proceedings in 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 613-616.
- [18] A. Hindle, M. W. Godfrey, and R. C. Hol, "Mining Recurrent Activities: Fourier Analysis of Change Events," in *Proceedings in 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 295-298.
- [19] A. Hindle, M.W. Godfrey, and R.C Holt, "Multifractal aspects of software development: NIER track," in *Proceedings in 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 968-971.
- [20] Siim Karus, "Measuring Software Projects Mayan Style," in *Proceedings in 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2013.

# Towards Understanding Large-Scale Adaptive Changes from Version Histories

Omar Meqdadi<sup>1</sup>, Nouh Alhindawi<sup>1</sup>, Michael L. Collard<sup>2</sup>, Jonathan I. Maletic<sup>1</sup>

<sup>1</sup>Department of Computer Science

Kent State University

Kent, Ohio, USA

{omeqdadi, nalhinda, jmaletic}@kent.edu

<sup>2</sup>Department of Computer Science

The University of Akron

Akron, Ohio, USA

collard@uakron.edu

**Abstract**—A case study of three open source systems undergoing large adaptive maintenance tasks is presented. The adaptive maintenance task involves migrating each system to a new version of a third party API. The changes to support the migration were spread out over multiple years for each system. The first two systems are both part of KDE, namely KOffice and Extragear/graphics. The adaptive maintenance task, for both systems, involves migrating to a new version of Qt. The third system is OpenSceneGraph that underwent a migration to a new version of OpenGL. The case study involves sifting through tens of thousands of commits to identify only those commits involved in the specific adaptive maintenance task. The object is to develop a data set that will be used for developing automated methods to identify/characterize adaptive maintenance commits.

**Keywords**—Adaptive Maintenance, Commit Types, Maintenance Classification.

## I. INTRODUCTION

Today's software systems have great longevity. We continually evolve and maintain them to assure their relevance and usefulness to their respective user communities. One direct side effect of longevity is that dependent platforms, compilers, libraries, frameworks, and APIs also change. Maintenance to address such changing dependent systems is termed adaptive. Adaptive-maintenance tasks involve changing a software system in response to changes in its environment. Examples include migrating a system to work on a new version of an operating system or to support a new API or library.

Adaptive maintenance is somewhat unique in that its cause is most often outside of the control of the organization/developer. That is, hardware changes, new versions of the compiler, and changes to APIs are enhancements that come from a third party. Additionally, adaptive-maintenance tasks are typically enterprise/system wide and impact large bodies of source code.

For example, many companies have recently (in the past couple years) migrated from Microsoft .Net 1.1/2.0 to .Net 3.5/4.0. This adaptive task has impacted thousands of organizations and millions of lines of code. The cost of this change is easily in the tens of millions of dollars.

Unfortunately, there is little (automated) tool support for adaptive maintenance tasks and the vast bulk of the work is done manually, thus adding to its overall cost. One of the authors' long-term research goals is to provide more automated support for the adaptive-maintenance process.

We feel one of the first steps to achieve this goal is to better understand what adaptive changes look like and how they are applied in actual software systems. There are no large studies that examine adaptive maintenance in any detailed or systematic manner. As such we undertook a case study of three open-source systems that previously underwent major adaptive maintenance tasks. We examined multiple years of version history of these systems to determine exactly which commits were involved in the particular adaptive-maintenance task. This inspection was done manually and involved reading system documentation, development notes, commit-log messages, and source code. It was a labor-intensive study that involved many person hours of work.

The result of the study was a categorization of all the commits (during a specific time frame) into either one of two categories, namely adaptive commits and non-adaptive commits. The adaptive commits involved changes to the system associated with the adaptive-maintenance task being examined. The non-adaptive commits were all other commits to the system that involved different maintenance tasks. All three of these systems continually underwent corrective maintenance and enhancements during the time of the adaptive-maintenance task. We then analyzed these adaptive-maintenance commits to identify any trends.

The remainder of this paper is organized as follows. First we describe the three systems used in the case studies and the reasons for selecting these systems. Next we present our findings from the manual investigation and describe the adaptive commits uncovered. Then an analysis of this data is presented along the measure of change size. Following is threats to validity, related work, and conclusions.

## II. CASE STUDY

The goal of this case study is simple. Examine the version history of a software system and determine which commits to the system are concerned with a specific adaptive-maintenance task. We selected three open-source systems that were known to have undergone a major adaptive change in their history. Knowing the adaptive-maintenance task allowed us to narrow down the time frame of version history to examine for each system.

We now present the details of the systems along with the reasons behind the choice of each and a brief background on version-control systems and the commit process. Lastly, we present the adaptive commits that were collected as an input for adaptive change analysis.

### A. Subject Systems

As a case study, we investigated and studied three large open source systems, the office-applications suite *KDE/KOffice*, graphical applications associated with the KDE project<sup>1</sup> in the package *KDE/Extragear/graphics*, and the high-performance 3D graphics toolkit *OpenSceneGraph* (OSG)<sup>2</sup>. These systems cover a number of application domains and sizes, and are prime examples of successful open-source development that utilizes a number of different APIs. Many of these third party APIs are regularly updated thus requiring migration to new API versions.

The two KDE packages we examined use Qt for the windowing and user interface. Qt is an open-source framework for developing graphical user interfaces and is widely used by many software projects. Qt uses standard C++ but makes broad use of the Meta Object Compiler, a special code generator, along with numerous macros to enhance C++.

The time frame selected for both KOffice and Extragear/graphics was from January 1, 2006 through December 31, 2010. In 2004 a major new version of Qt, namely Qt4, was released<sup>3</sup>, and in early 2006, the developers of the KDE project started to initiate the porting of all KDE systems to support Qt4 (i.e., move from Qt 3.x to Qt 4.0). The process of moving KDE completely to Qt4 lasted until the end of 2010, as indicated by developer commits. This included dealing with another substantial release of Qt, such as Qt 4.7. That is, we are looking for adaptive maintenance related to the migration of these two KDE packages from Qt3 to Qt4.

The third system, OSG, is written in C++ using the OpenGL specification. OpenGL<sup>4</sup> is a software interface to graphics hardware, which offers an abstract API for drawing 2D and 3D graphics and is widely used by a number of scientific visualization and modeling systems. In August 2008, a major new version of OpenGL, namely OpenGL 3, was released. The adaptive maintenance related to the migration to OpenGL 3 was completed at the end of 2010, as mentioned in the commit annotations by the OSG developers. Hence, we investigated the commits of OSG in the time period of August 1, 2008 through Dec. 31, 2010.

### B. Data Collection

The text message of a commit describes the change undertaken by the developers and the intended purpose of the change [2]. Therefore, with the text of a commit-log message we can, in most cases, determine if that commit is involved in an adaptive-maintenance task.

For the case study, we extracted the change-log file associated with each studied system using the *svn log* command. Then, we manually read and investigated each commit message in the change-log file to determine if

changes that occurred in the commit were adaptive changes to the specific APIs modifications we are investigating. Here, the adaptive changes involved porting from Qt 3.x to support Qt 4.x. The adaptive modifications are identified by searching the commit-log messages for indications that Qt3 features/interfaces were changed to support new features/interfaces found in Qt4. For instance, since the class *QPushButton* in Qt3 was replaced by the class *QAbstractButton* in Qt4, searching for these classes in the commit-log messages is a criterion indicating adaptive commits associated with this task. Lists of changing features and classes from Qt3 to Qt4 are listed in the system's online documentation. We used the same basic process to locate the relevant adaptive changes in OSG for the migration to OpenGL 3.

Manually identifying adaptive commits was a very costly task. For instance, during the time period of January 1, 2006 to December 31, 2010 there were over 38,000 commits (in subversion) to the KOffice project. Hence, *the manual investigation took several months* of arduous work to accurately differentiate the adaptive commits from non-adaptive.

### C. Summary of Findings

Our investigation shows that the vast majority of the commits during the studied time period had nothing to do with the API migrations. The majority of commits addressed other type of maintenance such as corrective, adding new features, or enhancements tasks going on in parallel. A summary of the obtained results is given in TABLE I.

It was a bit surprising how few actual commits were involved in these major API migrations. However, as we will show in the next section, the commits were on average quite large, compared to a typical commit [1], and each impacted a large number of files. Typically, a single adaptive commit involved addressing one part of the migration for the entire software system. That is, the migration process was done incrementally but system wide. The committer (developer) normally grouped the same types of adaptive transformations into a single large commit. Additionally, adaptive maintenance tasks normally do not involve large complex changes to functionality so often require few specific changes but applied in many locations.

TABLE II shows that the ratio of commits involved with these specific adaptive maintenance tasks decreased over time. Note that the OSG migration did not start until 2008. This further supports that the systems were incrementally migrated to new API versions, as the API versions were released, e.g., Qt 4.1, Qt 4.2 etc.

## III. CHARACTERIZING THE ADAPTIVE COMMITS

We now take a closer look at the three sets of adaptive commits and attempt to uncover any similar characteristics and trends. First the commits are categorized based on how

<sup>1</sup> See <http://www.KDE.org>.

<sup>2</sup> See [www.openscenegraph.org](http://www.openscenegraph.org).

<sup>3</sup> See <http://doc.qt.nokia.com/4.0/porting4.html>.

<sup>4</sup> See <http://www.opengl.org/registry/#oldspecs>.

they impact the system. That is, what was added, deleted, or modified in each commit.

TABLE I. ADAPTIVE AND NON-ADAPTIVE COMMITS FOR THE THREE SYSTEMS OVER THE GIVEN TIME PERIOD.

System	KOffice	Extragear/ graphics	OSG
<b>Adaptive Maintenance Task</b>	Migration from Qt3 to Qt4	Migration from Qt3 to Qt4	Migration to OpenGL 3
<b>Adaptive Task Starting-Date</b>	03/29/2006	11/07/2006	09/18/2008
<b>Adaptive Task Ending-Date</b>	12/31/2010	12/31/2010	12/31/2010
<b>Total Number of Commits</b>	38,980	26,336	4,310
<b>Number of Non-Adaptive Commits</b>	38,849 (99.7%)	26,117 (99.2%)	4,231 (98.2%)
<b>Number of Adaptive Commits</b>	131 (0.3%)	219 (0.8%)	79 (1.8%)

TABLE II. DISTRIBUTION OF ADAPTIVE COMMITS PER YEAR. PERCENTAGES ARE COMPARED TO TOTAL ADAPTIVE COMMITS

Year	Ratio of Adaptive Commits Per Year		
	KOffice	Extragear/ graphics	OSG
2006	40.5%	3.4%	0.0%
2007	24.2%	37.6%	0.0%
2008	14.1%	32.3%	50.3%
2009	11.8%	21.8%	37.1%
2010	9.4%	4.9%	12.6%

This size characteristic was selected for a variety of reasons. Given the raw data (version information) this measure is readily available from the log entries. There is little else that one can compute from the commits beyond this information. It has also been demonstrated [1] that the commit size and log message are correlated. Mockus and Votta [4] also discovered that the change size is essential to understanding why that change was performed. Hindle et al. [2] showed that the change size may be significant for predicting the type of a change.

To examine the size of the adaptive commits in the context of how many items were added, delete, or modified within each commit, the following three size-based measures were used:

- *Method*: Number of methods/functions added, deleted, or modified.
- *File*: Number of files added, deleted, or modified.
- *Module*: Number of directories that contain a change in the commit.

These measures are used to determine the overall impact a given commit has to a system. That is, a given commit (change) can be very localized to one file or function. Alternatively, it can be system wide impacting a large number of files. The goal is to characterize what a typical adaptive commit looks like. Or, more importantly, the goal is to see if there is even a typical adaptive commit. The file

and module size measures are inexpensive to calculate, as only the log entries need to be analyzed. For each commit, the three size measures are computed. We use the values of these measures as the data points for classification. To aid such classification, a descriptive-statistics method was used to classify the commits into different categories based on the number of files, methods, and modules being changed. We use a 5-point summary approach, the same approach that was used for commit categorization in [1]. This categorization can then be displayed using a boxplot. We developed a tool to apply this categorization over the collected commits, both adaptive and non-adaptive, from the version history of the examined systems in the study. Here, we examined the following questions.

- What is the distribution of adaptive-change commits?
- Do adaptive-change commits cause more files to change than other commit types?
- Can we classify the adaptive-change tasks as a system-wide maintenance operation?

TABLE III provides a summary (the data, quartiles) of the commit categorization using the file-size measure for KOffice. The *Range* column of this table represents the boundaries of each defined region for the corresponding boxplot. The results show that the majority of the non-adaptive commits fit into the small and extra small categories of the examined systems. However, larger commits (large and extra-large categories) occur with a higher percentage for adaptive commits. For instance, about 21% of adaptive commits, compared with 9% of non-adaptive commits, are classified as larger commits for KOffice. Given this trend, the adaptive-maintenance tasks on average touch more files than non-adaptive tasks. That is, developers performed an adaptive task by adding/deleting/modifying code statements across the entire system, and then committed these changes using one large commit. Therefore, the file-size measurement provides an explanation as to why the number of adaptive commits is much smaller than the number of other non-adaptive maintenance commits.

In an attempt to get a more detailed picture, we further processed the files to the granularity of methods/functions. We use the GNU Unix *diff* utility to identify modified, added, and deleted methods/functions occurring in each commit of the examined systems. Our observation, for the studied systems, is that nearly 27% of the adaptive commits are in the large or extra-large categories, while only approximately 7% of the non-adaptive commits are large or extra-large.

The outcomes of commit categorization histograms using module size measurements show that the module-based results reflect the fact that the adaptive changes were often system wide. That is, changing the GUI framework impacted the entire system in a similar manner.

Moreover, we computed the level of significance of these measures using the *Mann-Whitney U* non-parametric tests. In this statistical testing, a *p-value* of 0.05 represents the significance level. If the *p-value* is greater than 0.05, then

we assume there is no observable difference between adaptive and non-adaptive. The results of this test for the file measure are 0.0347 for KOffice, 0.0393 for Extragear, and 0.0414 for OSG (i.e., significant). Additionally, the p-values are also significant for module and method measures.

TABLE III. FIVE-POINT SUMMARY OF COMMITS CATEGORIZED BY FILE-SIZE FOR KOFFICE.

KOffice			
Quartile	Number of Files		
Q0 (Min)	1		
Q1	1		
Q2 (Median)	2		
Q3	4		
Q4 (Max)	3806		
IQR	Q3-Q1= 3		
Boxplot	Range (#files)	Ratio in Adaptive commits	Ratio in Non-Adaptive commits
Extra-Small	1-1	24.11 %	57.33 %
Small	2-4	32.59 %	23.30 %
Medium	5-8	21.69 %	10.57 %
Large	9-12	9.40 %	4.92 %
Extra-Large	>=13	12.21 %	3.88 %

#### IV. THREATS TO VALIDITY

Several threats to validity may influence the results of our work and the capability to generalize the obtained results. In other words, the initial research and findings seem promising, but a number of important questions still need to be addressed. The presented case study is based on investigating the version history. The packages that we examined are prime examples of successful open-source development that involve a number of API modifications. Moreover, the test systems for the preliminary experiments were chosen because of their availability and the fact that other researchers ran experiments using these systems. However, we do not claim that the obtained results would generalize to a broad range of systems, such as closed-source systems.

#### V. RELATED WORK

A number of studies based on software maintenance categorization have been suggested. In terms of the frequency of maintenance commits, Lientz et al. [3] published a survey of software maintenance, where they found that 18.2% of the accomplished maintenance was categorized as adaptive.

Several researchers have proposed studying and extracting information from the commits of software version histories. Robles et al. [5] investigated version histories to study the evolution of a software distribution in terms of the number of packages, lines of code, use of programming languages, and sizes of packages/files. The characterization of a typical version history commit is investigated with respect to the number of files, number of lines, and number of hunks committed together in [1]. Mockus and Votta [4] found several results on using version-control data in

detecting the relationships between the type and size of the software changes. One of their results is using the difficulty, size, and time intervals to identify the type of change. Hindle et al. [2] proposed an automated classifier for large commits by training machine learners on features extracted from the commit metadata. Their results demonstrate that the change size provides helpful information to allow large commits to be classified automatically.

To the best of our knowledge, this is the first work that not only uncovers but also further analyzes adaptive change commits, with comparison to non-adaptive maintenance tasks. There is no previous work in the literature that identifies the main characteristics of adaptive commits in large-scale systems, including the commit size.

#### VI. CONCLUSION AND FUTURE WORK

The results of a case study to identify and analyze the adaptive changes occurring due to a migration to a new API were presented. The study is based on data obtained by manually examining the version history to distinguish adaptive from non-adaptive commits of the KOffice, Extragear/graphics, and OSG open source projects. Both KOffice and Extragear/graphics underwent an adaptive maintenance task that involved migration from Qt3 to Qt4. OSG underwent a migration of OpenGL to a new version.

The study uncovered the main result that the commits involved in an adaptive-maintenance task are typically system wide and large. That is, on average adaptive-maintenance tasks touch more files than non-adaptive tasks. For instance, with respect to a file-size measure the obtained result shows that 21% of adaptive commits, compared with 9% of non-adaptive commits, are classified as large commits for KOffice.

In the future, we plan to repeat the study with additional causes of adaptive changes. We also plan to develop a general automated adaptive identification approach via Information Retrieval (IR) methods, such as vector space models (VSM) or latent semantic indexing (LSI). Moreover, we will expand our study by investigating more characteristics regarding adaptive maintenance.

#### II. REFERENCES

- [1] Alali, A., Kagdi, H., and Maletic, J. I., "What's a Typical Commit? A Characterization of Open Source Software Repositories", ICPC, , 2008, pp. 182-191.
- [2] Hindle, A., German, D. M., Godfrey, M. W., and Holt, R. C., "Automatic classification of large changes into maintenance categories", ICPC, 2009, pp. 30-39.
- [3] Lientz, B. P., Swanson, E. B., and Tompkins, G. E., "Characteristics of application software maintenance", C ACM, vol. 21, 1978, pp. 466-471.
- [4] Mockus, A. and Votta, L. G., "Identifying Reasons for Software Changes Using Historic Databases", ICSM, 2000, pp. 120 -130.
- [5] Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M., and Amor, J. J., "Mining large software compilations over time: another perspective of software evolution", MSR, 2006, pp. 3-9.

# Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness?

Tosin Daniel Oyetoyan<sup>1</sup>, Reidar Conradi<sup>1</sup>

<sup>1</sup>Department of Computer and Information Science  
Norwegian University of Science and Technology  
Trondheim, Norway  
{tosindo, conradi, dcruzes}@idi.ntnu.no

Daniela Soares Cruzes<sup>1,2</sup>

<sup>2</sup>SINTEF  
Trondheim, Norway  
danielac@sintef.no

**Abstract**— Previous studies have shown that dependency cycles contain significant number of defects, defect-prone components and account for the most critical defects. Thereby, demonstrating the impacts of cycles on software reliability. This preliminary study investigates the variables in a cyclic dependency graph that relate most with the number of defect-prone components in such graphs so as to motivate and guide decisions for possible system refactoring. By using network analysis and statistical methods on cyclic graphs of Eclipse and Apache-ActiveMQ, we have examined the relationships between the size and distance measures of cyclic dependency graphs. The size of the cyclic graphs consistently correlates more with the defect-proneness of components in these systems than other measures. Showing that adding new components to and/or creating new dependencies within an existing cyclic dependency structures are stronger in increasing the likelihood of defect-proneness. Our next study will investigate whether there is a cause and effect between refactoring (breaking) cyclic dependencies and defect-proneness of affected components.

**Keywords**— cyclic dependency graphs; defect-proneness; graph complexities; refactoring

## I. INTRODUCTION

Maintaining software systems is a non-trivial task since these systems grow both in size and complexity over time [1]. Thus, it is not surprising that the cost of software maintenance is estimated to be the highest in the overall software budget [2]. Despite high maintenance costs and continuous research efforts to improve software quality, there are still evidence of system and business failures due to software defects [3, 4].

The structural complexity of software systems has been associated with defects [5]. The more complex a system is, the higher the risk of defects and failures. One area of such complexity is cyclic dependencies among software components, yet evidence confirms that they are wide spread in software systems [6]. Recently, we have demonstrated that components in dependency cycles account for most number and severity of defects [7, 8] both at the class file and package levels. Our findings show that, about 65% of defect-prone<sup>1</sup> class files are in cyclic graphs. Where cyclic class files are approximately 44%<sup>2</sup> of the total number of class files.

<sup>1</sup> A defect-prone component as used in this study is defined as components with one or several defects

<sup>2</sup> This figure is an average of cyclic data from six different applications

Furthermore, the cyclic components account for an additional 11-17% of defect-prone components from the “depend-on-cycle<sup>3</sup>” group. In total, an approximate 80% of defect-prone class files are cyclic-related. Similarly, an approximate 90% of defect-prone packages are cyclic-related with cyclic packages accounting for about 58% of the total number of packages. We do not consider these figures to be trivial and based on the significance of the results; we are motivated to believe that further understanding of cyclic dependent components will be useful to guide decisions and provide reasoning for refactoring activities on cycles.

In relation to refactoring and software defects, Weissgerber and Diehl [9] found no correlation in particular between refactoring and defects opened in the subsequent days. Their results showed that there are periods where high refactoring was followed by increase in the number of defects as well as phases where refactoring led to no defects, although, the latter type were more prevalent. Ratzinger et al. [10] demonstrated that the number of software defects decreases when the number of refactoring increases in the preceding time period. Bavota et al. [11] showed that some kinds of refactoring are unlikely to be harmful but certain kinds such as refactoring involving hierarchies (e.g. pull up method) are likely to induce defects. These studies have not considered refactoring cyclic dependent components in relation to defect-proneness. Thus they differ from our work.

There have been previous studies on network analysis of dependency graphs in relation to defect proneness of components [12-14]. Zimmermann and Nagappan [15] analyzed three different types of dependency sub-graphs (INTRA, OUT and DEP) at three separate levels of granularity. Using graph complexity measures, they obtained correlation between these measures and defects in each of the three sub-graphs and built regression models across the sub-graphs. One significant difference between our work and theirs is in the type of sub-graphs. We have focused on cyclic dependency graphs, which are missed in their study. Furthermore, our focus is to obtain variables to guide decision making during refactoring activities.

This paper extends our previous studies [7, 8] by investigating the correlation between the number of defect-prone components and cyclic graphs complexities.

<sup>3</sup> We define a “depend-on-cycle” component as component that is not in cycle but depends on component that is in cycle

Consequently, we want to use these findings to motivate the refactoring (breaking) of defect-prone cyclic dependent components. This paper reports the preliminary results of this investigation and presents the direction for future work.

We are aware that correlation does not necessarily imply causality [16] because of the possibilities of hidden variables that may explain this higher number and severity of defects in the cyclic groups of these systems. Hence, our approach is to perform an experiment based on these preliminary findings in an industrial setup whereby we control for as many factors that could explain these effects and thus allow us to draw a reasonable conclusion on the effect of refactoring cyclic dependent components in relation to defect-proneness. We provide the details in Section IVa.

The rest of the work is structured as follows; in Section II, we lay the background to this study. Section III details our preliminary study. We report the results in Section IV and discuss the case study for our next study. Lastly, we conclude this study in Section V.

## II. BACKGROUND

In a software system, a component  $X$  is said to have dependency on another component  $Y$  if  $X$  requires  $Y$  to compile or function correctly [17]. Formally, a dependency graph of an object-oriented (OO) program, is defined as follows [18]:

**Definition 1.** An edge labeled digraph  $G = (V, L, E)$  is a directed graph, where  $V = \{V_1, \dots, V_n\}$  is a finite set of nodes,  $L = \{L_1, \dots, L_k\}$  is a finite set of labels, and  $E \subseteq V \times V \times L$  is the set of labeled edges.

**Definition 2.** The object relation diagram (ORD) for an OO program  $P$  is an edge-labeled directed graph (digraph)  $ORD = (V, L, E)$ , where  $V$  is the set of nodes representing the object classes in  $P$ ,  $L = \{I, Ag, As\}$  is the set of edge labels representing the relationships (Inheritance, Aggregation, Association) between the classes and  $E = E_I \cup E_{Ag} \cup E_{AS}$  is the set of edges.

Furthermore, we define the various measures of a graph that are important for our study [19].

1. **Geodesic** is the shortest path between two nodes in a graph
2. The **eccentricity** of a node is its longest geodesic.
3. The **diameter** of a graph is the maximum eccentricity of the nodes.
4. The **radius** of a graph is the minimum eccentricity of the nodes.
5. The **density** of a graph is the ratio of the number of edges in the graph to the maximum possible edges in the graph.

### A. Cyclic dependencies and the hypothesis of defect propagation

In graph theory [20], a cyclic dependency graph also known as strongly connected components (SCC) in a directed graph  $G = (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for every pair of vertices  $u$  and  $v$  in  $C$ , both are reachable from each other. An example is depicted in Figure 1a of a hypothetical cyclic graph. In this graph, all the six components

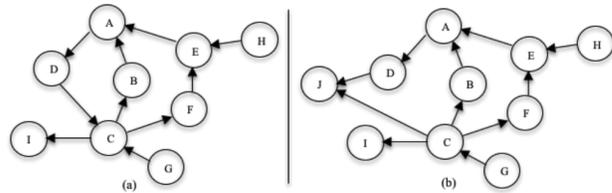


Fig. 1. Cyclic dependencies and defect propagation effect in a software network [8]

in two cycles (A, D, C, B, F, E) are mutually reachable from one another.

A related concept is the notion of dependency on components that are in cyclic relationships. We termed this as “depend-on-cycle” components [8]. Such “depend-on-cycle” components (e.g. G and H in Fig. 1a) obviously share the same complexity as the “in-Cycle” components that they depend on since they can reach all other components that are in these cyclic paths.

Cyclic dependency increases coupling complexities and thus has the potential to propagate defects in a software network [21]. Consider the hypothetical example in Fig. 1a, a defect in component I has the potential to propagate to components C, B, A, D, G, F, E and H. Let us say these cyclic components are refactored such that a new component J is introduced as depicted in Fig. 1b. The possible propagation of defects from I is significantly reduced to only C and G.

## III. PRELIMINARY STUDY

In this initial study, our goal is to find variables in cyclic dependency graphs that correlate with defect and thus motivate for refactoring possibilities. The research questions we want to investigate in this study and subsequent study are:

- RQ1.** What variables within a cyclic dependency graph correlate most with the number of defects and defect-prone components?
- RQ2.** Can the refactoring of factors in RQ1 correspond to a decrease in the number of defects and defect-prone components?

### A. Data Computation

**Dependency and defect data:** We have used the dependency data of two (Eclipse and ActiveMQ) of the systems collected in one of our previous papers [8] for this analysis. We subsequently compute the SCCs from this dependency data. Previous studies (e.g. [6]) already show that most systems have very long cycles such that hundreds of components are tangled in one large dependency cycle. In our analysis, the largest SCC in release 5.7.0 of ActiveMQ contains 414 class files while the largest SCC in Eclipse r3.0 has 690 classes. Our approach here is to break the long cycles into several smaller cyclic sub-graphs in such a way that all cyclic components are covered (Example of such sub-graphs is shown in Fig. 2). Using this approach allows us to verify whether an increase or decrease in size-based and distance-based cyclic graph measures correlate with defects. Similarly, we used the popular Eclipse defect data reported in [22] and

TABLE I. NETWORK METRICS USED IN THE STUDY

Metric	Formula
<b>Size-based</b>	
#Nodes	$ V $
#Edges	$ E $
<b>Distance-based</b>	
Diameter	$Max(Eccentricities)$
Radius	$Min(Eccentricities)$
<b>Other complexity</b>	
Density	$\frac{ E }{ V  \cdot  V }$

the defect data collected for ActiveMQ in our previous study [8].

**Network Measures:** As summarized in Table I, we have computed size and distance based measures. For size measures, we aggregate the number of nodes and number of edges in each graph. To compute the distance measures, we used the Floyd-Warshall algorithm [20] to calculate the geodesics i.e. “all-pairs shortest distance” between the nodes of each of the generated sub-graphs. We then compute the eccentricity (i.e. the maximum of the shortest distance) for each node. Subsequently, we calculate the diameter and the radius for each graph and sub-graphs from the values of the nodes’ eccentricities.

IV. PRELIMINARY RESULTS AND FUTURE STUDY

Concerning RQ1, the results (Table II) show that the size-based measures (number of nodes and edges) correlate strongly and better than distance based measures with the number of defects and defect-prone components in these systems and at both levels (class and packages) of granularity. In other words, the higher the number of components and dependency relationships in cycle, the higher the probability of defects and defect-proneness of components. Furthermore, as displayed in Fig. 3, the R-squared value shows a good linear fit that suggests that we can predict the number of defect-prone components in a cyclic dependency graph using the graph size measures.

We can further interpret these results to mean that adding a new class/package (node) or dependency relationships (edge)

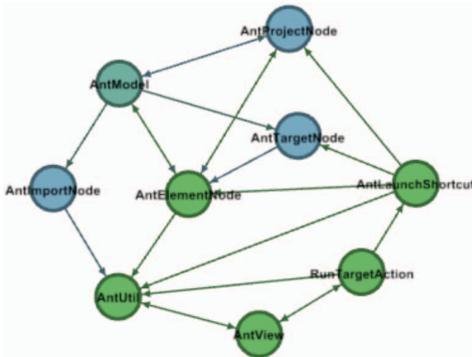


Fig. 2. A cyclic dependency graph with many inner cycles from Eclipse 3.0 (Generated with Gephi: <http://gephi.org/>)

TABLE II. CORRELATION BETWEEN NETWORK METRICS AND NUMBER OF DEFECT-PRONE COMPONENTS IN CYCLIC GRAPHS

Metric	Eclipse				ActiveMQ			
	Class		Package		Class		Package	
	S	P	S	P	S	P	S	P
Nodes	0.79	0.89	0.89	0.98	0.74	0.71	0.81	0.91
Edges	0.80	0.85	0.89	0.97	0.75	0.73	0.80	0.92
Diameter	0.41	0.19*	0.72	0.64	0.50	0.52	0.13*	0.26*
Radius	0.38	0.13*	0.61	0.61	0.38	0.45	0.51	0.59
Density	-0.52	-0.52	-0.78	-0.78	-0.38	-0.63	-0.77	-0.77

S - Spearman; P - Pearson (All non-asterisked results are significant at  $\alpha = 0.01$ )

to an existing cycle strongly increases the number of defect-prone components and defects in the cycle. In addition, size measures (i.e. node/edge) increase defect-proneness more than the strength of connection (diameter). These results agree with previous studies [12-14] that graph complexities correlate with defect-proneness of components and in comparison to [15] at the same granularity level, the correlation is higher and stronger for cyclic graphs as against other types of graph.

Until now, we have not seen a systematic measurement of the impact of refactoring cyclic dependent components in relationship to defect-proneness. This has therefore motivated us to verify this in our next study. Our speculation is that if this hypothesis is not rejected such knowledge can strengthen the refactoring of highly defect-prone cyclic related components at both class and package hierarchies for existing systems and dissuade developers from writing cyclically connected programs.

A. Proposed Experiment

Regarding RQ2, we want to investigate whether breaking (refactoring) defect-prone cyclic dependent components would have effect on such components. There are several tools<sup>4</sup> and approaches that we can take advantage of to dissuade developers from writing cyclic codes or to refactor existing cyclic codes. Since a high correlation does not necessarily imply causality and there could be other hidden variables [16] (e.g. other design measures and complexities) that account for the defect-proneness of those components in the cyclic graphs, we have taken a practical approach to verify this conjecture.

The case study for this experiment is an industrial Smart Grid system. The application is a distribution management system that provides real-time operational support by

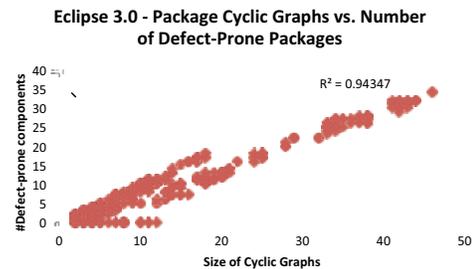


Fig. 3. Scatter plot between size of cyclic graph vs. #DPCs in Eclipse 3.0

<sup>4</sup> JDepend, NDepend, Dependometer, Dependency Structural Matrix

continuously receiving status data from the power grid. The system has been in development for about six years and we have analyzed six post releases data of this application. It is mostly developed with C# programming language. Furthermore, it has a size of about 380KLOC and contains 1459 class files and 2484 classes as of release 4.2.4 (Release date: Nov. 14, 2012). Our analysis of this application gave similar results with those presented in this paper.

Since the software undergo frequent releases by the company, we consider a six to nine months post refactoring data (after release) to be appropriate for our analysis. In order to eliminate unwanted factors that could possibly explain the difference between refactored cyclic groups and the rest of the groups, we decide to take measurements of well known object-oriented (OO) metrics and complexities in addition to the defect measures for three groups: (1) Refactored cyclic groups (2) Non-refactored cyclic groups and (3) Non-cyclic groups. We are particularly interested in post release defect data and therefore, our approach for the case study is as follows:

1. Select  $N$  sample of most defect-prone cyclic dependent graphs (class and package)
2. For each graph, record all measurement for the various metrics (number of defects, type of defects, severity of defects, correction efforts, lines of code, defect densities, OO metrics and complexities) for all components in these cyclic graphs (i.e. six/nine months pre-refactoring data)
3. Similarly, take measurements for the remaining cyclic groups and non-cyclic groups
4. Perform refactoring
5. From six/nine months after the next release, take new measurements detailed in steps 2 and 3 (post-refactoring)
6. Compare the new measures with the previous measures for each of the components in the groups

We admit that it is difficult to control and randomize all relevant factors in this experiment. For instance, the system may evolve and increase further in complexity as a result of dynamic changes in business requirements within the experiment phase. However, by going through these steps, whereby the measurements for the fixed sets of pre-refactoring data of components are compared against their post-refactoring data, we can, at least, move a step further to understanding the effect of refactoring defect-prone cyclic components and verify whether such activity can reduce the defect-proneness of affected components.

## V. CONCLUSIONS

We have considered dependency cycles as an important area of dependency graph with very high complexities. We show that cyclic graphs complexities certainly have very strong correlation to defect-proneness of components. An increase in the size and strength of connections in a software cyclic dependency graphs correspond to an increase in the number of defect-prone components. Our next study is therefore focused on verifying the hypothesis that refactoring highly defect-prone cyclic dependency graphs will reduce the defect-proneness of the affected components in these graph structures.

## REFERENCES

- [1] Lehman, M.M., *Programs, Life-Cycles, and Laws of Software Evolution*. Proceedings of the Special Issue Software Engineering, 1980. **68**(9): p. 1060-1076.
- [2] Erlikh, L., *Leveraging Legacy System Dollars for E-Business*. IT Professional, 2000. **2**(3): p. 17-23.
- [3] Leo, K. *Why banks are likely to face more software glitches in 2013*. [Web] 2013 April 24, 2013; Available from: <http://www.bbc.co.uk/news/technology-21280943>.
- [4] Lilley, S., *Critical Software: Good Design Built Right*. NASA System Failure Case Studies, 2012. **6**(2).
- [5] Basili, V.R., L.C. Briand, and W.L. Melo, *A validation of object-oriented design metrics as quality indicators*. IEEE Transactions on Software Engineering, 1996. **22**(10): p. 751-761.
- [6] Melton, H. and E. Tempero, *An empirical study of cycles among classes in Java*. Empirical Software Engineering, 2007. **12**(4): p. 389-415.
- [7] Oyetoyan, T.D., R. Conradi, and D.S. Cruzes. *Criticality of Defects in Cyclic Dependent Components*. in *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 22-23 September 2013 (Accepted)*.
- [8] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *A Study of Cyclic Dependencies on Defect Profile of Software Components*. Journal of Systems and Software, 2013. (in press).
- [9] Weissgerber, P. and S. Diehl, *Are refactorings less error-prone than other changes?*, in *Proceedings of the 2006 International Workshop on Mining Software Repositories 2006*, ACM: Shanghai, China. p. 112-118.
- [10] Ratzinger, J., T. Sigmund, and H.C. Gall, *On the relation of refactorings and software defect prediction*, in *Proceedings of the 2008 International Working Conference on Mining Software Repositories 2008*, ACM: Leipzig, Germany. p. 35-38.
- [11] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O., *When Does a Refactoring Induce Bugs? An Empirical Study*. in *IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2012*.
- [12] Bhattacharya, P., Iliofotou, M., Neamtii, I., & Faloutsos, M., *Graph-Based Analysis and Prediction for Software Evolution*. 2012 34th International Conference on Software Engineering (ICSE), 2012: p. 419-429.
- [13] Tosun, A., B. Turhan, and A. Bener, *Validation of network measures as indicators of defective modules in software systems*, in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering 2009*, ACM: Vancouver, British Columbia, Canada. p. 1-9.
- [14] Zimmermann, T. and N. Nagappan, *Predicting Defects using Network Analysis on Dependency Graphs*. 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 2008: p. 530-539.
- [15] Zimmermann, T. and N. Nagappan, *Predicting subsystem failures using dependency graph complexities*. ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 2007: p. 227-236.
- [16] Pease, C.M. and J.J. Bull, *Scientific Decision-Making*, 2000, Retrieved from <http://www.utexas.edu/courses/bio301d/index.html>.
- [17] Jungmayr, S. *Identifying test-critical dependencies*. in *Software Maintenance*. 2002.
- [18] Kung, D., Gao, J, Hsia, P, Toyoshima, Y, & Chen, C., *On Regression Testing of Object-Oriented Programs*. Journal of Systems Software, 1996. **32**(1): p. 21-40.
- [19] Wasserman, S. and K. Faust, *Social network analysis : methods and applications*. Structural analysis in the social sciences. 1994, Cambridge ; New York: Cambridge University Press. xxxi, 825 p.
- [20] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C., *Introduction to algorithms*. 2nd ed. 2001, Cambridge, Mass.: MIT Press. xxi, 1180.
- [21] Briand, L.C., J.W. Daly, and J.K. Wust, *A unified framework for coupling measurement in object-oriented systems*. IEEE Transactions on Software Engineering, 1999. **25**(1): p. 91-121.
- [22] Zimmermann, T., R. Premraj, and A. Zeller. *Predicting Defects for Eclipse*. in *International Workshop on Predictor Models in Software Engineering*. 2007.

# Towards a Taxonomy of Programming-related Difficulties during Maintenance

Aiko Yamashita  
 Mesan AS & Simula Research Laboratory  
 Oslo, Norway  
 Email: aiko@simula.no

Leon Moonen  
 Simula Research Laboratory  
 Oslo, Norway  
 Email: leon.moonen@computer.org

**Abstract**—Empirical studies that investigate the relationship between source code characteristics and maintenance outcomes rarely use causal models to *explain* the relations between the code characteristics and the outcomes. We conjecture that the lack of a comprehensive catalogue of programming-related difficulties and their effects on different maintenance outcomes is one of the reasons behind this. This paper takes the first step in addressing this situation based on empirical evidence collected in a longitudinal maintenance study on four systems. Professional developers were hired to implement a number of changes in each of the systems. These activities were observed in detail over a period of 7 weeks, during which we recorded on a daily basis *what specific problems* they faced. The collected data was transcribed and analyzed using open and axial coding. Based on an analysis of these results, we propose a preliminary taxonomy to describe the programming-related difficulties that developers face during maintenance. Our intention is not to replace the existing categorizations/taxonomies, but to take the first steps towards an integrated, comprehensive catalogue by aligning our empirical observations and the earlier literature.

**Keywords**—maintainability, maintenance difficulties, maintenance problems, program comprehension, empirical study.

## I. INTRODUCTION

The assessment of how well a software system can be maintained is one of the greater and long-lasting challenges of software engineering. There is a substantial body of work that investigates if certain source code characteristics affect given maintenance outcomes (such as effort needed or defect proneness). When assessing software maintainability, it is not only important to determine *if* different code characteristics have an effect on maintenance outcomes, but we also need causal models to better understand *how* those characteristics affect the outcomes. These models are needed, for example, to uncover and reason about the interaction effects between different factors that may play a role.

One approach to get a step further in the development of causal models in software maintenance research is closely studying the difficulties that developers actually experience while solving change requests during maintenance. The study of maintenance difficulties is relevant for developing such causal models because these experiences reflect and potentially explain different outcomes of a maintenance project, such as performance, product quality or developers' motivation levels.

To this date, the software engineering literature does not provide an *integrated* compendium of programming-related difficulties occurring during programming specific activities

during maintenance. A good starting point would be to look at *incremental change* work-flow described by Rajlich & Gosavi [1]. An integrated, well-defined taxonomy of difficulties during *each* of the steps in an working flow such as the *incremental change* flow can potentially support better understanding of the factors that affect maintainability.

By lack of *integration* we mean for instance, work on *program comprehension* only covers the initial steps (e.g., *concept extraction*, *concept location* [1]) of a maintenance task, leaving out code modification activities. Maintenance *challenges* have been depicted and categorized at project management level, and maintenance *activities* have been classified at the goal level, but these taxonomies are too coarse-grained to really address the “programming” aspects of maintenance and evolution at the level we see in [1].

In this paper, we present a proposal for a taxonomy on programming-related maintenance difficulties that is based on detailed observations of a maintenance project involving four Java web applications and six software professionals over a period of 7 weeks in total.

The remainder of this paper is structured as follows: Section 2 presents the related work and the knowledge gap. The approach for developing the taxonomy is discussed in Section 3. Section 4 presents the taxonomy and Section 5 argues for its relevance based on existing theories. We conclude and present plans for future work in Section 6.

## II. RELATED WORK AND KNOWLEDGE GAP

Several studies have addressed and described different maintenance-related problems or difficulties, as well as different factors causing difficulties during maintenance.

Lientz and Swanson [2], and Dekleva [3] describe maintenance problems from a managerial perspective. For instance, they reported: *user knowledge*, *programmer effectiveness*, *product quality*, *programmer time availability*, *machine requirements*, and *system reliability* as the sources for major problems during maintenance. Similarly, Dekleva describes maintenance problems elicited from experienced software developers, and identified four major categories: *maintenance management*, *organizational environment*, *personnel factors*, and *system characteristics*. Palvia et al. [4] elaborate further on the initial set of problems/issues in software maintenance reported by Lientz and Swanson. Chapin et al. [5] provide a classification for types of software maintenance and evolution, and describe how their characteristics can

impact different aspects of the product and the business. Hall et al. [6] and Chen et al. [7] describe maintenance issues and potential factors behind them from a Software Process Improvement perspective, and Reedy et al. [8] propose a catalogue of maintenance issues within the context of software configuration management.

Karahasanovic et al., [9] reported a catalogue of program comprehension difficulties resulting from an experiment designed to investigate whether following a *systematic* strategy for program comprehension is realistic or not in large programs. They also compared which difficulties were associated to which type of comprehension strategy (i.e., between “as-needed” and “systematic” strategies).

Webster et al. [10] propose a taxonomy of risks in software maintenance, which describes potential factors negatively affecting software maintenance. Examples include: antiquated system and technology, program code complexity, difficulties in understanding programming “tricks”, large number of lines of code affected by change request, and large number of principal software functions affected by the change.

From the maintenance problems illustrated within the identified literature, focus has been laid on -process and organizational factors, following more a managerial perspective. Technical or programming perspectives for maintenance issues seem to have a lower profile, with the work by Webster et al. and Karahasanovic et al. being the closest or most relevant to the focus of our work.

Most of the above-mentioned categorizations are not optimal for studying the relation between code characteristics and maintenance, since they: (1) do not cover the whole span of programming activities during maintenance, or (2) lack of enough level of detail (concreteness) to be of any use at the programming level. For instance, Karahasanovic et al. only focuses on comprehension-related difficulties, while comprehension only accounts for one portion of the maintenance activities, leaving out other tasks such *problem solving*, *impact analysis* and *debugging*. The work by Lientz and Swanson, Dekleva, Palvia, and Chen all focuses on managerial or process improvement perspectives. They cover the whole spectrum of the software maintenance process, and mention difficulties related to code quality very superficially.

### III. METHODOLOGY

In this section, we first describe the context of the maintenance project from which the observations were derived. Subsequently, we provide a description of the qualitative techniques used in order to develop our taxonomy.

**Maintenance Context:** In 2008, Simula’s Software Engineering department commissioned a maintenance project that involved adapting four existing Java applications to a new Content Management System (CMS). The project was conducted between September and December by six professional developers from two companies. The tasks covered the main maintenance classes (corrective, adaptive, perfective)

Table I  
EXAMPLE CODES FOR THE THREE PERSPECTIVES

<i>Activity</i>	<i>Difficulty</i>	<i>Factor</i>
Add source code	Confusion	Code Size
Modify source code	Error propagation	Bad naming
Look up information	Manual search	Logic entanglement
Configure library	Inconsistent change	Duplication
Debug	Scattered change	Rule violation

and included adapting the systems to retrieve information from the new CMS, replacing the existing authentication mechanisms by web services and adding new reporting functionality. The project was designed as an ethnographic study in which the developers’ activities were observed in detail (with consent). None of the maintainers, nor their companies, were involved in the original development, so they had no prior knowledge about of the systems. For more details on the study context we refer to [11].

**Data Collection:** Three main data sources were used in order to build the taxonomy. First, the researcher present at the study kept an *observation logbook* where the most important aspects of the study were annotated in a daily basis. Second, interviews or progress meetings (20-30 minutes) were conducted daily for all developers to keep track of the progress, and register difficulties encountered during the project (e.g., dev4: “*It took me 3 hours to understand this method because of strangely named variables*”). Third, recorded think-aloud sessions (ca. 30 minutes) were conducted every second day to observe the developers in their daily activities. Subsequently, the recorded material from the interview and the think-aloud sessions were transcribed, annotated and summarized.

**Data Analysis:** *Grounded theory* was used to analyze the collected data. To this end, the transcripts from the interviews were coded using *open* and *axial* coding techniques [12]. *Open coding* is a form of content analysis in which the statements from the developers were annotated using more abstract labels (codes) that were (initially) derived from the *observation logbook*. For generating the coding schema, three perspectives were used: *Activity*, *Difficulty* and *Factor*. *Activity* refers to the maintenance context in which the difficulty manifests. *Difficulty* refers to the concrete hindrance to the activity being performed at the time, and *Factor* is the potential cause of that hindrance. For example, the following statement: “*...while searching of the place to make the change, I spent many hours because I had to examine many classes...*” would be coded as *Activity*: concept location [1], *Difficulty*: time consuming/manual search, and *Factor*: complexity. The coding schema was revised in an iterative fashion, during the annotation process. An example of various codes is shown in Table I. Subsequently, a technique called *axial coding* was used to group the annotated statements according to the most similar concepts, using all three perspectives described previously, thereby making connections between the initial categories and creating higher level abstractions.

#### IV. THE TAXONOMY

Based on the analysis of the data, and alignment of the terminology and concepts with the existing literature (discussed in Section II), we identified the following five types of maintenance difficulties:

**(1) Introduction of faults as result of changes:** comprises the introduction of faults to the system as result of changes performed in the code. Faults normally manifest in the form of *failures* identified by the developers during maintenance or detected during acceptance testing.

**(2) Confusion and erroneous hypothesis generation during program comprehension:** relates to developers generating an initial hypothesis on the behavior of the system or one of its components, and subsequently finding contradictory or inconsistent evidence. This category also included situations where the developer generates an erroneous hypothesis from “confounding” evidence in the code (which in some situations lead to the introduction of defects).

**(3) Slow acquisition of overview/general understanding of the system:** relates to developers spending considerable time on getting an overview of the system before they feel enough confident on the strategies they choose to solve the tasks. Also, difficulties in understanding the mechanics of subsystems fall into this category.

**(4) Time-consuming information quests:** relates to developers taking considerably long time before they could find the relevant information for their tasks.

**(5) Wide-spread (time-consuming) changes:** relates to situations where developers were forced to perform changes in extensive areas of the system in order to complete a given task. Often these changes required to manually inspect the areas in the code prior to any modifications.

A detailed description of this categorization, overall statistics and concrete examples of the difficulties identified can be found in the dissertation by Yamashita [13, pt. 2, ch. 4].

When observing the difficulties, is natural to see that some of them are related or one may constitute the cause for the other. For example, one can argue that (4) and (5) are the same type of difficulty. However, if we consider the *incremental change* steps described by Rajlich & Gosavi [1], the problems (2), (3), and (4) all relate to concept *extraction* and *location*. The difficulty (5) occurs later, when the developer performs the changes (i.e., *actualization*, *incorporation* and *propagation*), where for example the solution has to be reworked or the solution involves extensive “manual work”.

Some of these difficulties relate to the work by Sillito et al., [14] who describes *challenges* developers face during change requests. For example, their challenge: “Gaining a sufficiently broad understanding” relates to our difficulty (3), while “Making and relying on false assumptions” relates to difficulty (2).

#### V. TAXONOMY RELEVANCE

This section discusses the relevance of this taxonomy using two perspectives: First, we discuss the importance of identifying situations where developers get confused due to misleading cues using the notion of *information beacons* [15]. Second, we discuss the notion of *cognitive overload* based on Wood’s perspective on *task complexity* [16]. We show how our taxonomy helps to capture the interplay between source code properties and programming-related maintenance difficulties.

**Confusion during maintenance:** During the study, we could observe that two of the systems (which also happened to be smaller in LOC than the other two) displayed problems at structural or local level that confused the developers. In the first system, confounding elements were present at a structural level, where data/functionality allocation was not semantically coherent and moreover, not consistent. This resulted on early *false assumptions* of developers with respect to behavior of the code. In latter stages, developers would get confused while finding contradictory evidence on their initial assumptions. In many cases, the false assumptions would hold across the strategy selection and code modification process, leading to the *introduction of defects*. In the second system, variables were not used consistently, probably because of inconsistent *copy-paste* behavior from the programmers who initially wrote the application. In both of these cases, the challenges are closely related to confusing *beacons* or *misleading information cues*. Wiedenbeck defines beacons as: stereotypical segments of code, which serve as typical indicators of the presence of a particular programming structure or operation [15]. If we assume that software maintenance is a goal-oriented, hypothesis-driven, problem-solving process (cf. [17]), any form of confusion that leads to the generation of *false hypotheses* during program comprehension and strategy selection is an important aspect to observe when assessing the maintainability of the source code. Several studies have pointed out the criticality of beacons and information cues for program comprehension. Ko et al. observed how misleading information cues induced failed searches for task-relevant code during maintenance [18]. Wiedeneck reported that beacons which were inappropriately placed in a program led to “false comprehension” of the program’s function [15]. Soloway & Erlich discussed how expert performance drops to near the level of the novice programmers due to rule violations [19].

**Cognitive overload during maintenance:** In our study, the two largest systems were found to be more difficult to understand (or get an overview of their behavior) and to modify than the other two smaller systems [13]. Particularly in the case involving the largest system of all four, the understanding was difficult because information cues were spread across a wide cognitive space, and developers were forced to examine many areas of the code, gathering the

“pieces of the puzzle” before they could achieve enough understanding of the overall system’s behavior. In the same system, developers reported the usage of an extensive, third-party library, as the most time-consuming task. What they referred as “time-consuming” work was not to understand the behavior of the library, but actually to put the elements provided by the library together in order to solve the task. In contrast to the smaller systems, *consistency* was not a problem, but the *complexity*, which resulted in a different type of difficulty. No developers reported being *confused* while trying to understand the large systems, they said it was just hard to get used to the library, and then to use it. These observations are echoed by Wood’s perspective on *task complexity*, in particular to the notion of *component complexity* [16]. *Component complexity* refers to the number of distinct information cues that must be processed simultaneously in the performance of a task. They also relate to Swessler’s notion of *intrinsic cognitive load* [20], which states that high element interactivity in tasks imposes a high cognitive load, where elements interactivity means that they are related in a manner that requires them to be processed simultaneously. In our study, we could clearly identify situations in which *component complexity* increased the effort of *understanding*, and of *using* large libraries, because they required developers to gather distinct, wide-spread information cues.

## VI. CONCLUDING REMARKS AND FUTURE WORK

We are aware that many of the concepts and phenomena that we have describe here, have also been reported in other studies and across different research domain areas (e.g., HCI, Program Comprehension, Software Maintenance). Our intention is not to *replace* the existing categorizations/taxonomies, but to take the first steps towards an integrated, comprehensive catalogue of difficulties/problems developers face during programming-related maintenance activities.

In the future, we would like to refine the taxonomy by improving the consistency, conciseness and descriptive richness of each type of difficulty, and by mapping the difficulties to each of the different programming-related maintenance activities. A preliminary idea is to build a catalog of programming-related maintenance activities based on the stages involved in the *Incremental Change* process, described by Rajlich & Gosavi [1]. Another idea to extend our catalog of difficulties is to “reverse-engineer” the literature on factors *affecting* maintainability (e.g., work by Nierstrasz [21]) and extract and describe in detail the *difficulties* caused by the different factors.

## REFERENCES

- [1] V. T. Rajlich and P. Gosavi, “Incremental change in object-oriented programming,” *IEEE Softw.*, vol. 21, no. 4, pp. 62–69, 2004.
- [2] B. P. Lientz and E. B. Swanson, “Problems in application software maintenance,” *Communications of the ACM*, vol. 24, no. 11, pp. 763–769, 1981.
- [3] S. Dekleva, “Delphi study of software maintenance problems,” in *IEEE Int’l Conf. Softw. Maintenance*, 1992, pp. 10–17.
- [4] P. Palvia, A. Patula, and J. Nosek, “Problems and Issues in Application Software Maintenance,” *J. Information Technology Management*, vol. 4, no. 3, pp. 17–28, 1995.
- [5] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, “Types of software evolution and software maintenance,” *J. Softw. Maintenance: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.
- [6] T. Hall, A. Rainer, N. Baddoo, and S. Beecham, “An empirical study of maintenance issues within process improvement programmes in the software industry,” in *IEEE Int’l Conf. Softw. Maintenance*, 2001, pp. 422–430.
- [7] J.-C. Chen and S.-J. Huang, “An empirical analysis of the impact of software development problem factors on software maintainability,” *JSS*, vol. 82, no. 6, pp. 981–992, 2009.
- [8] A. Reedy, D. Stephenson, E. Dudar, and F. Blumberg, “Software configuration management issues in the maintenance of Ada software systems,” in *IEEE Int’l Conf. Softw. Maintenance*, 1989, pp. 234–245.
- [9] A. Karahasanović, A. K. Levine, and R. Thomas, “Comprehension strategies and difficulties in maintaining object-oriented systems,” *JSS*, vol. 80, no. 9, pp. 1541–1559, 2007.
- [10] K. Webster, K. M. de Oliveira, and N. Anquetil, “A risk taxonomy proposal for software maintenance,” in *IEEE Int’l Conf. Softw. Maintenance*, 2005, pp. 453–461.
- [11] A. Yamashita, “Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data,” *Empirical Softw. Eng.*, pp. 1–33, 2013.
- [12] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 1998.
- [13] A. Yamashita, “Assessing the Capability of Code Smells to Support Software Maintainability Assessments,” Doctoral Thesis, University of Oslo, 2012.
- [14] J. Sillito, K. De Volder, B. Fisher, and G. Murphy, “Managing software change tasks: an exploratory study,” in *Int’l Symp. Empirical Softw. Eng.* IEEE, pp. 23–32.
- [15] S. Wiedenbeck, “The initial stage of program comprehension,” *J. Man-Machine Studies*, vol. 35, no. 4, pp. 517–540, 1991.
- [16] R. E. Wood, “Task complexity: Definition of the construct,” *Organizational Behavior and Human Decision Processes*, vol. 37, no. 1, pp. 60–82, 1986.
- [17] J. Koenemann and S. P. Robertson, “Expert problem solving strategies for program comprehension,” in *SIGCHI Conf. Human Factors in Computing Systems*, 1991, pp. 125–130.
- [18] A. Ko, B. Myers, M. Coblenz, and H. Aung, “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks,” *TSE*, vol. 32, no. 12, pp. 971–987, 2006.
- [19] E. Soloway and K. Ehrlich, “Empirical Studies of Programming Knowledge,” *TSE*, vol. SE-10, no. 5, pp. 595–609, 1984.
- [20] J. Sweller, “Cognitive load theory, learning difficulty, and instructional design,” *Learning and Instruction*, vol. 4, no. 4, pp. 295–312, 1994.
- [21] O. Nierstrasz and M. Denker, “Supporting Software Change in the Programming Language,” in *OOPSLA Ws. Revival of Dynamic Languages*, 2004, p. 5.

# A Pilot Experiment to Quantify the Effect of Documentation Accuracy on Maintenance Tasks

Maurizio Leotta<sup>1</sup>, Filippo Ricca<sup>1</sup>, Giuliano Antoniol<sup>2</sup>, Vahid Garousi<sup>3,5</sup>, Junji Zhi<sup>4</sup>, Guenther Ruhe<sup>3,4</sup>

<sup>1</sup> Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy

<sup>2</sup> Département de Génie Informatique et Génie Logiciel (DGIGL), École Polytechnique de Montréal, Québec, Canada

<sup>3</sup> Dept. of Computer and Electrical Engineering, <sup>4</sup> Dept. of Computer Science, University of Calgary, Alberta, Canada

<sup>5</sup> Graduate School of Informatics, Middle East Technical University, Ankara, Turkey

maurizio.leotta@unige.it, filippo.ricca@unige.it, antoniol@ieee.org, vgarousi@ucalgary.ca, zhij@ucalgary.ca, ruhe@ucalgary.ca

**Abstract**—This paper reports the results and some challenges we discovered during the design and execution of a pilot experiment with 21 bachelor students aimed at investigating the effect of documentation accuracy during software maintenance and evolution activities. As documentation we considered: a high level system functionality description and UML documents. Preliminary results indicate a benefit of +15% in terms of efficiency (computed as number of correct tasks per minute) when a more accurate documentation is used. The discovered challenging aspects to carefully consider in future executions of the experiment are as follows: selecting “the right” documentation artefacts, maintenance tasks and documentation versions, verifying that the subjects really used the documentation during the experiment and measuring documentation-code alignment.

**Keywords**—Documentation Accuracy; Maintenance Tasks; Controlled Experiment.

## I. INTRODUCTION

Several experiments have been conducted to investigate the costs and benefits associated with the UML usage during maintenance and evolution tasks [1], [2]. These works provide very clear insights in terms of the kinds of benefits that can be expected from using UML: for complex tasks and past a certain learning curve, it has been observed that the availability of UML documents could result in improvement of the functional correctness of changes as well as their design quality. These experiments [1], [2] were run providing the subjects (both students and professionals) with and without UML documents (use case, class and sequence diagrams), i.e., the treatment group had access to UML documents while the control group did not have any access. Both groups had access to high level system functionality description, manual pages, and source code.

However, these works did not address the problem of how *non-aligned UML documents* impacts on maintenance tasks. In practice, the experiments [1], [2] treat only the more extreme cases: no UML documents vs. (100%) aligned UML documents. The goal of our long term plan, about which this pilot study constitutes a first step, is to fill the gap left by the above papers, investigating the impact of partially *outdated UML documentation* on maintenance tasks. We believe that this kind of experiments is important because the *outdated documentation* problem is often the case in real-world projects in the software industry [3].

In this paper, we report the design and initial results of a pilot experiment<sup>1</sup> conducted with 21 bachelor students that aims to compare in a maintenance task scenario a “less” aligned technical documentation (i.e., design documents) with a “more” aligned one. The contributions of this paper are two-fold: (1) the results of the experiment, and (2) a list of challenges we found in conducting the experiment.

The paper is structured as follows. Section II provides details on the design of the experiment. Section III presents some preliminary results, a list of non-trivial challenges that we encountered during the execution of the experiment, and threats to validity that could affect our results. Section IV concludes the paper with final remarks and future directions.

## II. EXPERIMENT AND PLANNING

We conceived and designed the experiment following the guidelines by Wohlin *et al.* [4]. The *goal* of this study is to analyse the effect of documentation accuracy on the efficiency of software maintenance tasks, for the *purpose* of investigating different levels of alignment (or up-to-dateness) between UML documents and code and their impact on maintenance tasks, from the *point of view* of software managers and developers in the *context* of maintaining a software system. In what follows, design and other aspects of the experiment are briefly discussed. The experimental material is available at <http://softeng.disi.unige.it/2013-DocAccuracy.php>.

### A. Research Question and Hypothesis

The research question for the study is the following:

*Does the level of alignment between UML documents and code have any impact on efficiency of maintainers?*

From the above stated research question, we derived the following null hypothesis:

–  $H_{0a}$  The level of alignment between UML documents and code has no impact on efficiency of maintainers.

We will consider  $H_{0a}$  as a one-tailed hypothesis because we expect a positive effect of a more aligned documentation on the efficiency of software maintainers.

### B. Context (Objects and Subjects)

Two Java desktop applications comparable in size — EasyMarks (a software for handling student marks composed by 17KLoCs and 89 classes) and AMICO (a software for

<sup>1</sup>A pilot study is often used to evaluate the design of the full-scale experiment which then can be adjusted.

condominium management composed by 14KLoCs and 162 classes) — are the objects of the study. The documentation of EasyMarks and AMICO were developed during two editions (2006 and 2008) of the Software Engineering course at the University of Genova [5] and used within the laboratory part of that course to implement the corresponding systems. These two documents underwent a number of modifications (11 for AMICO and 12 for EasyMarks) since their first version until the execution of experiment presented here. The documentation of each system consists of a high level system functionality description and UML documents (class and sequence diagrams). As source code for the experiment, we selected (and refined) the implementation from the best group for each course edition (i.e., 2006, 2008). Some essential comments to the code were added.

The subjects were 21 students from the Software Engineering course, in their last year of the bachelor degree in Computer Science at the University of Genova (Italy). The subjects had a good programming knowledge<sup>2</sup>, especially Java programming, and an average UML knowledge (which was explained during the course).

### C. Treatment and Experimental Design

Two cases can be distinguished: (i) maintenance tasks executed using a “less” aligned documentation and (ii) maintenance tasks executed using a “more” aligned documentation. Thus, only one independent variable occurs in our experiment, which is nominal. It assumes two possible values: **LESS** or **MORE**. During maintenance tasks, both students with LESS and MORE treatments had access to the high level system functionality description, UML documents and source code. The only different thing is the level of alignment of the UML documents (i.e., class and sequence diagrams). In one case, they are more aligned with the code in the other case less. In our experiment, we used the last version of the documentation of each system for the MORE treatment and selected a previous version with a “sufficient distance” for the LESS treatment. To compute the alignment between design documents and code, we applied a preliminary measure (see also Section III-A). For each UML class diagram belonging to the LESS and MORE treatments: (1) we count the total number of classes (*CsM*), (2) we count the classes in the diagram that are also in the system’s source code (*CsM&S*) and for them (3) we compute the Jaccard index<sup>3</sup> to quantify the similarity between model and source code for each class in terms of fields and methods and (4) we calculate its average value over all the *CsM&S* classes (*AJaccard*) and (5) we compute a total alignment index defined as:  $CsM\&S * AJaccard / CsM$ , and finally (6) we compute the distance between LESS and MORE in percentage. The total alignment index takes into account the number of classes that are both in the model and in the source code, their similarity computed with the Jaccard index and the total number of classes. The total alignment is equal to 1 if and only if all the classes in the model have a match in the code and the averaged Jaccard index is equal to 1<sup>4</sup>. For the

<sup>2</sup>We estimated it by means of a pre-experiment questionnaire.

<sup>3</sup>[http://en.wikipedia.org/wiki/Jaccard\\_index](http://en.wikipedia.org/wiki/Jaccard_index)

<sup>4</sup>Note that, to compute the total alignment, we did not consider the classes that are in the source code but not in the model, since, the UML documents have been used to locate the portions of source code to modify. Thus, we calculated the alignment of the model with respect to the source code and not vice versa.

TABLE I. UML CLASS DIAGRAMS AND SOURCE CODE ALIGNMENT

System	UML Model Version		CsM	CsM&S			Total Alignment	Distance
				#	%	Ajaccard		
EasyMarks	1.3I	LESS	30	11	36,7	0,8707	0,3192	27,90%
	2.0	MORE	43	28	65,1	0,6271	0,4083	
AMICO	2.0	LESS	85	52	61,2	0,7127	0,4360	31,02%
	3.7.1	MORE	84	60	71,4	0,7998	0,5713	

UML sequence diagrams we did not compute the alignment, assuming a misalignment similar to the one computed for the class diagrams. The assumption is motivated by the observation that sequence diagrams have been built starting from class diagrams. Table I reports the alignment measures.

The experiment adopts a counterbalanced design. This design was chosen because: it controls order effects, mitigates possible learning effects between labs and ensures that each subject works with both the treatments (LESS and MORE) in the two lab sessions. Subjects were split into four groups<sup>5</sup>, each one working in Lab 1 on the maintenance tasks of a system with a treatment and working on Lab 2 on the other system with a different treatment. Between the two laboratory sessions a break was given.

For each lab, the subjects had three hours available to complete four maintenance tasks: MT1-MT4. We designed the maintenance tasks in increasing order of difficulty. The maintenance tasks, for the two different systems, are very similar (in practice, three feature modifications and one feature insertion/restructuring for both systems) and of comparable difficulty. For example, a sample maintenance task (i.e., MT1) for the AMICO system asks to simplify the already implemented rule that checks the correctness of the Italian fiscal code of the persons (something similar to the SSN in the United States).

### D. Dependent Variables

The only one dependent variable to be measured in the experiment is *Efficiency* in performing maintenance tasks. It measures the number of correctly performed tasks per minute, and it is defined as:

$$OverallEfficiency = \sum_{i=1}^4 Corr_i / \sum_{i=1}^4 Time_i \quad (1)$$

As it can be noticed from the above formula, the efficiency sums over all (four) maintenance tasks.

The correctness  $Corr_i = 1$  if the  $i$ -th maintenance task was correctly performed, 0 otherwise. The time was measured by means of time sheets. Students recorded start and stop time for each implemented maintenance task.

### E. Material, Procedure and Execution

The development environment was Eclipse 4.2. For each group, we prepared a zip file containing an Eclipse project of the software system (EasyMarks or AMICO) and the related documentation. The zip files were made available on a Web server. The experiment was introduced as a laboratory assignment about software maintenance. Before the experiments, all the subjects have been trained with a two hours tool-demo.

Every subject received:

- an Eclipse project, a high level system functionality description and UML documents (class and sequence diagrams);

<sup>5</sup>We equally distributed (as much as possible) high and low ability students among these groups.

- instructions to set-up the assignment (how to download the zipped Eclipse project, import and execute it);
- a sheet containing the four maintenance tasks.

For each Lab session, the experiment execution followed the steps reported below:

- 1) Subjects were required to download from a given URL the zip file containing the Eclipse project and the documentation.
- 2) Subjects were given 15 minutes to: (i) read the high level description of the system, (ii) import the corresponding project in Eclipse and (iii) execute it.
- 3) A sheet containing the four maintenance tasks was delivered.
- 4) For each maintenance task (MT1-MT4) subjects had:
  - a) to execute the maintenance tasks (for EasyMarks or AM-ICO) on the Eclipse project surfing the UML documents.
  - b) to record the time they need to execute the modification (start/stop time).

Finally, subjects were asked to compile a post experiment questionnaire<sup>6</sup>, aimed at both gaining insights about the students’ behaviour during the experiment and finding motivations for the quantitative results.

### III. PRELIMINARY RESULTS

This section reports some preliminary results from the performed experiment, analysing only the effect of the treatment (LESS or MORE) on the Efficiency dependent variable. More detailed analyses (e.g., analysis of co-factors) and analysis of the post experiment questionnaires are not presented here for space reasons. Results of statistical tests are considered to be significant for a significance level of 95%.

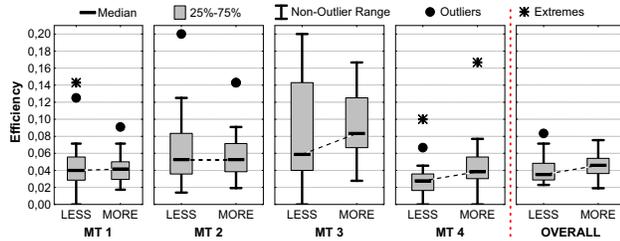


Fig. 1. Boxplots of Efficiency

Figure 1 summarizes the distribution of the Efficiency variable by means of boxplots. Observations are grouped by treatment and application and shown partitioning by maintenance task (MT1, MT2, MT3, MT4) and in aggregate form (Overall). The y-axis represents the efficiency computed as explained in the previous section. The boxplots show that the subjects achieved a better efficiency level when accomplishing the tasks with the MORE treatment. This is more evident for MT3 and MT4 tasks (see the slopes of the lines connecting the medians of the distributions in Figure 1).

We test the difference in efficiency with a paired non-parametric statistical test<sup>7</sup>. By applying a one-tailed Wilcoxon rank-sum test, we found the overall difference to be statistically significant ( $p\text{-value}=0.04$ ), therefore we can reject our null hypothesis ( $H_{0\alpha}$ ).

TABLE II. DESCRIPTIVE STATISTICS OF THE EFFICIENCY VARIABLE AND ONE WAY WILCOXON TEST

Exp	LESS				MORE				p-value
	Subjects	Median	Mean	SD	Subjects	Median	Mean	SD	
Overall	21	0.035	0.040	0.016	21	0.046	0.046	0.014	<b>0.04</b>
MT1	21	0.040	0.047	0.034	21	0.042	0.041	0.018	0.41
MT2	21	0.053	0.066	0.044	21	0.053	0.058	0.034	0.28
MT3	21	0.059	0.084	0.061	21	0.083	0.099	0.061	0.38
MT4	21	0.028	0.027	0.023	21	0.038	0.044	0.033	<b>0.009</b>

To complete the picture, Table II reports the essential descriptive statistics for the Efficiency variable. The variability, expressed as standard deviation, is mostly comparable across groups (except for MT1).

#### A. Discussion

Globally, the results confirm the general belief that a more aligned documentation helps more during maintenance tasks than a less aligned documentation, thus increasing our awareness on the benefit deriving from the usage of aligned UML documents. Looking at the means, students conducting the maintenance tasks with the MORE treatment obtained an overall benefit of +15%<sup>8</sup> in terms of efficiency. This improvement, that could appear small, has to be considered in relationship with the “distance” between the documentation versions. In fact, the bigger the distance between documentation versions is, the bigger the benefit in terms of efficiency should be. That distance, averaged by application and computed considering only class diagrams, is in our case 29.46% (mean of the values shown in the last column of Table I). Thus, in our case the subjects gained benefits in efficiency equal, in percentage, to approximately the half of the computed misalignment.

The study also opens a number of interesting discussion points and challenging aspects that we encountered during the design and execution of the experiment:

**Documentation artefacts selection.** During the design of the experiment, the first problem we had was which kind of documentation artefacts to consider and at which level of abstractness. Finally, we opted for a documentation composed by a high level system functionality description and UML diagrams (class and sequence diagrams) because this kind of documentation is often used in industry. Moreover, we decided to use software perspective UML diagrams at implementation level; these show in detail the classes in the system and how they interrelate. This last choice was done for two reasons: (1) because “low level” documentation is usually more used than “high level” one in maintenance tasks, (2) to simplify the documentation-code alignment measure (see below).

**Maintenance tasks selection.** It is difficult to select/find the “right” maintenance tasks for the experiment and fine-tune their difficulty/complexity. With the word “right” we mean tasks in which the selected documentation provides a concrete benefit. In particular, we would like that the two compared documentation versions differ in the portion used by the subjects to execute the maintenance tasks. In [1], the authors conclude that for complex tasks the availability of UML documents results in significant improvements of the functional correctness of maintenance tasks. On one hand, it is clear that the documentation is not very useful for really simple maintenance tasks but, on the other hand, it is not possible put too much stress on the experimental subjects with really

<sup>6</sup><http://softeng.disi.unige.it/2013-DocAccuracy.php>

<sup>7</sup>We opted for non-parametric tests because of the limited sample size.

<sup>8</sup>Computed using the equation:  $0.040 + 0.040x=0.046$ , see Table II.

complex tasks in a limited amount of time (we recall that in our case students had only 3 hours to complete the maintenance tasks). Moreover, measuring a priori the difficulty/complexity of a maintenance task is really difficult if not impossible. We designed the maintenance tasks in increasing order of complexity but checking it a posteriori we discovered that, for the students, MT3 resulted the simplest task (see Fig. 1). The problem of fine-tuning the difficulty of the maintenance tasks could explain why in our experiment only the difference in efficiency of completing MT4 was significant. We could speculate that the first three tasks were too simple for revealing a significant effect.

**Documentation usage.** It is difficult to measure how much the documentation has been really used by the subjects during the experiment. Clearly, the results of the experiment will be untrustworthy and useless if the documentation is not properly consulted and used during the maintenance tasks [3]. To try to measure it, we used a time tracking software<sup>9</sup>, i.e., a software that allows its users to record time spent on different tasks. The tool we selected counts the time of the active window. In our case, this was only a partial successful solution because in some cases some students forgot to: (1) select the consulted window (and so render it active) and, (2) reset the time tracking tool when starting each task, thus, partially invalidating the tracking.

**Documentation-code alignment measure.** One of the problems we had was how to quantify the alignment between the two selected versions of the documentation and the code. We resorted to a simple, inaccurate and partial way to measure the alignment. However, several possible options are available to compute the alignment between UML class diagrams — e.g., one could use the algorithm proposed in [6] instead of using the Jaccard index or consider only the portions of the diagrams useful to execute the maintenance tasks and not the complete diagrams as we did (task related alignment vs. general alignment) — and it is not clear what is the more adapt for our problem. Moreover, given that the provided documentation includes also sequence diagrams, one should also compute the alignment for them and aggregate the results to obtain an overall measure. We omitted it in this preliminary work because computing the alignment between sequence diagrams and source code is not a trivial task and combining the measures of different types of alignments (class and sequence diagrams) is still more complex.

**Documentation versions selection.** Ideally, for a successful conduct of the experiment one should find/select two “right” documentation versions for the target maintenance tasks. Starting from the last version of the documentation (the more aligned), and selecting it as MORE treatment, there are two options: 1) artificially creating the LESS documentation with the wanted misalignment or 2) performing a careful manual analysis of all the available versions and select “the best one”. Both the options have disadvantages. In the first case, we are not comparing two real versions, in the second one we can not execute the experiment with the wanted “distance” between the two versions. In this preliminary work we opted for the second solution but we are well-aware that the choice of the versions is extremely critical and different chosen versions may bring to different results of the experiment. Clearly, this

aspect is related to the problem of selecting the maintenance tasks and measuring the alignment between versions.

## B. Threats to Validity

This section discusses threats to validity that could have affected our results [4].

**Internal validity threats.** Since the students had to perform a sequence of four maintenance tasks, a learning effect may intervene. However, the students were previously trained and the chosen experimental design should limit this effect. Thus, we expect learning has not influenced too much the results.

**Construct validity threats.** We are well-aware that the adopted alignment measure is preliminary. However, the values in Table I correspond to our qualitative evaluation. Finally, the correctness assessment was performed by one of the authors who inspected the provided code and run a set of test cases. It is possible that the used test suite does not provide an adequate means to measure the quality of the maintenance tasks implementation.

Threats to *conclusion validity* can be due to the sample size (only 21 students) that may limit the capability of statistical tests to reveal any effect. We chose to use non-parametric tests due to the size of the sample.

Threats to *external validity* can be related to: (i) the choice of simple systems as objects, (ii) the use of students as experimental subjects and (iii) documentation used and tasks chosen. Further controlled experiments with larger systems and more experienced developers are needed to confirm or contrast the obtained results.

## IV. CONCLUSION AND FUTURE WORK

A pilot study is usually carried out before large-scale experiments, in an attempt to avoid time being wasted on an inadequately designed experiment. Thus, the result of our pilot have been twofold: (1) a first confirmation of the general belief that an aligned documentation helps during maintenance tasks and, (2) discovering a list of challenging aspects (and possible ways to face them) to consider in future. Future work will be devoted to: address the challenges that we faced and replicate this experiment in different contexts, i.e., with different applications and different subjects (e.g., professionals).

## REFERENCES

- [1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, “The impact of UML documentation on software maintenance: An experimental evaluation,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 365–381, Jun. 2006.
- [2] W. J. Dzidek, E. Arisholm, and L. C. Briand, “A realistic empirical evaluation of the costs and benefits of UML in software maintenance,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 3, pp. 407–432, May 2008.
- [3] G. Garousi, V. Garousi, M. Moussavi, G. Ruhe, and B. Smith, “Evaluating usage and quality of technical software documentation: an empirical study,” in *Proceedings of EASE 2013*. ACM, 2013, pp. 24–35.
- [4] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.
- [5] E. Astesiano, M. Cerioli, G. Reggio, and F. Ricca, “Phased highly-interactive approach to teaching UML-based software development,” in *Symposium at MODELS 2007*. University of Goteborg, 2007, pp. 9–19.
- [6] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Gueheneuc, “Madmatch: Many-to-many approximate diagram matching for design comparison,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1090–1111, 2013.

<sup>9</sup><http://www.kviptech.com/time-tracking-software/free-time-tracker.shtml>

# Task-Driven Software Summarization

Dave Binkley<sup>1</sup>, Dawn Lawrie<sup>1</sup>, Emily Hill<sup>2</sup>, Janet Burge<sup>3</sup>, Ian Harris<sup>4</sup>,  
Regina Hebig<sup>5</sup>, Oliver Keszocze<sup>6</sup>, Karl Reed<sup>7</sup>, John Slankas<sup>8</sup>

<sup>1</sup>Loyola University Maryland, Baltimore, MD, USA

<sup>2</sup>Montclair State University, Montclair, NJ, USA

<sup>3</sup>Miami University, OH, USA

<sup>4</sup>University of California, Irvine, CA, USA

<sup>5</sup>Hasso Plattner Institute at the University of Potsdam, Germany

<sup>6</sup>University of Bremen, Germany

<sup>7</sup>La Trobe University, Australia

<sup>8</sup>North Carolina State University, NC, USA

binkley@cs.loyola.edu, lawrie@cs.loyola.edu, hillem@mail.montclair.edu, burgeje@miamioh.edu harris@ics.uci.edu,  
regina.hebig@hpi.uni-potsdam.de, keszocze@informatik.uni-bremen.de k.reed@latrobe.edu.au, jbslanka@ncsu.edu

**Abstract**—There is a growing interest in software summarization and tools for automatically producing summaries. Discussions of relevant papers at recent conferences led to the observation that software summarization needs to consider migrating away from “is this a good summary?” and towards “is this a useful summary?” As a result, it has been suggested that to judge usefulness, one needs to view the summary through the lens of a particular task. A preliminary investigation of this suggestion was undertaken at the 2013 ICSE workshop NaturalISE. Initial results and lessons learned from this investigation support the notion that task plays a significant role and thus should be considered by researchers building and accessing automatic software summarization tools.

**Index Terms**—Source code summary, summarization evaluation, task-oriented

## I. INTRODUCTION

Comprehending source code is hard. A well written summary can aid a software maintainer in this task. The challenge here is that “well written” can depend on the current task. For example, *Renee the Reuser* benefits from context information, best practices for the effective use of the code, and the identification of replicated code. In contrast, *Teddy the Tester* benefits more from summaries focused on the functionality. Current summaries found in source code (i.e., comments) are often general, out of date, and even incorrect. They may, therefore, be inadequate for a given task. This is largely due to the fact that comments are written by engineers intimately familiar with the code, who can end up writing comments of use only to someone who already understands the code.

Recently, software engineering researchers have begun tackling the problem of automatic summarization of source code [1], [2], [3], [4], [5], [6]. The goal of automatic text summarization can be stated as “to generate a brief yet accurate representation (or summary) of a source document. An ideal summary is significantly shorter than the source document but retains its important information” [5]. What is missing from this notion of summarization is the potential influence of the consumer of the summary. The underlying question raised in this paper is

*Does effective summarization require a specific target task or can source code be summarized in the abstract?*

While this could be the subject of a formal hypothesis test, the results reported herein are of a small experiment conducted recently as an indication of possible outcomes.

One of the major obstacles for source code summarization is that unlike natural language text, where all school children practice writing summaries, developers do not have as much practice summarizing computer programs. Therefore, they do not have as strong an innate sense for what should be part of a software summary.

Because of the challenges assessing the quality of automatic summarization’s output, current empirical study tends to evaluate based on *goodness*, where a collection of programmers are asked if the summary satisfies some goodness measure, for example, if the summary is readable. While this assessment represents an important first cut, a more realistic assessment considers a summary’s (down stream) *usefulness* on a software maintenance task. Does the summary help an engineer accomplish a task, perform it better or faster? To encourage movement beyond *good*, this paper presents a pilot study aimed at investigating how a target task impacts the content of a summary. The following section introduces the state of the art in text and source code summarization. Then the next section describes the target tasks and presents a pilot study for creating summaries for the target tasks. The paper concludes with lessons learned and a summary.

## II. STATE OF THE ART

Beginning with Luhn, the automatic summarization of natural language text has been of interest for over fifty years [7]. As the field matured, template summaries and extractive summaries were proposed [8]. In template summaries, the system fills in blanks using text from a document. These types of summaries are limited by the scope of the templates. Extractive summaries extract sentences from documents and paste them together to form a summary. An extractive summary is more general purpose and can be used to summarize multiple

documents. In addition, it nicely avoids the difficult task of generating language; however, even the best summarizers (*i.e.*, humans) cannot produce good extractive summaries because they are hampered by the need to use sentences found in independent articles [9]. Another challenge is that a summary should include important facts, requiring highly subjective and content dependent judgments.

Recent summarization tasks have attempted to address these problems by disqualifying extractive summaries and giving more direction to both systems and humans when creating the summaries. One representative example of this is the Guided Summarization task at the Text Analysis Conference (TAC) where systems compete to produce the best summaries of ten newswire documents [10]. The summaries consist of 100 words and must include some predefined aspects based on the topics of the ten documents. For example, documents in the health and safety domain must address five aspects such as “who is affected by the health/safety issue.” Human assessors then evaluate the automatically-produced summaries for readability, content, and overall responsiveness.

Automatic summaries are commonly evaluated relative to human produced summaries using the ROUGE (Recall Oriented Understudy for Gisting Evaluations) score, which evaluates an automated summary relative to a set of human summaries [11]. In particular the human summaries are reduced to  $n$ -grams, which are runs of  $n$  words from a document. For example, ROUGE-2 is based on the set of all the bi-grams in all the human summaries. The score for a summary is the number of  $n$ -grams from the human summaries found in the automated summary divided by the total number of unique  $n$ -grams in the human summaries.

Moving from automatic summarization of general texts to the summarization of software, the state of the art is surveyed by considering six recent projects. Early work on software summarization includes Murphy’s dissertation on summarizing structural information in source code [1]. She studied how semi-automatic (iterative) summaries can enable an engineer to assess, plan, and execute changes to a software system. While this study was done in the context of software evolution, it did not consider producing software evolution specific (*i.e.*, task specific) summaries.

More recently Moreno and Aponte compared tool-generated summaries with those created by Java developers [12]. They discovered that developers create summaries of similar length for all types of entities (*e.g.*, methods and classes) while the length of automatically generated term-based summaries correlated with the length of the artifact summarized. They conclude that to get the gist of source code artifacts automatically, the length of a term-based summary should range from ten to twenty words nearer ten for methods and twenty for packages. One approach to encourage a target length is the use of template-based summaries such as those described in Sridhara’s dissertation [2], where templates are used in the automatic generation of comments to summarize Java methods, collections of statements, and formal parameters.

A similar approach was recently presented by Moreno et al. who aim to generate human readable summaries for Java

classes using class and method stereotypes [3]. While the summaries focus on content and responsibilities of a class rather than its relationships with other classes, they fall short of being task specific. When asked, programmers judged the generated summaries readable and understandable. This evaluation leaves open the *usefulness* of the summaries for a particular task, which was pointed out in the discussion following the presentation of this work at the conference.

Finally, at ICPC 2013 Gail Murphy’s group presented two summarization approaches that hint at the need for task specific summarization. In the first, Rastkar and Murphy propose the use of multi-document summarization techniques to generate a natural language description of why code changed [6]. Their approach is extractive as it extracts full sentences to form a summary from the documents related to the change. Initial results show that overall developers found the summaries to contain information related to the reason behind the code change. However, they also note the evaluation needs to go further and investigate whether developers find the generated summaries *useful*. Doing so is likely to motivate the incorporation of task specific summarization techniques.

In the second approach, Kamimura and Murphy observe that automatic test generators (*e.g.*, CodePro) produce code that can be difficult to comprehend [13]. They propose a technique for generating human-oriented summaries of test cases aimed at improving a humans’ ability to quickly comprehend unit tests. They conclude by noting that “much more work is needed to make truly usable human oriented summaries.” Here again this future work is likely to consider task summarization specifically *useful* to the task.

### III. PILOT STUDY

From the previous section, it is evident that recent work on automatic summarization hints at taking task into account. As a precursor to the construction of such tools, the case study presented in this section considers *task specific manually produced summaries*. These are useful in assessing the role task plays. On the one hand, if task does not play a role in manually produced summaries then it is unlikely to play a role in automatic summaries. However, if it does play a role then there is a need for manually produced task-specific summaries against which to measure automatic tools.

For the case study, the participants of the 2013 NaturaLiSE workshop formed pairs to produce summaries of two classes selected from the program jEdit 4.2 by the first two authors. Over the course of an hour and a half, each pair examined one or both of the classes and produced two summaries for each class, one aimed at a reuser and the other at a tester. This section first describes the two tasks in greater detail, then the two selected classes, and finally the analysis of the summaries produced.

#### A. Tasks

To investigate the impact that task has on summarization, two tasks were considered. The tasks were personified by

Table I  
A COMPARISON OF DisplayManager SUMMARIES TO StatusBar SUMMARIES. (BOLD ENTRIES SHOW TASK DOMINANCE.)

	StatusBar-Renee	StatusBar-Teddy
DisplayM-Renee-1	<b>0.148</b>	0.068
DisplayM-Teddy-1	0.076	0.072
DisplayM-Renee-2	<b>0.124</b>	0.090
DisplayM-Teddy-2	0.072	<b>0.084</b>
DisplayM-Renee-3	<b>0.060</b>	0.055
DisplayM-Teddy-3	0.065	<b>0.132</b>

```
public void appendPosition(String tag,
    int start, int end) {
    LinkedList<TMarkedStoreItem> ll =
        tagMap.get(tag);
    if (ll == null) {
        ll = new LinkedList<TMarkedStoreItem>();
        tagMap.put(tag, ll);
    }

    TMarkedStoreItem item =
        new TMarkedStoreItem();
    ll.add(item);
    item.end = end;
    item.start = start;
}
```

Figure 1. The method `appendPosition` for the program `JabRef`

*Renee the Reuser and Teddy the Tester.* To illustrate how summaries might differ given the specific tasks, Figure 1 shows a method found in `JabRef` 2.6. A summary directed towards Renee the Reuser might read:

Dear Renee,  
Please be aware that this method creates lists of position pairs based on the `<tag>`.

In contrast, a summary directed towards Teddy the Tester might read:

Dear Teddy,  
Please consider calling the same `<tag>` twice, once to test found and once to test not found.

While this example is simplistic, it is nevertheless evident that task driven summaries should emphasize different aspects of the code.

### B. Code to Summarize

The classes for which the summaries were produced came from `jEdit` 4.2, a text editor for source code. This program was selected because it was presumed that workshop participants would be familiar with the domain vocabulary.

The two classes selected were `DisplayManager` and `StatusBar`. From each class multiple methods were also selected. The instructions asked participants to provide a summary for the chosen methods and then the class as a whole. For the `DisplayManager` class, the methods `expandFold` and `narrow` were chosen. Quantitatively `DisplayManager` includes 868 LoC and the two methods 111 LoC and 30 LoC respectively. For `StatusBar`, the methods `propertiesChanged`, `statusUpdate`, and `updateBufferStatus` were selected. Here `StatusBar` includes 482 LoC and the methods 69 LoC, 134 LoC, 9 LoC, respectively.

To give the reader a sense of the task of summarizing `DisplayManager`, understanding what a fold is within the editor along with the data structures that keep track of the folds are important background for writing a summary. The implementation uses parallel arrays of integers rather than, for example, creating a single array of fold descriptors. Considering `StatusBar`, the class visually displays information, which means that seeing the interface is likely to aid a person writing a summary.

### C. Analysis

Pairs of participants examined the source code and then wrote two summaries (one for each task). One pair failed to identify the two separate summaries for the two tasks. Of the remaining three, only one pair considered both classes due to limited time. Thus, in the end, eight summaries were produced: two for `StatusBar` and six for `DisplayManager`.

To analyze the summaries, statistics were collected and summaries were compared pairwise using cosine similarity [14], which was computed by creating a vector space model in which each summary is treated as a bag of words comprised of word weights. The standard term frequency-inverse document frequency (tf-idf) was used to weight the words in each document's vector from which the cosine similarity was computed.

The eight summaries contained from between 18 and 103 non-stops words. The mean number of non-stop words was 42.5. A standard English language stop list was used for identifying the stop words, as summaries did not contain source code. The summaries were also stemmed using `kstem` [15] before computing cosine similarity, which is a standard practice in natural language to conflate words with different suffixes. When examining the unique stemmed vocabulary, summaries contained between 11 and 43 unique words with an average of 25.1 words. The total number of unique words used in all the summaries was 120, so a word occurred on average three times across the summaries.

Several observations concerning the summaries can be made by examining the similarity between pairs of summaries. The first observation is that there is evidence that there are words that are specific to testing and others that are specific to reuse. This can be seen in Table I, where the two `StatusBar` summaries are compared to the `DisplayManager` summaries. In the table, the summaries are labeled by task, followed by class name, and finally, for `DisplayManager` a number indicating the pair that produced the summary. The table reports the cosine similarity. In 83% of the cases (5 of 6), the summary for the particular task is more similar to the summary of the same task than the other task. These cases are shown in bold in the table. These patterns manifest in the summaries in the vocabulary used. For example, testing summaries included phrases such as “tester,” “true and false,” “range,” and “completely test” while reuse summaries include “overall” and “the parameters.”

The second observation comes from comparing the data in Table I with that shown in Table II, which shows the pairwise similarities between all pairs of `DisplayManager` summaries. First notice that the diagonal has all ones in it. This is expected

Table II  
COSINE SIMILARITY COMPARISON OF DisplayManager SUMMARIES TO DisplayManager SUMMARIES

	DisplayM-Renee-1	DisplayM-Teddy-1	DisplayM-Renee-2	DisplayM-Teddy-2	DisplayM-Renee-3	DisplayM-Teddy-3
DisplayM-Renee-1	1.000	<b>0.434</b>	0.205	0.311	0.212	0.127
DisplayM-Teddy-1	<b>0.434</b>	1.000	0.322	0.293	0.202	0.150
DisplayM-Renee-2	0.205	0.322	1.000	<b>0.363</b>	0.179	0.140
DisplayM-Teddy-2	0.311	0.293	<b>0.363</b>	1.000	0.179	0.215
DisplayM-Renee-3	0.212	0.202	0.179	0.179	1.000	<b>0.579</b>
DisplayM-Teddy-3	0.127	0.150	0.140	0.215	<b>0.579</b>	1.000

when one compares something to itself because the vectors are identical. When considering the two tables together, the values in Table II are nearly all larger than the largest value in Table I, showing a dominance of terms from the domain of the code as opposed to the domain of the task (testing versus reuse). This is further supported by the fact that summaries for Renee are not necessarily more similar to each other than they are with summaries for Teddy in Table II, where all summaries are of the same source code.

Finally, Table II shows the expected result that the authors of the summary are more similar to themselves, moving between tasks than they are to summaries written for a particular task. This is born out in the similarity scores that appear in bold, which are the second largest in each row and likely indicate the importance of personal word choices. Examples causing authors to be more similar to themselves include the use of the term “method” versus “function”.

#### IV. LESSONS LEARNED

After producing the two summaries, an open discussion was held by all participants to gather impressions and lessons learned. The dominant take home message was that summarizing non-trivial unfamiliar code is extremely challenging. This manifests itself in requests for the kind of information traditionally held in a data dictionary (e.g., acronym expansions, a data structure summaries, and an explanation of vague variable names).

Several participants suggested that dynamic information would have been helpful. For example, movies showing the Status Bar GUI in action or an example of a fold operation. This hints at potential future work on incorporating dynamic information into source code summarization.

Finally, a handful of the comments were meta comments regarding the summarization process itself. These included crowd sourcing the process to involve more geographically-separated researchers in developing a shared corpus and an appreciation for the two codes used not being trivial or just *academic exercises*. One pair writing the reuse summary first found the test summary easier to produce (and the resulting summary shorter). This is likely evidence of a learning effect. Finally, unlike the summarization of more standard natural language texts such as news articles, it was not clear to participants what level of detail was needed for the summarization. For example, one participant asked “does Renee need to understand how the code works to re-use it or is it sufficient just to know what it does?” Such a question further motivates the study of task-driven software summarization.

#### V. CONCLUSION

Task driven summarization is an important direction for software summarization as it provides focus for the summaries, which is evident in the fact that a person produces different summaries for different tasks. However, the task must be well defined. The development of aspects for the task, as in the TAC Guided Summarization Task, is most likely needed to produce greater consistency among human generated summaries. These could then be used as gold sets against which system performance could be judged using, for example, the ROUGE score.

#### ACKNOWLEDGMENT

Thanks to all workshop participants and in particular Alexandre Bassel and Michael Pradel for their help with the summaries. Support for this work was provided in part by NSF grant CCF 0916081.

#### REFERENCES

- [1] G. Murphy, “Lightweight structural summarization as an aid to software evolution,” PhD thesis, University of Washington, Washington, DC, USA, 1996.
- [2] G. Sridhara, “Automatic generation of descriptive summary comments for methods in object-oriented programs,” PhD thesis, University of Delaware, 2012.
- [3] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, “Automatic generation of natural language summaries for java classes,” in *International Conference on Program Comprehension*, 2013.
- [4] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, “On the use of automated text summarization techniques for summarizing source code,” in *Proceedings of the Working Conference on Reverse Engineering*, 2010.
- [5] B. Eddy, J. Robinson, N. Kraft, and J. Carver, “Evaluating source code summarization techniques: Replication and expansion,” in *International Conference on Program Comprehension*, 2013.
- [6] S. Rastkar and G. Murphy, “Why did this code change?” in *International Conference on Program Comprehension*, 2013.
- [7] H. P. Luhn, “The automatic creation of literature abstracts,” *IBM Journal of research and development*, vol. 2, no. 2, pp. 159–165, 1958.
- [8] I. Mani, *Automatic summarization*. John Benjamins Publishing Company, 2001, vol. 3.
- [9] P.-E. Genest, G. Lapalme, and M. Yousfi-Monod, “Hextac: the creation of a manual extractive run,” in *Proceedings of the Second Text Analysis Conference, Gaithersburg, Maryland, USA. National Institute of Standards and Technology*, 2009.
- [10] K. Owczarzak and H. Dang, “Overview of the tac 2010 summarization track,” in *Proceedings of TAC 2010*, 2010.
- [11] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, 2004, pp. 74–81.
- [12] L. Moreno and J. Aponte, “On the analysis of human and automatic summaries of source code,” *CLEI ELECTRONIC JOURNAL*, vol. 15, no. 2, 2012.
- [13] M. Kamimura and G. Murphy, “Towards generating human-oriented summaries of unit test cases,” in *International Conference on Program Comprehension*, 2013.
- [14] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge Press, 2008.
- [15] R. Krovetz, “Viewing morphology as an inference process,” in *Proceedings of the 16th ACM SIGIR Conference*, R. K. et al., Ed., June 1993.

# Determining “Grim Reaper” Policies to Prevent Languishing Bugs

Patrick Francis

Industrial Software Systems  
ABB Corporate Research  
Raleigh, NC, USA  
patrick.francis@us.abb.com

Laurie Williams

Department of Computer Science  
North Carolina State University  
Raleigh, NC, USA  
williams@csc.ncsu.edu

**Abstract**— Long-lived software products commonly have a large number of reported defects, some of which may not be fixed for a lengthy period of time, if ever. These so-called *languishing bugs* can incur various costs to project teams, such as wasted time in release planning and in defect analysis and inspection. They also result in an unrealistic view of the number of bugs still to be fixed at a given time. The goal of this work is to help software practitioners mitigate their costs from languishing bugs by providing a technique to predict and pre-emptively close them. We analyze defect fix times from an ABB program and the Apache HTTP server, and find that both contain a substantial number of languishing bugs. We also train decision tree classification models to predict whether a given bug will be fixed within a desired time period. We propose that an organization could use such a model to form a “grim reaper” policy, whereby bugs that are predicted to become languishing will be pre-emptively closed. However, initial results are mixed, with models for the ABB program achieving F-scores of 63-95%, while the Apache program has F-scores of 21-59%.

**Keywords**—*languishing bugs, software management, policy*

## I. INTRODUCTION

Inevitably, through the course of software development engineers inject defects into program code. As a software project sees increasing adoption, its creators may then be “rewarded” with an increasing number of defect reports. The defect tracking database of mature software projects, therefore, may contain several thousands of defects, if not more.

Unfortunately, most software projects have a finite amount of resources to devote to bug fixing, which may be exceeded by the quantity of reported bugs. In such cases, a project may accumulate an ever-expanding backlog of unfixed defects. Software managers must prioritize maintenance efforts, often based on factors such as defect severity, effort to fix, importance to the customer, or impact on the code base. However, this prioritization activity may result in defects that are repeatedly given a low priority or deferred to a future release. We use the term *languishing bug* to refer to the defects that remain open and unfixed for a lengthy period of time.

In many cases, project members understand that these languishing bugs are unlikely to ever be fixed. However, various costs may be incurred by having these open defects languishing in the bug database. For example, these defects may need to be

repeatedly analyzed to determine whether they should be targeted for an upcoming release. They likewise distort a project’s bug metrics and hinder efforts to form a realistic view of the remaining work. Furthermore, any time spent investigating and debugging a defect that is ultimately never resolved is largely wasted effort.

This problem of an overwhelming stream of incoming tasks will be familiar to anyone with an overflowing email inbox. Many people receive emails faster than they can process them, until they have collected an infeasible backlog of unanswered messages. One strategy for dealing with this problem is to declare “email bankruptcy”, and simply delete (or ignore) any unanswered emails older than some given age. [5] Any email that was actually important will presumably be resent. We hypothesize that a similar approach can be effective for managing languishing bugs.

The goal of this work is to *help software practitioners mitigate their costs from languishing bugs by providing a technique to predict and pre-emptively close them*. We propose that software teams develop a “grim reaper” policy, named after the personification of death. This policy would use a predictive model to identify potentially languishing bugs, which could then be closed, acknowledging the reality that they are unlikely to be fixed. The bugs could be reopened if there are any future duplicate reports, but in the meantime developers are free from having to manage them.

To further our goal, we formulate several research questions:

RQ1: How prevalent are languishing bugs in a software project?

RQ2: How well can a model be learned from the defect data to predict which defects will languish?

To answer these questions we examined the defect repositories from two subject programs: an embedded program developed by ABB Inc., and the Apache HTTP server. We measured the duration that each defect was open and analyzed the distributions of these values. We have also devised and evaluated a technique for constructing a grim reaper policy, based on historical defect data. The technique includes a decision tree model trained to identify bugs that are unlikely to be fixed within some desired period of time, and are therefore considered languishing. We have trained several models for

varying values of parameters and assessed their accuracy at predicting languishing bugs.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents data on languishing bugs in our subject programs. Section IV discusses grim reaper policies and our results in mining them from historical data. Section V concludes and discusses future work.

## II. RELATED WORK

Many previous researchers have looked at predicting measures about bugs, such as the fix time. For example, Anbalagan and Vouk [1] predicted fix times for bugs in Ubuntu based on the number of people involved in the report. Hooimeijer and Weimer [7] analyzed Firefox bugs to predict whether they would be triaged within a given time. Bhattacharya and Neamtiu [3] trained fix time models on several large open source projects and found that features that other authors had reported as predictive performed poorly on these projects. Guo et al. [6] studied the characteristics, such as the reputation of the submitter, that distinguish bugs that get fixed from those that do not. They also trained a logistic regression model to predict whether a given bug will ever be fixed. Our work differs from these in that it seeks to predict whether a bug will be fixed, rather than just triaged, and fixed within a given time period. Also, we use only the data directly available in the bug report, rather than computing our own features, such as the submitter reputation.

The work that is perhaps most similar to ours is by Zhang et al. [9]. They analyzed several commercial projects and used a k-Nearest Neighbor model to predict whether a given bug would be fixed slowly or quickly, i.e. above or below a given time threshold. However, our work is distinguished by our proposal to use such a model to form a policy for pre-emptively closing bugs.

## III. PREVALENCE OF LANGUISHING BUGS

Before developing solutions for managing languishing bugs, we first need to assess how common they are. Therefore, our first research question is: How prevalent are languishing bugs in a software project?

### A. Subject Programs

We examined two subject programs. The first is a proprietary embedded program developed by ABB Inc., which we refer to as ABB1. The program contains approximately 2.1 million lines of code, written primarily in C and C++, and has been developed since 2000. We extracted the defect data directly from the bug database and included all available data as of July 2012.

The second subject program is the open source Apache HTTP server, version 2.x. [2] This program contains 340 thousand lines of code, written in C and C++. We collected the defect data from the project’s Bugzilla web page and included all available data as of March 2013.

For each bug, we calculated its Fix Time, which we define as the time (in days) between when the bug was opened and when it was closed. For ABB1, the bug data includes both of these dates as fields. Apache includes only the Opened Date, and so we extracted the Closed Date from the change history, using

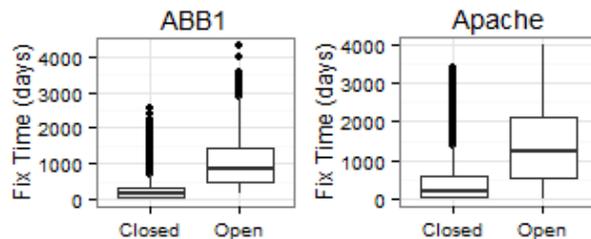


Fig. 1. Distributions of defect fix times for ABB1 and Apache, excluding defects that were closed without a fix.

the date when the Status field was most recently set to a closed status. For bugs that were still open, and therefore had no Closed Date, we instead used the date that we extracted the data from the database. Therefore, the “Fix Time” for an open bug is equivalent to the age of the bug.

### B. Distribution of Defect Fix Times

For both subject programs, we found that the fix times are highly skewed toward short fix times. Nevertheless, a substantial proportion of the defects have fix times that are much longer than average. Fig. 1 summarizes the distributions of fix times for defects that are currently open (as of the data extraction date mentioned in Section III.A) and ones that are closed. For these boxplots, any defects that are closed as invalid are excluded (e.g. they are marked as Not Repeatable, or Works As Designed, or similar). For ABB1, 23.7% of the defects are open, while for Apache 33.3% of the defects are open.

As shown in the figure, the fix times for open defects are generally much higher than for defects that were closed. For ABB1, the median fix time for open defects is 862 days, versus 131 days for closed defects. In fact, 88% of the open defects have fix times longer than the 75<sup>th</sup> percentile time for the closed defects. Apache is similar; the median fix time for open defects is 1242 days, versus 202 days for closed defects. Furthermore, 73% of the open defects have fix times longer than the 75<sup>th</sup> percentile time of the closed defects.

For both programs, the closed defects do have a number of outliers with very long fix times. However, defects with long fix times are much more likely to remain open rather than be closed, and both projects contain many defects with long fix times. These results show that both subject programs contain a substantial number of languishing bugs that are unlikely to be fixed.

## IV. “GRIM REAPER” POLICY

To avoid the costs and problems associated with languishing bugs, organizations should strive to prevent them from languishing in the first place. However, the obvious solution of resolving all bugs in a timely manner is often not feasible due to resource constraints or market realities. We therefore propose that organizations develop a “grim reaper” policy (named after the personification of death). This policy would use a model to predict whether a given bug is likely to be fixed within some chosen period of time, or whether it will languish. Those bugs that are predicted to be languishing can be pre-emptively closed.

Bugs closed in this way can be given a disposition such as “Languishing”, “Inactive”, or “Expired”, to distinguish them from bugs that have been resolved. Some defect management systems already have a disposition of “Won’t Fix”, to indicate defects that are legitimate but that nevertheless will not be fixed. This idea is the same as what we propose, and the grim reaper policy may serve as a formalized means to apply this disposition.

#### A. Policy Parameters

We consider two primary parameters in constructing a grim reaper policy: the policy application date, and the languishing criteria. Values for these parameters are necessary to train the classification model.

The first parameter is the *policy application date*, which specifies the date at which a given defect should be assessed to see if it should be closed. Two approaches an organization could use to set this parameter include:

- *Bug age*: apply the policy to a bug once it has been open for a certain amount of time. The chosen age might be a fixed time, such as one year or two years, or based on the project data, such as the mean bug fix time plus X%.
- *Release cycle*: apply the policy to all open bugs at a given point in the release cycle, e.g. one month after a release, or mid-way between two releases.

The second parameter for a grim reaper policy is the set of *languishing criteria*. These criteria determine which of the bugs in the repository should be labeled as languishing and are used to train the classification model. That is, at what point should a bug be considered “languishing”? Three possible criteria to determine the languishing bugs are:

- *Bug age*: label as languishing those bugs that have been open for more than some defined period of time. This period might be an arbitrary cutoff such as two years or five years, or it might be determined based on the historical likelihood of a bug being closed after that age.
- *Release cycle*: bugs that have remained open for more than X releases of the product are considered languishing.
- *User interest*: bugs that have not been re-reported for some given period of time are considered languishing.

An organization may also choose to combine several languishing criteria.

#### B. Experimental Setup

To answer RQ2, we have trained a number of decision tree models to predict languishing bugs. As described above, many possible settings for the parameters of policy application date and languishing criteria exist. We selected one scheme for each and evaluated several values to assess the sensitivity of the models to these parameters.

For the policy application date, in this paper we analyze a scheme based on the bug age, and selected values of one year, two years, and three years. That is, each bug would be evaluated against the policy when it reached the given age.

However, to train the models using the correct defect data, each bug needed to be reverted to the state it was in at the desired

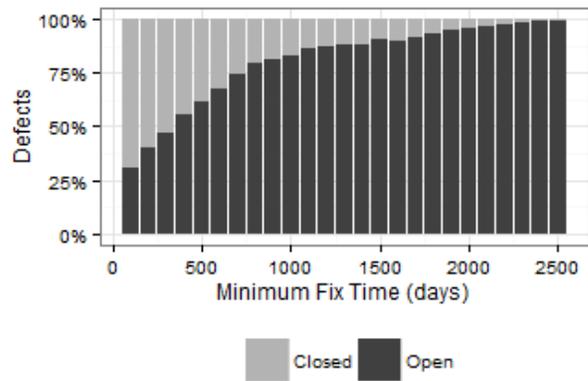


Fig. 2. Proportions of Open and Closed Bugs in ABB1 with fix times greater than the specified values

application age, rather than its most recent state. We reverted the data using the change history in the bug database, which records all the changes made to a given bug. We un-did each change individually, in reverse chronological order, until the bug had been reverted to the desired age (1 year, 2 years, etc.).

To determine which bugs should be labeled as languishing, we used a scheme based on the bug age, using cutoff values chosen based on the historical bug data. We selected the bug ages at which 80%, 90% or 95% of the bugs open at least that long were still open. This idea is illustrated in Fig. 2. The horizontal axis shows the minimum fix time. The vertical axis indicates, among the bugs that were open for at least that time, what proportion were eventually closed (light gray) and what proportion are still open (dark gray). We then identified the minimum ages at which at least 80%, 90% or 95% of the bugs remained open.

For each combination of parameter values, we trained five decision tree classification models, using five-fold cross validation to form the training and testing sets. The models were created using the rpart package in R [8], which implements the CART classification tree algorithm by Breiman, et al. [4]

We trained the models using only those defects that were still open at the given policy application age. We also excluded any defects that were currently open but had a fix time less than the given languishing age cutoff. (Since these defects have not been closed, but have also not existed for long enough to be considered languishing, they cannot be properly labeled as either languishing or not.) We trained the models using all the fields recorded in the bug database, such as Severity, Component or Reported Build. However, we excluded those fields that would be obviously spurious (such as the BugID, or Title) or would contain inaccurate data. For example, several fields from the Apache data were excluded because they were not recorded in the change history and therefore could not be properly reverted to their state at the policy application date. In total, we included 81 fields for ABB1, and 33 fields for Apache. (We do not list them here due to space constraints.)

#### C. Model Results

For our experiment, we used the two subject programs described earlier. For ABB1, the languishing criteria of 80%

TABLE I. PERFORMANCE OF CLASSIFICATION MODELS ON ABB1. EACH CELL SHOWS THE AVERAGE PRECISION, RECALL AND F-SCORE FOR EACH MODEL.

		Languishing cutoff		
		80% open (850 days)	90% open (1500 days)	95% open (2000 days)
Policy application date (defect age)	365 days (1 year)	P: 0.745 R: 0.727 F: 0.736	P: 0.654 R: 0.669 F: 0.661	P: 0.629 R: 0.636 F: 0.631
	730 days (2 years)	P: 0.939 R: 0.961 F: 0.950	P: 0.756 R: 0.768 F: 0.762	P: 0.727 R: 0.709 F: 0.717
	1095 days (3 years)	NA	P: 0.884 R: 0.931 F: 0.906	P: 0.790 R: 0.853 F: 0.820

open, 90% open, and 95% open correspond to 850 days, 1500 days, and 2000 days, respectively. Furthermore, 10.65% of the total bugs have fix times longer than 850 days, 5.19% longer than 1500 days, and 2.34% longer than 2000 days.

For Apache, the languishing criteria correspond to 2300 days, 2950 days, and 3200 days, respectively. Furthermore, 4.89% of the total bugs have fix times longer than 2300 days, 1.95% longer than 2950 days, and 1.29% longer than 3200 days.

The performance of our models is summarized in Tables I and II. Each cell shows the average of the Precision, Recall and F-Score for each model. (For ABB1, the combination of 1095 days and 80% open was omitted because the policy application date is higher than the languishing cutoff.) The models for ABB1 perform relatively well, with F-scores ranging from 63% up to 95%. The performance for Apache, however, is much worse; the F-scores range from 21% to 59%. For both subject programs, the accuracy of the model decreases as the time between the application date and the languishing cutoff date increases. That is, models that have to predict further into the future perform worse.

We also examined the potential costs of using these models in a grim reaper policy. One measure of this cost is the number of bugs that were predicted to languish, but were actually closed as fixed. In other words, how many bug fixes would have been “lost” by closing the bugs early? This number includes both false positives, i.e. bugs mis-predicted by the model, and open bugs that did reach the languishing cutoff age, but were later closed. For ABB1, 10% to 24% of the predicted languishing bugs were eventually actually fixed. With Apache, 34% to 47% of the predicted languishing bugs were fixed.

The poor performance of the models on Apache may be partly explained by the choice of languishing criteria. Both subject programs use the same criteria (80% open, etc.), but with Apache these correspond to much later ages. Since the policy application dates are the same, though, the intervals between the application dates and the cutoff dates are much longer, which correlates with worse predictive accuracy, as mentioned above. Models trained on Apache using an application date of 1 year and languishing cutoffs of 60% open (1050 days) and 70% open (1650 days) showed greatly improved F-scores: 73% and 57%, respectively.

TABLE II. PERFORMANCE OF CLASSIFICATION MODELS ON APACHE. EACH CELL SHOWS THE AVERAGE PRECISION, RECALL AND F-SCORE FOR EACH MODEL.

		Languishing cutoff		
		80% open (2300 days)	90% open (2950 days)	95% open (3200 days)
Policy application date (defect age)	365 days (1 year)	P: 0.474 R: 0.226 F: 0.301	P: 0.473 R: 0.193 F: 0.265	P: 0.349 R: 0.149 F: 0.209
	730 days (2 years)	P: 0.516 R: 0.368 F: 0.422	P: 0.490 R: 0.274 F: 0.339	P: 0.476 R: 0.239 F: 0.313
	1095 days (3 years)	P: 0.578 R: 0.612 F: 0.593	P: 0.509 R: 0.214 F: 0.283	P: 0.520 R: 0.284 F: 0.355

## V. CONCLUSIONS AND FUTURE WORK

In this paper we present an approach to help organizations mitigate their costs from languishing software bugs. Specifically, we propose a “grim reaper” policy whereby a model predicts which bugs will become languishing and the organization pre-emptively closes them. We analyzed two subject programs, one industrial and one open source, and found that they both contained a substantial number of languishing bugs. We then trained decision tree models for nine combinations of policy application date and languishing cutoff date. We found that the models predict languishing bugs with moderate accuracy for ABB1, but did not perform as well on Apache. For both programs, the models performed best when the policy application date and languishing cutoff were closest together. We also measured the costs incurred by the use of this policy, in terms of lost bug fixes. We found that 10% to 47% of predicted languishing bugs were actually fixed. Future work will include detailed study of the costs and benefits of the proposed grim reaper policies. We will also investigate alternative classification methods, such as improved machine learning algorithms or natural language analysis of the defect report text.

## REFERENCES

- [1] P. Anbalagan and M. Vouk, “On predicting the time taken to correct bug reports in open source projects,” in IEEE International Conference on Software Maintenance, 2009, pp. 523–526.
- [2] Apache HTTP Server. <http://httpd.apache.org>
- [3] P. Bhattacharya and I. Neamtiu, “Bug-fix time prediction models: can we do better?,” in Proceedings of the 8th Working Conference on Mining Software Repositories, New York, NY, USA, 2011, pp. 207–210.
- [4] L. Breiman, J. Friedman, R. A. Olshen, and C. Stone, Classification and Regression Trees. Chapman and Hall/CRC, 1984.
- [5] N. Douglas. (2007, April 23). “Declaring Email Bankruptcy”. <http://gawker.com/254608/declaring-e+mail-bankruptcy>.
- [6] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows,” in Proceedings of the 2010 International Conference on Software Engineering, New York, NY, USA, 2010.
- [7] P. Hooimeijer and W. Weimer, “Modeling bug report quality,” in Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering, New York, NY, USA, 2007, pp. 34–43.
- [8] R Core Team, R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. <http://www.R-project.org>
- [9] H. Zhang, L. Gong, and S. Versteeg, “Predicting bug-fixing time: an empirical study of commercial software projects,” in Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, 2013, pp. 1042–1051

## Which Practices are Suitable for an Academic Software Project?

Václav Rajlich

Dept. of Computer Science  
Wayne State University  
Detroit, MI 48202, U.S.A.  
rajlich@wayne.edu

Jing Hua

Dept. of Computer Science  
Wayne State University  
Detroit, MI 48202, U.S.A.  
jinghua@wayne.edu

**Abstract**— This paper presents the practices observed in successful academic projects. It classifies them by the software lifecycle stage they belong to; most of the academic projects are in stage of evolution. It also classifies them by their purpose into the organizational and code development practices. The classification may help academic project managers and developers, who are often specialist in fields other than software engineering, to select the appropriate practices for their project.

**Keywords**- *Computational science and engineering project, software evolution, software change, phased model of software change, iterations, concept location, impact analysis, refactoring, actualization*

### I. INTRODUCTION

Numerous software development projects take place within academic environment. The managers and developers of these projects search for suitable project practices; the experience from the other successful academic projects can serve as their guide.

Recent publications described several academic projects from the domain of computational science and engineering, and identified the practices that the project teams used [1, 2]. Other authors compared academic software processes to the industrial agile software development processes of Scrum and XP [3, 4].

We believe that academic projects need to tailor their software development processes towards their specific circumstances, and select suitable practices from the available set. However the number of the available practices is large and some of them conflict with the others. This paper proposes classification criteria for the software development practices that should help in the selection of the appropriate practices for a specific project.

We note that each programming team has a long list of practices; some of them are obvious and are present in all reasonable projects, hence they do not need to be discussed. We are focusing our attention to the practices that distinguish one software process from another.

Section II presents attributes that we are using for the classification and Section III discusses examples of projects and practices from one directly observed project and two projects from literature; it also presents several recommendations. Section IV presents the summary and future work. Table I in the appendix summarizes three projects discussed in this paper.

### II. ATTRIBUTES OF THE PRACTICES

The practices studied in this paper are classified by their applicable lifecycle stage and by their purpose.

#### A. Stages of Software Lifecycle

An insight into the suitability of practices is offered by the studies of software life-cycles, because different practices are suitable for different life-cycle stages. Staged model of software lifecycle divides the life-cycle of software into several stages [5].

*Initial development* produces the first – and often very preliminary - version of the software. The developers make several fundamental decisions; they select the project architecture and the technology portfolio (i.e. programming languages, libraries, GUI framework, software tools, etc.) These decisions usually accompany the project through the entire life-cycle.

Initial development is a short stage and at any particular time, very few academic projects are in that stage. The practices of design-and-develop are appropriate for that stage. These practices have been explored extensively in software engineering literature and they have been proposed for academic projects [1]. The practices first prepare design using one of the many available design methodologies, and then implement code based on this design. These practices are usable only when developing software from scratch, as they do not address what to do with the existing code.

In the next stage of *software evolution*, programmers add new capabilities and features, and correct previous mistakes and misunderstandings. Evolution is the stage that covers most of the duration of academic projects, and hence it is the focus of this paper. Software changes are the basic building blocks of software evolution and each change introduces a new feature or some other new property into software.

*Exploratory programming* is an extreme form of evolution where the developers do not know in advance details of the features that they are developing, and establish them by trial-and-error. Exploratory programming is very common in research oriented academic development.

Software stops evolving and enters *final stages* for several different reasons. Software may achieve *stability*, where the problem is solved and no longer requires any large evolutionary changes. Evolution also may end for *business reasons*, when the managers decide to stop the expensive evolution and limit changes to a bare minimum. It may also

end involuntarily because of *loss of evolvability*: There can be code decay that makes continuing evolution unfeasible, or the software team loses the skills necessary for evolution. In the latter situation, the developers often resort to superficial changes that confuse and complicate software structure, leading to further code decay.

Final stages of software lifecycle are *servicing*, where the programmers no longer do major changes in the software, but they still make small repairs that keep software usable. When software is not worthy of any further repairs, it enters the *phase-out* stage. Sometime later, the managers or customers completely withdraw the system from the production and this is called *close-down*. The final stages are short in academic projects and this paper does not deal with them.

### B. Organization of the Team

Another insight is offered by division of the practices by their purpose into organizational practices and code development. An example of an organizational practice is self-organizing team, where developers volunteer for tasks, as opposed to the practice of the task assignment by the project leader. Organization of the team is greatly affected by the circumstances in which the team operates, and there is a large variety of the successful team organizations; three examples in Table I differ by their organization. An example of a well-defined and successful team organization is Scrum [6]; the circumstances in which Scrum is effective are discussed in [7].

### C. Code Development Practices in Software Evolution

Compared to the large literature dealing with organizational practices, code development received less attention and is a focus of this paper. This paper uses *Phased Model of Software Change (PMSC)* [5] as the organizing principle and classifies the code development practices by the applicable phases of this model; Figure 1 contains representation of PMSC. Case studies that documented PMSC in software development were published in [8-11]. Other well-known models of software change are special cases of PMSC [12, 13]

*Initiation* phase deals with origin, prioritization, and assignment of the new requirements. Next phase is *concept location* that finds the module or snippet in the existing code that must be changed in order to implement new feature. Concept location can be a small task in small programs, but it can be a considerable task in large or unfamiliar programs [14]. Well-known techniques of concept location are grep and dependency search [5, 15]. Once the location of the change is found, *impact analysis* determines the strategy and the extent of the change [5, 16].

*Actualization* implements the new functionality and incorporates it into the old code. Actualization may also require change propagation that makes secondary changes in the old code [5].

Refactoring changes the structure of software without changing the functionality. During the typical software change, refactoring can be done as two different phases: When it is done before the actualization, it is called

*prefactoring*. Prefactoring prepares the old code for the actualization and gives it a structure that will make the actualization easier. For example, it gathers all the bits and pieces of the functionality that is going to change and makes the actualization localized, so that it affects fewer software modules; this makes the actualization simpler and easier. The other refactoring phase is called *postfactoring* and it is a clean-up after the actualization [9, 13].

*Verification* aims to guarantee the quality of the work, and it interleaves with phases of prefactoring, actualization, postfactoring, and conclusion. Although no amount of verification can give a 100% guarantee of software correctness, numerous bugs and problems can be identified and removed through systematic verification.

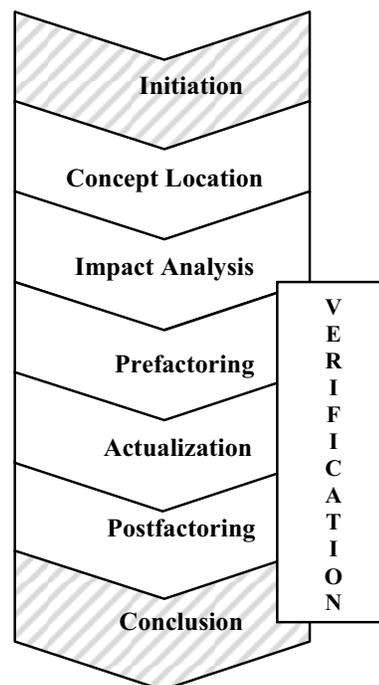


Figure 1. Phased model of software change [5]

*Conclusion* is the last phase of software change; after the new code is completed and verified, the programmers commit it into the version control system repository. This can be an opportunity to create new baseline, update documentation, prepare new release, and so forth.

## III. EXAMPLES OF PROJECTS, PROCESSES, AND PRACTICES

Table I contains a brief survey of three academic projects: The last column contains the summary of a project at Wayne State University, while the first and second columns contain a summary of two previously observed academic projects [3]. All these projects are in evolutionary stage.

### A. Academic Projects

Projects *feniCS* and *Dalton* were described in [3] and they illustrate the variability of the academic processes. Both

are large-scale distributed projects where several independent research groups cooperate.

In contrast, project BrainSpace is a project of a single research group and is described in the last column of Table I. It supports “visualization and analysis of digital representations of heterogeneous real world”. Like the other projects in the table, it is in the evolution stage. It has approximately 200 kLOC of code, and employs mostly graduate students as developers, supervised by a professor, and funded through NSF and NIH. When students graduate with their M.S. or Ph.D. degree and leave the project, their code is further evolved by new students who join the team.

For these new students, it is essential to receive a good training in the application domain and to learn the tacit knowledge of the project. It takes them about one year before their knowledge reaches the level when they can contribute to the project in a minor way, servicing the code written by previous students. Software engineering practices that they have to master are the practices of software evolution; these practices allow them to start their work on this system even when they are not familiar with the details.

It takes students typically another year before they are able to fully participate in evolution of the new code; the knowledge of geometry and visualization and the knowledge of the current system are the prerequisites for that.

#### B. Academic Organizational Practices

Table I separates the practices into organizational and code development practices. The organizational practices were discussed extensively in the literature. Table I contains “Organization” section that concentrates only on the most characteristic practices.

*Roles* in the software development processes categorize different project stakeholders and they are to a large degree dictated by circumstances of these projects. They in turn affect the *coordination and monitoring*, summarized in second row.

All three academic projects reviewed in Table I emphasize *domain knowledge* as an essential tool for the developers on the team. In case of BrainSpace, the student developers are divided into two roles: beginners who have one year of schooling, and advanced developers who typically need another year in order to contribute on advanced level.

#### C. Academic Code Development Practices

Requirements defined by stakeholders are common in commercial software and are for example assumed in Scrum process [6]. In contrast, many academic projects involve exploratory programming, where software change is initialized with only very sketchy specifications and developers have a great freedom in their implementation. This practice requires that the developers have an intimate knowledge of the domain so that they can make important decisions as they go. It also requires them to adjust the features, based on the feedback from other stakeholders. All three projects in Table I display this characteristic.

Code ownership is a practice where the code of the project is divided among the owners. Each owner knows

his/her part of the code and is responsible for it. The code owner is expected to locate the concepts within his/her part of the code. Two of the projects in Table I use this practice to locate concepts; the concept location practices of the feniCS project have not been determined.

Impact analysis in BrainSpace project is done by the supervisor or the project leader, who assess the extent of the change before it is assigned to one of the developers. For the remaining two projects, the specific practice of impact analysis is not recorded. All projects in Table I do ad hoc actualization, and refactoring is done rarely.

The verification practices done in the projects include inspections (feniCS and BrainSpace) and regression tests (feniCS and Dalton). BrainSpace conducts functional tests and efficiency tests that are important for fine tuning of some of the algorithms.

BrainSpace uses irregular iterations that are driven by funding and academic schedule, and releases are planned in approximately 6 months intervals. Dalton and feniCS do irregular and rare releases.

The projects face difficulties listed in the last row of Table I. While Dalton and feniCS list the difficulties related to coordination of distributed teams, BrainSpace lists a difficulty in management of variants that are produced for the users with diverse needs.

#### D. Recommendations

The reported observations and the generally accepted software engineering knowledge lead to the following recommendations:

All three projects practice code ownership for concept location, but this practice can be used only by experienced developers. More advanced concept location techniques would shorten start-up time for the newcomers. Improved impact analysis techniques would make software changes more predictable and as a result, the management and coordination of the projects would become easier. Increased use of prefactoring would make software changes easier, and increased postfactoring would make code more logical and slow down its decay. Currently only BrainSpace uses some refactoring, but not on sufficient scale.

## IV. SUMMARY AND FUTURE WORK

This paper summarized practices that were observed in selected academic projects and classified them according to the stage they belong to and the purpose they serve. Two of the projects are large and distributed, and one is developed by a professor and his research group.

As the future work, we want to further study the projects that involve a small academic research group, because this is a very common type of the project in academia. Very often, these projects are headed by an expert in a field other than software engineering, who is not familiar with the results of the new software engineering research. We want to develop a comprehensive set of recommended practices that will make these projects more productive and successful.

The recommendations in section III.D require further empirical validation. We are planning to conduct additional empirical work that would refine these recommendations.

## REFERENCES

- [1] S. M. Baxter, S. W. Day, J. S. Fetrow, and S. J. Reisinger, "Scientific software development is not an oxymoron," *PLoS Computational Biology*, vol. 2, p. e87, 2006.
- [2] L. Hochstein and V. R. Basili, "The ASC-Alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, pp. 50-58, 2008.
- [3] M. T. Sletholt, D. Pfahl, J. Hannay, and H. P. Langtangen, "What do we know about agile practices in scientific software development," *Computing in Science and Engineering*, 2011.
- [4] W. A. Wood and W. L. Kleb, "Exploring XP for scientific research," *Software, IEEE*, vol. 20, pp. 30-36, 2003.
- [5] V. Rajlich, *Software Engineering: The Current Practice*. Boca Raton, FL: CRC Press, 2012.
- [6] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall, 2002.
- [7] L. Rising and N. S. Janoff, "The Scrum software development process for small teams," *Software, IEEE*, vol. 17, pp. 26-32, 2000.
- [8] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," *IEEE Software*, vol. 21, pp. 62-69, July-August 2004.
- [9] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," presented at the 20th IEEE International Conference on Software Maintenance, 2004.
- [10] N. Febraro, Rajlich, V., "The Role of Incremental Change in Agile Software Processes," in *Agile 2007*, pp. 92-103.
- [11] C. Dorman and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report," in *Agile*, 2012, pp. 22-30.
- [12] K. Beck, *Test driven development: By example*: Addison-Wesley Professional, 2003.
- [13] M. Feathers, "Working Effectively with Legacy Code," ed. Upper Saddle River, NJ: Prentice Hall PTR, 2005.
- [14] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanik, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, pp. 53-95, 2013.
- [15] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," in *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00)*, Limerick, Ireland, 2000, pp. 241-249.
- [16] S. Bohner and R. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

TABLE I SUMMARY OF THREE ACADEMIC PROJECTS

	feniCS	Dalton	BrainSpace
web site	<a href="http://fenicsproject.org/">http://fenicsproject.org/</a>	<a href="http://daltonprogram.org/">http://daltonprogram.org/</a>	<a href="http://www.cs.wayne.edu/~jinhua/NSF/ImagingInformatics.htm">http://www.cs.wayne.edu/~jinhua/NSF/ImagingInformatics.htm</a>
# contributors	7 institutions + open source participants	more than 60, but few people involved in any particular time	~10 contributors in the history of the project
languages	C++, Python	Fortran 77/90, C, C++	C++
availability	open source	free, licensed	available to collaborating users
domain	differential equations	molecular chemistry	visual analytics
<b>Organization</b>			
roles	equal peers, core team, participants choose how much effort, dedicated tester	board, lab supervisors, students	supervisor (varying availability), project leader, advanced students (research assistants 50% involvement), beginners
coordination, monitoring	LaunchPad tool, distributed team	ad hoc, occasional meetings, distributed team	weekly meeting during academic year, monitoring by project leader
domain knowledge	developers are domain experts	developers are domain experts, users also provide expertise	developers are domain experts, users (physicians) also provide expertise
<b>Code development</b>			
initiation	exploratory specifications, personal initiative	exploratory specifications, participating lab priorities	both defined and exploratory, selection based on funding and stakeholder needs
concept location		solved by code ownership	solved by code ownership
impact analysis			estimated by supervisor and project leader
actualization	undisclosed	ad hoc	ad hoc
refactoring	yes	no	very few
verification	inspections, regression tests	regression tests	functional tests, efficiency tests, inspections
conclusion	Buildbot used for build, test, and release	four releases since 1997	irregular iterations, based on funding and academic schedule, releases about 6 months
<b>Open problems</b>	coordination of different requirements related to the same functionality	hard to integrate branches, errors in the underlying theory	management of variants of the main software

# WSDARWIN: A Decision-Support Tool for Web-Service Evolution

Marios Fokaefs and Eleni Stroulia  
 Department of Computing Science  
 University of Alberta  
 Edmonton, AB, Canada  
 Email: {fokaefs, stroulia}@ualberta.ca

**Abstract**—Service-oriented systems are fundamentally distributed in nature, relying on external services accessible through their public interfaces. Distributed ownership and lack of implementation transparency imply special challenges in the evolution of such systems. In order to alleviate the challenge faced by the consumers of their services, providers should, in principle, take into account the impact that service changes may have on the client applications, in addition to considering the potential benefits to be gained from the evolution of these services. In this paper, we present a decision tree to support the provider's service-evolution decision-making process. Using game theory, we construct the tree that makes explicit the value-cost trade-offs involved in considering the potential evolution of services.

## I. INTRODUCTION

The service-oriented system paradigm and associated technologies were conceived to support the reuse of functionality in the context of the development of large-scale distributed systems. Since services are consumed through standard public specifications (primarily in XML), service systems are technology agnostic and do not require any knowledge about implementation details of the services they rely upon. Although these properties have led to the easy development of component-based applications, they have also given rise to new challenges. One of them concerns the evolution of such systems. Due to the lack of implementation information between the two parties, i.e., the provider and the client, the latter is not in a position to reason about the changes of the service or the motivation behind the change. This can potentially increase the cost of adaptation for client applications and can potentially motivate the client to abandon the current provider.

When changing a service, a provider aims at maximizing some anticipated benefits, originating from the improvement of the service functionality or the extension of its features in order to acquire more revenue from clients, while, at the same time, minimizing the costs associated with actually implementing the change. These benefits and costs are direct and visible to the provider. On the other hand, there are indirect benefits and costs (also known in economic theory as *externalities* [1]) for the provider stemming from the client's reaction to the change. If the change improves the service and its quality, and meets better (or more of) the clients' needs, then they may become more committed to the service and they may be willing to pay more for it, thus generating additional revenue for the provider. On the other hand, if the clients' cost to adapt to

the provider's evolved service far exceeds the benefits from the new version or there are other alternatives in the market that better suit their requirements, they may decide to switch to a competitor service, depriving the current provider from a source of income. The implication is that service providers, while being in principle self-interested, should also consider their clientèle, since they would risk losing clients and/or failing to attract new ones. When calculating the implications of a particular service change, the provider must also consider the properties of the ecosystem (providers-services-clients) and work towards the mutual benefit of all involved parties.

In this work, we extend our previous work [2], where we proposed a provider-client game to analyse their interactions during service evolution. In this paper, we propose a decision tree as a decision support tool, based on the provider-client game. We, first, revise the game and outline how it is changed from the previous version to allow for a more realistic ecosystem with potentially many providers and many clients. We also revise the best-response analysis to define the conditions, under which certain decisions are made and which are used as the decision nodes in the proposed tree. The resulting decision tree not only allows providers to make optimal decisions, but also to understand why these decisions are optimal for them and for the ecosystem as a whole. The tool is currently being built as part of WSDARWIN, our web-service evolution platform [3].

The rest of the paper is organized as follows. Section II describes the proposed framework. In Section III, we provide an overview of this work's background by presenting a selection of related papers. Finally, Section IV concludes the paper.

## II. THE DECISION SUPPORT FRAMEWORK

### A. The Provider-Client Game

In our previous work [2], we argued that the provider-client relationship involves conflicting interests and contradicting goals. For this reason, we proposed a provider-client game to capture this relationship. Table I shows the variables used in the game and their description. We denote provider specific variables with the superscript  $P$  and client-specific variables with the superscript  $C$ . We denote the current provider with the subscript  $i$  and the competitor with the subscript  $j$ . The differential values with subscript  $ei$  refer to the different value/price of the old version of the current service and the

new version. The differential values with subscript  $ej$  refer to the different value/price of the current service (old or new) and the competitive service.

TABLE I: The variables and their definitions as used in the provider-client game.

Variables	Definition
$V_{ei}^C$	the differential value of the service of the current provider $i$
$p_{oi}$	the original price the client pays for the service before the evolution
$C_e$	the cost of the evolution process
$C_{ai j}$	the cost of adaptation to the new version of the current provider's service $i$ or the competitive service $j$ for the client application
$p_{ei}^{E S}$	the price differential for the updated software when the provider just evolves ( $E$ ) or when the provider also supports the client's adaptation ( $S$ )
$p_{ej}$	the price differential between the current provider and the competitor
$V_j^C$	the value of the competitor provider $j$ 's software for the client
$a$	the subsidy rate, which determines what portion of the adaptation costs the provider will cover
$b$	the final portion of the adaptation costs that remains for the client after the provider's support

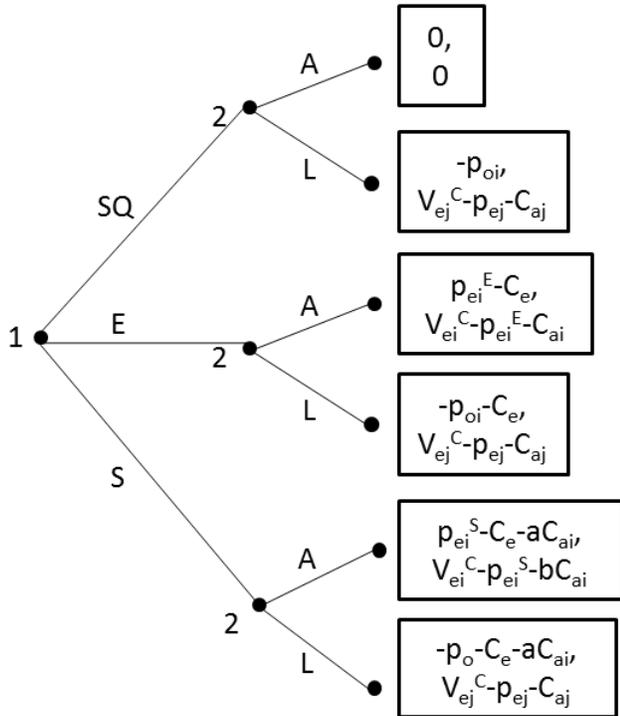


Fig. 1: The provider-client game in extensive form with the payoffs as the values for the leaves.

Figure 1 shows the provider-client game in its extensive form. The two values in the leaves of the tree correspond to the payoffs for the provider (first value) and the client (second value). In the game, the provider plays first and the possible

actions are to:

- 1) **Maintain the status quo (SQ)** of the service and make no change. In this case, there is no difference in the provider's payoff, if the client decides to stay, but the provider loses the original income  $p_{oi}$ , if the client decides to leave.
- 2) **Evolve the service (E)**, gain the increased revenues  $p_{ei}^E$ , if the client stays or lose the original income  $p_{oi}$ , if the client leaves and incur the evolution costs  $C_e$ .
- 3) **Support the client's adaptation (S)** and evolve the service. In this case, the provider gains the increased revenues  $p_{ei}^S$ , if the client stays or loses the original income  $p_{oi}$ , if the client leaves and incur the evolution  $C_e$  and part of the adaptation costs  $aC_{ai}$ .

Then the client responds and the possible actions are to:

- 1) **Adapt to the change (A)**, gain the differential of the value of the service from the old version to the new  $V_{ei}^C$  (or zero if the provider doesn't evolve the service) and incur the price differential (zero, if the provider doesn't evolve the service,  $p_{ei}^E$ , if the provider evolves or  $p_{ei}^S$ , if the provider evolves and supports) and any adaptation costs (zero, if the provider doesn't evolve the service,  $C_{ai}$ , if the provider evolves or  $bC_{ai}$ , if the provider evolves and supports, where  $C_{ai} > bC_{ai}$ ).
- 2) **Leave the current provider (L)**, gain the differential of the value of service between the current provider and the competitor  $V_{ej}^C$  and incur the price differential  $p_{ej}$  and any adaptation costs  $C_{aj}$ .

The reason why we permit only two alternatives for the client is due to the nature of service systems. Unlike other modular software, for which clients can have a local copy of the software module and invoke on demand, in service systems, the web service has to be always online, since it is accessible over a network. Any change in the service may disrupt the proper function of the clients. Maintaining multiple versions of the service at the same time may prove very costly for the provider. In this case, the provider has two options. The first is to give to the clients a grace period before making the changes to the service, so that they have time to react properly to the change. An example of this case is Twitter <sup>1</sup>, which released API v1.1 in September 2012 as a replacement for the old v1, which was completely retired in March 2013. At the end of the grace period, the clients will still have to decide whether they will adapt to the new version or switch providers. The second option is that the provider creates adapters that have the old version's interface but invoke the new version of the service. This idea was proposed and investigated by Kaminski et al. [4] and Fokaefs and Stroulia [3]. This option is in fact modelled by the support action of the provider in the proposed game.

A first difference between this game and its previous version [2] is that we no longer consider a value of the service for the provider and the income is only represented by the price.

<sup>1</sup><https://dev.twitter.com/blog/planning-for-api-v1-retirement> (last accessed 17 June 2013)

Another very important difference is that the client can now leave the current provider even if the latter retains the status quo. Assuming rationality from the client's part, if no provider evolves their services, then the client has no reason to switch providers. However, if one of them evolves the service, the client may opt to leave the current provider, if a competitor offers a better service. For simplicity purposes, we do not include the competitors as strategic players in the game, but we rather include the effect of their decisions in the calculations of the payoffs of the client.

Our analysis can be easily extended for multiple clients. We can safely argue that a client's decision does not depend on the other clients' decisions. Therefore, the outcomes and the payoffs for all players depend only on the provider's decisions and the competition. Therefore, we can run an instance of the game for each client and then aggregate the results and make the decisions that satisfies as many clients as possible.

### B. Solution Concepts and the Decision Tree

Having defined the interactions between the provider and the client, we can now proceed to analyse the game using the backward induction algorithm and the method of best-response analysis, i.e., what action a player prefers given the preferences of the other players. Backward induction is a solution algorithm for sequential games, which first calculates the optimal decision for the player that plays last for each of the previous player's actions. Then these optimal actions are taken as a given for the previous player. The process is continued until we reach the player that plays first. The final path gives us the equilibrium of the game. The best-response analysis will give us the conditions under which a player has certain preferences and which we will use to construct the decision nodes of the decision tree.

Table II presents the best-response analysis for the provider-client game. As we can see from the table, conditions 2, 4 and 6 (i.e., when it's more preferable for the client to leave the current provider) will never hold. This means that if the client leaves (i.e., conditions 7 and 8 do not hold), the provider will opt to retain the status quo of the service since this is the action that minimizes the losses for the provider. In an expanded ecosystem with many clients, the provider may balance the losses from clients that leave by attracting potentially new clients.

Figure 2 shows the final decision tree for the provider when considering the evolution of a service. The decision nodes are shown as rectangles with the number of the corresponding conditions. For each of the decisions we have two edges; one when the conditions is True in the left and another when the condition is False in the right. The end nodes of the tree representing the outcome of the decision process are shown as solid triangles with the final action for the provider.

As discussed above, all the paths for which the client leaves (i.e.,  $1(T) \rightarrow 5(T) \rightarrow 3(T) \rightarrow 8(F)$ ,  $1(T) \rightarrow 5(F) \rightarrow 7(F)$  and  $1(F) \rightarrow 5(T) \rightarrow 3(T) \rightarrow 8(F)$ ) lead the provider to retain the status quo of the service. The provider is led to the same outcome in two more paths ( $1(F) \rightarrow 5(F)$  and  $1(F) \rightarrow$

TABLE II: Best-response analysis for the provider-client game.

		When the client prefers...	
		$A \succ L$	$L \succ A$
Provider	$E \succ SQ$	1. $p_{ei}^E > C_e$	2. $-C_e > 0$
	$S \succ SQ$	3. $p_{ei}^S > C_e + aC_{ai}$	4. $-C_e - aC_{ai} > 0$
	$S \succ E$	5. $aC_{ai} < p_{ei}^S - p_{ei}^E$	6. $-aC_{ai} > 0$
		When the provider prefers...	
		$E \succ S$	$S \succ E$
Client	$A \succ L$	7. $V_{ej}^C - V_{ej}^C > p_{ej}^E - p_{ej}^S + C_{aj}$	8. $V_{ei}^C - V_{ej}^C > p_{ei}^S - p_{ej}^S + bC_{ai} - C_{aj}$

$5(T) \rightarrow 3(F)$ , in which, independently of the client's decision, it is not in the interest of the provider to evolve since the costs exceed any potential income. For the other outcomes, one path leads to the evolution of the service ( $1(T) \rightarrow 5(F) \rightarrow 7(T)$ ). The subpath  $5(T) \rightarrow 3(T) \rightarrow 8(T)$  leads to the evolution of the service with support to the client regardless of whether condition 1 holds or not. In the leftmost subtree under the root, condition 3 can never be false, since, because conditions 1 and 5 are true, we have that  $S \succ E \succ SQ$ . Furthermore, under the same subtree, when condition 5 is false, we have that  $E \succ SQ$  and  $E \succ S$ , which means that action  $E$  is a dominant strategy for the provider and we don't have to check condition 3.

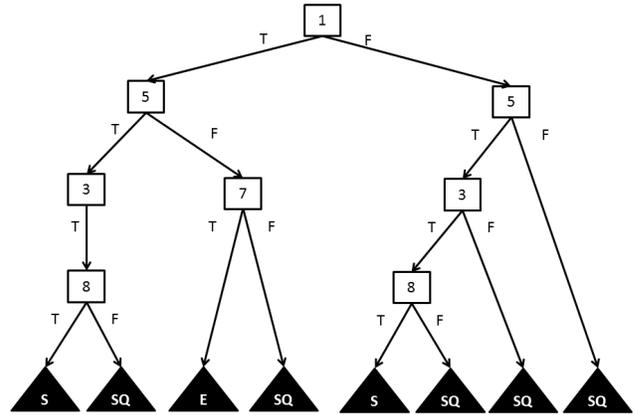


Fig. 2: The evolution decision tree for the service provider.

This decision tree is a simple tool that requires a considerable amount of information to be useful. However, this information is reasonably easy to obtain and in fact good enough estimates can be produced for all the variables required. First of all, there are variables that can be manipulated to guide the provider's decision towards a desired outcome. The provider can increase the value of the service at will and offer it at a competitive price, thus effectively overcoming competition. Furthermore, the provider can decide on the level of support

to the client to ease the adaptation process giving the client the incentive to stay. We can estimate the rest of the variables either by using formal models (i.e., for costs) or by surveying the environment of the service (i.e., for values and prices of competitive services).

### III. SOFTWARE ENGINEERING ECONOMICS BACKGROUND

Although software economics is a relatively mature field, analysing the cost and value of a particular software-engineering activity is an ever challenging problem. This problem is exacerbated in the case of service systems, because of the peculiarities of such systems, some of which we have highlighted in this work. In their work, Boehm and Sullivan [6], [7] outline these challenges and also how software-economics principles can be applied to improve software design, development and evolution.

Boehm and Sullivan define software engineering fundamentally as an activity of making decisions over time with limited resources, and usually in the face of significant uncertainties. *Uncertainties* pose a crucial challenge in software development that can lead to failure of systems. Uncertainties can arise from inaccurate estimation. For example, cost-estimation techniques and models that used to work for traditional development processes may not apply directly to modern architecture styles and development processes, such as web services. Furthermore, due to lack (or inadequacy) of economic and business information software projects may be at risk. They recognize the need of including the *value added* from any design or evolution decision. However, as they point out usually there are no explicit links between technical issues and value creation. It is critical to understand that the value added by evolving a system does not only depend on technical success but also on market conditions. It is stressed that the cost should not be judged in isolation. As Parnas suggests “for a system to create value, the cost of an increment should be proportional to the benefits delivered” [8]. Finally, the authors claim that there is a need for not only better cost estimation models but also stronger techniques for analysing benefits.

The provider-client game as presented in this work is a clear example of an ecosystem where externalities exist. An externality is an indirect cost or benefit of consumption or production activity, in other words, effects on agents other than the originator of such activity which do not work through the price system [1]. External effects such as these can lead to suboptimal, or inefficient outcomes, for the system as a whole, whereby both parties by acting independently end up less well off than they could do if they coordinated their actions or if the decision maker (in this case the provider) took into account the external effects of any action.

The Coase theorem [5] argues that an efficient outcome can be achieved through negotiations and further payments between the involved parties under certain conditions (the parties act rationally, transactions costs are minimal, and property rights are well-defined). In this work, the relationship between the provider and the client as we have described it through the game is an example of the Coase theorem.

### IV. CONCLUSION AND FUTURE PLANS

We argue that evolution in service-oriented architectures is a complicated task due to the complex dependencies between the participants of the service ecosystem. These dependencies are not only of technical nature but they also have economic and business implications. As a result strategic decisions concerning the evolution of a service should consider both technical and business dimensions of the ecosystem. In this paper, we showed how Game Theory can be used to model these dependencies and interactions between a service provider and a service client. We also proposed a decision tree as a tool to support the provider’s decision-making process. This tool is based on estimates of economic parameters such as values, costs and prices and takes into account the clients’ reactions to providers’ decisions while at the same time considering the competition.

Although the model presented in this paper focuses on a single provider and a single client, it can be easily extended for more complicated ecosystems, while the general idea remains the same; service evolution in the presence of externalities. To this end, we plan to create decision trees for providers in different market settings: few clients with many providers (*oligopsony*), few providers with many clients (*oligopoly*), or many providers with many clients (*free/competitive market*). Another important part of our future plans is the validation of the decision support system. This can be achieved by simulating various evolution scenarios and testing our decision tree with (pseudo)randomly generated values. An alternative is an on-site evaluation of the decision tree by real service providers when considering the evolution of their service.

### REFERENCES

- [1] J. Laffont, “externalities,” in *The New Palgrave Dictionary of Economics*, S. N. Durlauf and L. E. Blume, Eds. Basingstoke: Palgrave Macmillan, 2008.
- [2] M. Fokaefs, E. Stroulia, and P. R. Messinger, “Software Evolution in the Presence of Externalities: A Game-Theoretic Approach,” in *Economics-Driven Software Architecture*, I. Mistrik, R. Bahsoon, R. Kazman, K. Sullivan, and Y. Zhang, Eds. Elsevier, 2013.
- [3] M. Fokaefs and E. Stroulia, “Wsdarwin: Automatic web service client adaptation,” in *CASCON '12*, 2012.
- [4] P. Kaminski, M. Litoiu, and H. Müller, “A design technique for evolving web services,” in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06*. New York, New York, USA: ACM Press, Oct. 2006, p. 23.
- [5] R. H. Coase, “The Problem of Social Cost,” *Journal of Law and Economics*, vol. 3, pp. 1–44, 1960.
- [6] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- [7] B. Boehm and K. Sullivan, “Software economics: status and prospects,” *Information and Software Technology*, vol. 41, no. 14, pp. 937–946, 1999.
- [8] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

# A Study on Developers' Perceptions About Exception Handling Bugs

Felipe Ebert and Fernando Castor  
 Informatics Center  
 Universidade Federal de Pernambuco  
 Recife, PE, Brazil  
 {fe, castor}@cin.ufpe.br

**Abstract**—Several studies argue that exception handling code usually has poor quality and that it is commonly neglected by developers. Moreover, it is said to be the least understood, documented, and tested part of the implementation of a system. However, there are very few studies that attempt to understand developers' perceptions about exception handling, in general, and exception handling bugs, in particular. In this paper, we present the results of a survey conducted with 154 developers that aims to fill in this gap. According to the respondents of the survey, exception handling code is in fact documented and tested infrequently. Also, many of the respondents have had to fix exception handling bugs, in particular those caused by empty catch blocks or exceptions caught unintentionally. The respondents believe that exception handling bugs are more easily fixed than other kinds of bugs. Also, we found out a significant difference in the opinion of the respondents pertaining to the quality of the exception handling code: more experienced developers tend to believe that it is worse. We present a comprehensive classification of exception handling bugs based on the study results.

**Index Terms**—exception handling; bugs; survey.

## I. INTRODUCTION

Several modern object-oriented programming languages implement exception handling. Nevertheless, developers tend to focus on the normal behavior of the applications and deal with error handling only during the system implementation, in an ad hoc manner. This practice creates a proper situation for the appearance of design faults (bugs) related to exception handling. Several studies [1], [2], [3] argue that exception handling code usually has poor quality and that it is commonly neglected by developers.

There is no study that attempts to understand developers' perceptions about exception handling bugs. In this paper, we present the results from a survey we conducted asking 154 developers about their practices and perceptions regarding exception handling bugs. To the best of our knowledge, this is the largest study to date that aims to uncover what developers think about exception handling.

Respondents of our survey believe that exception handling bugs are more easily fixed than other kinds of bugs and also that exception handling code is in fact documented and tested infrequently. Also, most responses have claimed that their use of exception handling stems mainly from an interest in improving code quality. Nevertheless, according to them, organizations seldom have policies regarding the documentation and testing of exception handling code.

There is also a significant difference in the opinion of the respondents pertaining to the quality of the exception handling code: more experienced developers tend to believe that it is worse. Furthermore, most of them have at some point fixed exception handling-related bugs. Common causes include empty `catch` blocks and exceptions caught unintentionally. Finally, based on the data we collected, we present a proposal of classification of exception handling bugs.

## II. METHODOLOGY

In this section we describe the methodology of this study. It aims to answer four research questions:

- Do organizations and developers pay attention to exception handling?
- How commonplace are exception handling bugs?
- Are exception handling bugs harder to fix than other bugs?
- What are the main causes of exception handling bugs?

**What is an exception handling bug?** There is no widely accepted definition of exception handling bug. Our goal is to study bugs where exception handling is associated with the cause of the problem. Bugs in exception definition, exception throwing, exception handling, exception propagation, exception documentation, and clean-up actions (`finally` blocks) are all of interest. Therefore, an exception that is not thrown when it should be according to the expected behavior of the system is an exception handling bug. The same applies to a `catch` block that captures exceptions that it should not.

On the other hand, if a program performs a division by zero, that error will be trapped by the runtime system of the language (in the case of Java and C#, among others) and an exception will be thrown. The raising of the exception in this case is not the cause of the problem. From a bug fixing perspective, as soon as the bug is fixed, the part of the code where the bug manifested will not necessarily have a relationship with exception handling anymore. Therefore, we do not consider this to be an exception handling bug.

**Survey.** The survey was designed according to the recommendations of [4], [5] and our target population consists of developers with some experience in Java. After defining all

the questions of the questionnaire, we obtained feedback iteratively and clarified and rephrased questions and explanations. We had particular care in explaining what we meant by an exception handling bug. Together with the instructions of the questionnaire, we included some simple examples in an attempt to clarify our intent. Table I presents a summary of the questions of the survey. The complete list of questions as well as all the responses to the survey are available at the companion Web site<sup>1</sup>.

The survey was sent to two distinct groups of people. The first one, with 96 responses, consisted mostly of Portuguese speaking developers in Brazilian companies and universities. The second group, with 58 responses, consisted of bug reporters and developers of Eclipse and Tomcat. We sent more than 4000 emails to reporters of bugs in the repositories of these two systems. In total we obtained 154 responses during a period of 2 months.

### III. WHAT DEVELOPERS SAY ABOUT EXCEPTION HANDLING BUGS

The main goal of the survey is to understand developers' perceptions about exception handling bugs. It is known that non-trivial systems usually have bugs that are difficult to find, stemming from complex control flow and overly general `catch` blocks [6] and from I/O operations [7]. However, even though the perceptions of developers about exception handling in general have been studied, at least on a small scale [3], to the best of our knowledge there are no studies that attempt to understand how developers regard exception handling bugs.

Respondents had on average between 7 and 10 years of software development experience (question 1). Most of them are currently working on medium-sized projects ranging between 50 and 100KLoC (question 2). Most respondents have developed systems using at least three different programming languages (question 3). In the remainder of this section we discuss the main findings of the survey based on the four research questions presented in Section II.

**Do organizations and developers pay attention to exception handling?** The survey included five questions (4, 5, 9, 18 and 19) whose goal is to determine whether developers worry about exception handling when they are not directly implementing the system, e.g., designing, testing, etc. For question 5, only 27% of the developers answered "yes". In a similar vein, 30% said that there are specific tests for exception handling code in their organization (question 9). And for question 4, 61% of the respondents said that none to little importance is given to the documentation of exception handling. Moreover, just 16% said that much or very much importance is given to exception handling documentation.

For question 18, about 40% of the respondents consider that the quality of exception handling code ranges between good and very good. Surprisingly, only 14% of the respondents consider that it is bad or very bad. This result correlates with developer experience. We found that the more experienced a

respondent, the greater his/her tendency to consider the quality of exception handling code to be bad with the Student's T-test producing a p-value of 0.02478.

Inspired by the work of Shah et al. [3], we asked question 19. We provided them with a list of possible causes and gave them the opportunity to suggest additional ones. Table II presents the reasons more frequently cited by the respondents. Most of the respondents said that creating ways to tolerate faults and improving the quality of a functionality are the main reasons to use exception handling. One of the survey respondents provided a particularly interesting spontaneous answer for this question:

*"exception handling is part of code flow - not using exception handling for some arbitrary reason would be like not using 'if' blocks, or not having your code compile."*

Only 17% of the respondents said they use exception handling for debugging purposes. In contrast, in the work of Shah et al. [3], which interviewed a group of 8 novice developers and 7 experts, most of the novice developers claimed to use exception handling mostly for debugging and because of language requirements. Expert developers, on the other hand, claimed that they use exception handling mainly to convey understandable failure messages. In contrast, in our study the most experienced and the least experienced respondents coincided in terms of both the most often cited reasons and the least often cited reasons (Table II).

Based on previous work discussing the problems with checked exceptions in Java [8], we expected a larger percentage of respondents to mention "language requirement" as a reason for using exceptions. Moreover, only 21% use exception handling because of organizational policies. This result and the previously discussed ones pertaining to testing and documentation suggest that organizations do not pay attention to exception handling, even though developers do. Finally, it is interesting to note that the 3 respondents who claimed not to use exception handling have professionally worked only with languages that implement exception handling and all of them have professionally worked with Java.

Developer experience does not seem to make a difference in the main reasons for the use of exception handling. For both the least experienced (until 5 years of development experience) and the most experienced (10+ years) respondents, improving the quality of a feature and creating ways to tolerate faults are the most common reasons to use of exception handling. Also, for both groups, organization policies and the need to debug the code are the least often cited reasons.

TABLE II  
WHY DO DEVELOPERS USE EXCEPTION HANDLING

To create ways to tolerate faults	66%
To improve the quality of a functionality	63%
Importance of exception handling	53%
Language requirement	43%
Organizational policies	21%
To debug a specific part of the code	17%
Does not use exception handling	2%

**How commonplace are exception handling bugs?** To assess

<sup>1</sup>Address: <https://sites.google.com/a/cin.ufpe.br/eh-bugs/>

TABLE I  
SUMMARY OF SURVEY QUESTIONS

Experience	1. For how long have you been a Java developer? 2. What is the approximate size of the project you are currently working on (LoC estimate)? 3. Which programming languages have you professionally worked with?
Context	4. In the design phase of your projects, what importance is given to the documentation of exception handling?
Documentation	5. Are there any specifications, documented policies or standards that are part of your organization's culture related to the implementation of error handling? 6. How often are bugs related to exception handling reported at your organization? 7. How often are bugs reported at your organization? 8. Does your organization use any tool for reporting and keeping track of bugs?
Testing	9. Are there specific tests for the exception handling code in your organization?
Bugs	10. How often do you find bugs related to exception handling? 11. How often do you find bugs that are not related to exception handling? 12. Estimate the percentage of bugs related to exception handling code in your projects 13. Have you ever needed to fix bugs related to exception handling? 14. If you answered yes to question 13, please describe some of these situations 15. Select the main causes of bugs related to exception handling you have ever needed to fix, analyze or have found documented 16. What is the average level of difficulty to fix bugs related to exception handling? 17. What is the average level of difficulty to fix other bugs that are not related to exception handling? 18. What is your opinion about the quality of exception handling code in your projects compared to other parts of the code 19. Why do you use exception handling in your projects?

how commonplace developers believe exception handling bugs to be, we asked them questions 10 and 11. The answers were given in a scale comprising: “never”, “rarely”, “sometimes”, “most of the time” and “always”. We put the answers in a numeric scale and used Student’s T-test to check whether the answers are significantly different. According to the respondents, exception handling bugs are less frequently found than other bugs. We obtained a p-value of less than 0.0001. We also asked them to estimate the percentage of exception handling bugs in their projects (question 12). The mean estimate was 9.79% and the median was 5%.

Complementarily, we presented the respondents with the questions 6 and 7. Most of the respondents answered that exception handling bugs are reported “sometimes” (question 6) and other bugs are reported “most of the times” (question 7). The answers for the two questions differ significantly with a p-value of less than 0.0001.

**Are exception handling bugs harder to fix than other bugs?** Our study revealed that respondents consider exception handling bugs easier to correct than other types of bugs. 43% of the respondents consider exception handling bugs to be easy or very easy to fix (question 16). In sharp contrast, only 7% of the respondents say the same about other kinds of bugs (question 17). The answers for questions 16 and 17 were given in a scale comprising: “very easy”, “easy”, “medium”, “hard” and “very hard”. We put the answers in a numeric scale. Then we employed the T-test to analyze whether the answers for the difficulty of fixing exception handling and other bugs are significantly different. We found a p-value of less than 0.0001, thus, according to the respondents, exception handling bugs are easier to fix than other bugs.

**What are the main causes of exception handling bugs?** To uncover the main causes of exception handling bugs according to the survey respondents, we posed three questions (13, 14 and 15). Question 15 directly asked them about the main causes. The respondents were allowed to select zero or more causes from a list and could also suggest additional ones. The

most commonly cited cause for exception handling bugs was the “lack of a handler that should exist”. Followed by the causes “no exception thrown in a situation of a known error” and “programming error in the catch block”.

To better understand developer perceptions about the causes of exception handling bugs, we asked questions 13 and 14. 83% of the respondents have had to fix an exception handling bug at some point (question 13) and 113 out of 154 survey respondents answered the latter question (question 14). The answers varied widely and many of them refer to specific technologies, frameworks and applications. Out of the 113, 19 had to fix bugs caused by empty catch blocks and 16 have fixed bugs stemming from exceptions caught unintentionally. The final result of this question is the exception handling bug classification presented in Table III.

TABLE III  
EXCEPTION HANDLING BUGS CLASSIFICATION.

1. Lack of a handler that should exist
2. No exception thrown in a situation of a known error
3. Programming error in the catch block
4. Programming error in the finally block
5. Exception is caught unintentionally
6. Catch block where only a finally would be appropriate
7. Exception that should not have been thrown
8. Wrong encapsulation of exception cause
9. Wrong exception thrown
10. Lack of a finally block that should exist
11. Error in the exception assertion

There are diametrically different opinions on the subject of empty catch blocks. Even though a number of respondents consider them to be sources of bugs, some survey respondents seem to disagree:

*“I’m also an Eclipse committer (on Platform/UI). On 4.2 we’ve changed how parts (e.g., editors and views) are rendered. Our new system silently swallows otherwise-uncaught exceptions. Tracing what happens when an EditorPart or ViewPart throw an uncaught exception is a teensy bit annoying.”*

The citation above shows that the respondent does not want to know about problems in some parts of the system and

swallows exceptions to achieve that goal.

#### IV. THREATS TO VALIDITY

**Internal Validity.** Our survey was conducted with an online and self-administered questionnaire. It is possible that respondents may have misunderstood the definition of exception handling bug and answered the questions based on a different understanding meaning. We tried to reduce the probability of this occurring by providing a clear definition of exception handling bug and some simple examples in the survey instructions.

**External Validity.** Our survey involved 154 respondents. This number is small and limits the generalizability of the results. Nonetheless, respondents of the survey came from different professional and cultural backgrounds. Furthermore, the largest study to date on the viewpoints of developers about exception handling [3] involved only 15 respondents (they were interviewed, instead of responding to a survey). Therefore, from a comprehensiveness standpoint, we can say that our study is an improvement over the current state-of-the-art.

**Construct Validity.** Our survey might not have covered all questions that we could have been asked of the respondents. Nonetheless, the final questionnaire was the result of several discussions between the authors, one of them a specialist in exception handling, and with a number of software developers and academics. Moreover, we run at least two small pilot studies before finally making the questionnaire public.

#### V. RELATED WORK

The study most strongly related to ours was conducted by Shah et al. [3]. It involved 8 novice and 7 expert software developers and had the goal of understanding their viewpoints on exception handling. The authors conducted semi-structured interviews with developers and the results show that novice developers neglect exception handling until there is an error or until they are forced to address the problem due to language requirements. Furthermore, they do not like being forced by the language to use exception handling constructs. In contrast, the experienced developers think that exception handling is a very important part of development.

Marinescu [9] and Sawadpong et al. [10] analyzed bug reports from Eclipse. Marinescu [9] analyzed the defect-proneness of classes that use exception handling. Her study revealed that classes that throw or handle exceptions are indeed more defect-prone than others classes. Sawadpong et al. [10] aimed to determine if the usage of exception handling is relatively risky by analyzing the defect densities of exception handling code and the overall source code. They discovered that exception handling defect density of exception handling constructs is approximately three times higher than overall defect density.

None of the two aforementioned studies have also accounted for the perceptions of developers about exception handling bugs. Furthermore, none of them analyzes issues such as whether exception handling bugs are easier to fix nor what are their causes.

#### VI. CONCLUSIONS AND FUTURE WORK

This paper presented a study on exception handling bugs based on a survey with software developers and researchers. The results show that exception handling code is not documented and tested frequently and also that developers assume that fixing exception handling bugs is easier than fixing other bugs. Also, developers claim that they use exception handling mainly to improve the quality of the systems they produce and also more experienced developers tend to believe that the quality of exception handling code is worse.

We also present a comprehensive classification of exception handling bugs based on the study results. The results of this study emphasize that the views of developers and organizations about exception handling bugs are in conflict. To improve the quality of software systems, these views must be reconciled so that exception handling code can receive more attention. The presented classification of exception handling bugs can provide assistance in that task, e.g., by working as a checklist for code inspections or a guide to the design of test cases.

In the future, we plan to conduct interviews with developers since very useful information in our study came from spontaneous answers provided by the respondents of the survey. Also, we are currently analyzing bug reports from two real systems so that we can compare what developers say and what they do in fact, when it comes to exception handling bugs.

#### VII. ACKNOWLEDGEMENTS

We would like to thank all respondents of the survey that took some precious time to answer our questionnaire. We also would like to thank the anonymous referees, who helped to improve this paper. Fernando is supported by CNPq (306619/2011-3), FACEPE (APQ-1367-1.03/12), and by INES (CNPq 573964/2008-4 and FACEPE APQ-1037- 1.03/08).

#### REFERENCES

- [1] F. Cristian, "Exception handling," in *Dependability of Resilient Computers*. Blackwell Science, 1989, pp. 68–97.
- [2] D. Reimer and H. Srinivasan, "Analysing exception usage in large java applications," in *Proceedings of ECOOP Workshop on Exception Handling in Object-Oriented Systems*, July 2003, pp. 10–19.
- [3] H. Shah, C. Gorg, and M. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 150–161, 2010.
- [4] R. M. Groves, F. J. Fowler, M. P. Couper, J. M. Lepkowski, E. Singer, and R. Tourangeau, *Survey Methodology*, 2nd ed. Wiley, 2009.
- [5] B. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds., 2008, pp. 63–92.
- [6] M. Robillard and G. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM TOSEM*, vol. 12, no. 2, pp. 191–221, April 2003.
- [7] P. Zhang and S. G. Elbaum, "Amplifying tests to validate exception handling code," in *Proc. of the 34th ICSE*, June 2012.
- [8] B. Cabral and P. Marques, "Exception handling: a field study in java and .net," in *Proc. of the 21st ECOOP*. Springer-Verlag, 2007, pp. 151–175.
- [9] C. Marinescu, "Are the classes that use exceptions defect prone?" in *Proceedings of the 12th International Workshop on Principles of Software Evolution*, September 2011, pp. 56–60.
- [10] P. Sawadpong, E. B. Allen, and B. J. Williams, "Exception handling defects: An empirical study," *9th IEEE International Symposium on High-Assurance Systems Engineering*, pp. 90–97, October 2012.

# On the Relationship between the Vocabulary of Bug Reports and Source Code

Laura Moreno, Wathsala Bandara, Sonia Haiduc, Andrian Marcus

Department of Computer Science  
Wayne State University  
Detroit, MI, USA

lmorenoc@wayne.edu, waths@wayne.edu, sonja@wayne.edu, amarcus@wayne.edu

**Abstract**—Text retrieval (TR) techniques have been widely used to support concept and bug location. When locating bugs, developers often formulate queries based on the bug descriptions. More than that, a large body of research uses bug descriptions to evaluate bug location techniques using TR. The implicit assumption is that the bug descriptions and the relevant source code files share important words. In this paper, we present an empirical study that explores this conjecture. We found that bug reports share more terms with the patched classes than with the other classes in the system. Furthermore, we found that the class names are more likely to share terms with the bug descriptions than other code locations, while more verbose parts of the code (e.g., comments) will share more words. We also found that the shared terms may be better predictors for bug location than some TR techniques.

**Keywords**—Bug location, text retrieval, source code vocabulary.

## I. INTRODUCTION

Determining where to fix a bug (i.e., *bug location*) is an instance of *concept location* in source code, and it is one of the main tasks performed by developers during software evolution. Text Retrieval (TR) based approaches have been proposed to partially automate this task [1]. These techniques rely on natural language queries often formulated—either manually or automatically—based on the descriptions in the bug reports. The use of TR techniques for bug localization is based on the assumption that a bug report and its related faulty code share important vocabulary. However, no studies have presented specific evidence to support this assumption. Furthermore, no previous work has studied where the shared terms between the two vocabularies come from in the code, although it has been shown that such information can be used to improve TR-based bug location [2].

In this paper we analyze the common vocabularies of bug reports and source code classes in a collection of software systems and some of its properties. We use this analysis to find evidence that supports the use of this common vocabulary and TR techniques for bug location and ways to improve them. In particular, we address the following research questions:

RQ1. *To what extent are the vocabularies of bug reports reflected in the identifiers and comments of classes?*

RQ2. *What is the code location (i.e., class name, method name, attribute name, etc.) of the shared terms between bug reports and patched classes?*

RQ3. *Is the number of shared terms between bug reports and classes an adequate measure to support bug location?*

Our study indicates that bug reports share more terms with the corresponding patched classes than with the other classes in a system. Moreover, these shared terms are not randomly distributed in the patched classes. Locations with more terms tend to have more of the shared terms, but class names are more likely to have shared terms than other locations. This confirms assumptions made in recent work on integrating structural information with TR for bug location [2], where code locations are considered when retrieving source code. Finally, we also found that bug location may be better supported by using the shared terms between bug descriptions and the source code, than by using some complex TR techniques.

## II. METHODOLOGY

The main goal of our study is to explore the vocabularies of bug reports and the source code.

### A. Data Collection

We used data (i.e., bug descriptions and patched classes) previously used for evaluating TR-based techniques in concept location [3, 4]. The complete data is available online at: <http://www.cs.wayne.edu/~severe/era13>. The data is from six open source Java software systems from different problem domains, which are summarized in Table I. ADempiere is an extended business solution that provides enterprise resource planning, and customer relationship management and support. Art of Illusion is a 3D modeling, animation, and rendering tool. iTunes is an audio player and manager. Eclipse is a popular integrated development environment. JEdit is a powerful text editor oriented to programmers and customizable via plugins. In this study, we considered two versions of Eclipse (i.e., 2.0 and 3.5), and a single version of the rest of the systems. In total, our data collection has 34,375 classes (see Table I).

TABLE I. SYSTEMS USED IN THE STUDY AND THEIR PROPERTIES

System	Version	# of Classes	# of Bug Reports	# of Patched Classes
ADempiere	3.10	1896	16	16
Art of Illusion	2.4.1	570	10	13
iTunes	1.10	439	17	22
Eclipse	2.0	7,689	13	14
Eclipse	3.5 <sup>a</sup>	22,980	40	74
JEdit	4.2	801	18	27
<i>Total</i>		34,375	114	166

The data set contains 114 bug reports in total and 166 classes that were patched while fixing those bugs (see Table I). Most of the bug reports in the data collection (88.77%) are

related to one or two patched classes, while the rest of the reports are associated with three to eight classes. Only one report resulted in more than ten patched classes.

### B. Corpus Creation

We represented the documents in the collection as “bags of words”, a commonly adopted approach for TR-based bug location. In the case of bug reports, we extracted the text from their title and description. In the case of classes, we extracted the text from their identifiers, comments, and other literals. For each bug report and each class we created one document, respectively. Bug reports were used as queries to retrieve classes. We normalized the text in the following manner. First, we split identifiers according to Camel case and underscore separators. Next, we filtered out common English stop words and programming keywords. Finally, we stemmed the words using the Porter stemmer.

### C. Measures

We compute the *size* of a document/query as the number of its terms, and the number of *unique terms* as the *size of its vocabulary*. We define measures to analyze the vocabulary of the documents/queries (i.e., classes/bug reports).

To answer RQ1, we extract the *shared terms* between each possible pair  $\langle \text{bug report}, \text{class} \rangle$  in a software system as the intersection of terms between their vocabularies. We compute the size of this intersection and the frequency of occurrence of its terms in the bug report and the class. For the same pairs, we measure the *Simpson similarity index* given by the number of shared terms between the bug report and the class divided by the minimum size of their vocabularies.

For addressing RQ2, we compute the size (i.e., the number of terms) of different locations in the patched classes (e.g., comments, class names, method names, etc.). We extract the set of shared terms between each location and the bug reports, and their frequency of occurrence in the location.

For answering RQ3, we compare the use of the size of the common vocabulary between bug reports and classes with two TR approaches, namely Latent Semantic Indexing (LSI) and the Lucene implementation of the Vector Space Model, both widely used for bug location. For each bug title and description (i.e., query), we rank the classes of each system based on three measures: (1) the number of shared terms between the class and the bug report; (2) the cosine similarity between the class and the query using LSI; and (3) the cosine similarity between the class and the query using Lucene. We measure and compare the effectiveness of each technique, i.e., the rank of the first patched class in the list.

## III. RESULTS AND FINDINGS

Most of the bug reports (75%) are less than or equal to 64 terms in size. The largest report in the corpus has 250 terms and belongs to Eclipse 3.5; while the shortest one has eight terms and belongs to ADempiere. The average size of the bug reports is 56 terms. Approximately two thirds of the terms (65.86%) in each bug report are unique, in average. The largest bug report also contains the largest set of unique terms (114), whereas the bug report with the smallest set of unique terms, consisting of five terms, belongs to aTunes. We verify

the correlation between the size of a code location and the size of its vocabulary. According to the Spearman coefficient, the monotonic relationship between these variables is very strong ( $r = 0.96$ ,  $p\text{-value} < 0.01$ ), which means that the larger the bug report, the more unique terms it contains.

Larger than the bug reports, 75% of the classes have more than 82 terms. In average, the size of a class is 464 terms, i.e., about 85% more than the largest bug report in the collection. Once again, the largest class in the corpus belongs to Eclipse 3.5, while the shortest ones (consisting of one term) are from Art of Illusion, Eclipse 2.0, and Eclipse 3.0. Moreover, less than one third of the terms in the classes (27.1%), in average, are unique. The Spearman correlation between the number of unique terms and the document size is also very strong in the case of classes ( $r = 0.95$ ,  $p\text{-value} < 0.01$ ), so larger classes have larger vocabularies.

### A. Common Vocabulary between Bug Reports and Classes

In order to answer RQ1 we analyzed the common terms between bug reports and classes. We computed this set for 1,077,074 pairs  $\langle \text{bug report}, \text{class} \rangle$ , corresponding to every possible combination of these elements per system. We found that 75% of the pairs (i.e., 808,928 pairs) share between one and 13 terms. 21.68% of the pairs, however, do not share any term, and only 3.22% (34,646 pairs) have more than 13 terms in common. These percentages vary little when analyzing the pairs by software system, (see Table II). We conclude that

Bug reports share terms with a large number of classes (almost 80%) in a software system.

TABLE II. PERCENTAGE OF PAIRS SHARING NO TERMS, ONE TO 13 TERMS AND MORE THAN 13 TERMS

System	Pairs $\langle \text{bug report}, \text{class} \rangle$ sharing		
	0 terms	1-13 terms	More than 13 terms
ADempiere 3.10	17.82%	81.89%	0.29%
Art of Illusion 2.4.1	19.39%	78.91%	1.7%
aTunes 1.10	31.5%	67.94%	0.56%
Eclipse 2.0	18.83%	79.09%	2.09%
Eclipse 3.5	22.03%	74.47%	3.5%
JEdit 4.2	22.92%	75.76%	1.32%
All	21.68%	75.10%	3.22%

To determine whether this observation holds for the code relevant to the bug reports, we analyzed the subset of  $\langle \text{bug report}, \text{corresponding patched class} \rangle$  pairs, which consists of 166 elements. We refer to this as the *patched subset* and to its complement as the *non-patched subset*. We found that the set of shared terms is non-empty for 99.6% of the pairs (i.e., 165 out of 166) in the patched subset and 78.32% of the pairs in the non-patched subset. There is only one patched class with no common terms with its corresponding bug report, in aTunes. This class consists of 54 unique terms, while the bug report consists of only 5 unique terms. Figure 1 reports statistics on the size of the shared vocabularies. According to the Mann-Whitney test, the difference of shared terms in these subsets is statistically significant for all the systems ( $p\text{-value} < 0.05$ ). The patched classes share, in average, 11.7 terms with their corresponding bug reports, whereas the other classes share only 3.39 terms, in average, with the bug reports. We conclude that

Bug reports have more terms in common with the patched classes than with the non-patched classes, in average.

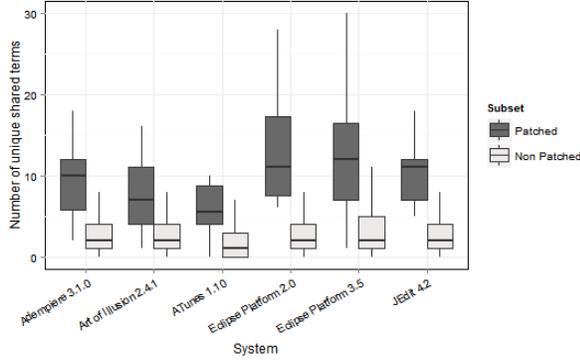


Figure 1. Number of unique shared terms between bug reports and classes in each system. The outliers have been omitted.

In order to account for any size difference between the patched and non-patched classes, we compute the Simpson similarity index. The Simpson similarity is appropriate in cases where there is a difference in the size of two documents, as it captures the overlap with respect to the document having the smallest size. Figure 2 shows boxplots corresponding to the Simpson similarities for each system. We found that the similarity between bug reports and classes are significantly higher ( $p\text{-value} < 0.05$ ) for patched classes than for the non-patched classes. In average, the similarity values in the non-patched and patched subsets are 0.11 and 0.37, respectively. Interestingly, we found that the Simpson similarity is above 0.24 in 75% of the cases in the patched subset and below the same value in 88.59% of the pairs in the non-patched subset. We confirm the previous observation and conclude that

Bug reports have higher textual similarity with the patched classes than with the non-patched classes, in average.

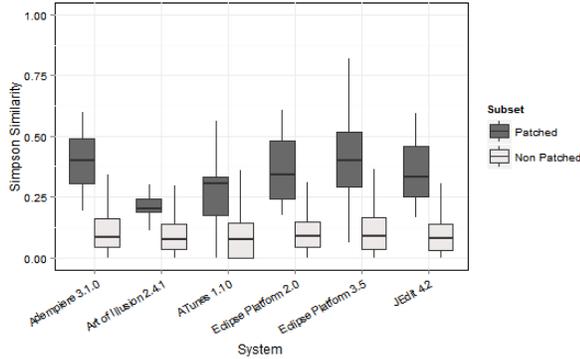


Figure 2. Simpson similarity index between bug reports and classes in each system. The outliers have been omitted.

### B. Code Locations of Common Terms

Most TR-based techniques consider the source code as bag of words. Recent work [2] argued that considering code locations when retrieving source code improves bug location. So, we studied the locations in the code where the common terms between bug reports and patched classes occur. For this, we represented the source code as srcML [5] documents. Then, we determined the terms in the locations described in Table III and recorded their frequency of occurrence. We also

identified which of them are shared with the corresponding bug reports. We found that the terms in the patched classes are not evenly distributed among the locations. The largest contributor locations to the class vocabulary are: the comments (26.98%), method calls (20.28%), and types (12.69%) - see the third column in Table III. This is somewhat expected, as these locations are more verbose or more frequent than others.

TABLE III. CODE LOCATIONS CONSIDERED IN THE STUDY. (THE VALUES REFER ONLY TO PATCHED CLASSES)

Code location	Refers to	Avg. % of terms	Avg. % of shared terms	Avg. contrib. to shared terms
Argument	Argument name	11.71%	19.06%	11.34%
Attribute	Attribute name	8.40%	20.10%	8.45%
Comment	Comment/Javadoc	26.98%	19.86%	27.33%
Literal	String literal	2.99%	21.40%	4.64%
Method call	Called method	20.28%	15.78%	16.99%
Method name	Declared method	4.40%	21.52%	4.58%
Parameter	Parameter name	4.25%	19.97%	3.81%
Type	Name of a type	12.69%	20.69%	12.97%
Class Name	Declared class	0.55%	53.57%	1.70%
Variable	Variable name	7.86%	15.68%	7.11%

When determining the contribution of each code location to the total shared terms of the classes, we found that the shared terms are primarily present in the comments (27.33%, in average), method calls (16.99%), and types (12.97%) of the patched classes. These locations are the same with the highest percentage of terms in the classes. According to the Spearman correlation, there is a strong association ( $r > 0.6$ ,  $p\text{-value} < 0.01$ ) between the number of terms and the number of shared terms in all the locations, except for class names. Therefore, we can state that

The more verbose a code location is, the more it contributes to the common vocabulary between a patched class and its corresponding bug report.

Size is not everything, though. A short location may contribute a few terms to the total of shared vocabulary, but the size of its contribution may be very large compared to its size. In other words, most of the words in a location may be part of the shared vocabulary. So, we compute the percentage of shared terms for each location, i.e., the number of common terms in the location with respect to its size (see the fourth column in Table III). We found that class names share, by far, the largest percentage of their terms (53.57%). The other code locations share between 15% and 21% of their terms with the bug reports. We conclude that

The names of the patched classes are more likely to have share terms with the corresponding bug reports than other locations.

While the granularity we used here is different than in previous work (i.e., classes vs. methods [2]), we can hypothesize that the names of the software entities that make up the documents are most important for bug location. Further work and evidence are needed to confirm this hypothesis.

### C. Common Vocabularies vs. TR techniques

We constructed a very simple technique to support bug location, where we used the number of shared terms (ST) as measure for locating the classes relevant to bug reports. We built a corpus for each system, as explained in section II.B.

We indexed each corpus with Lucene and LSI (we used  $d=100$  as parameter for LSI). For each query (i.e. bug report) from each system, we ranked the document (i.e., classes) using ST, LSI, and Lucene, respectively. Then we compared the best ranked of the patched classes (i.e., effectiveness).

The number of cases when ST performs better, equal, or worse than Lucene and LSI is reported in Table IV. The average and median effectiveness values for the three approaches in each system are reported in Table V. A lower effectiveness measure indicates better results.

TABLE IV. NUMBER OF CASES WHEN THE SHARED TERMS (ST) APPROACH IS BETTER, EQUAL OR WORSE THAN LSI AND LUCENE

System	LSI			Lucene		
	ST better	Equal	ST worse	ST better	Equal	ST worse
ADempiere	9 (56%)	0 (0%)	7 (44%)	1 (6%)	1 (6%)	14 (88%)
Art of Illusion	4 (40%)	0 (0%)	6 (60%)	0 (0%)	0 (0%)	10 (100%)
aTunes	9 (53%)	0 (0%)	8 (47%)	1 (6%)	1 (6%)	15 (88%)
Eclipse 2.0	11 (85%)	0 (0%)	2 (15%)	2 (15%)	1 (8%)	10 (77%)
Eclipse 3.5	24 (60%)	0 (0%)	16 (40%)	9 (22%)	3 (8%)	28 (70%)
JEdit	15 (83%)	0 (0%)	3 (17%)	5 (28%)	1 (6%)	12 (66%)
All	72 (63%)	0 (0%)	42 (37%)	18 (16%)	7 (6%)	89 (78%)

TABLE V. AVERAGE AND MEDIAN EFFECTIVENESS OF THE SHARED TERMS (ST), LSI AND LUCENE APPROACHES

System	ST		LSI		Lucene	
	Average	Median	Average	Median	Average	Median
ADempiere	41	19	124	23	11	4
Art of Illusion	87	35	84	30	52	10
aTunes	26	10	31	18	9	3
Eclipse 2.0	180	80	1152	681	49	3
Eclipse 3.5	430	66	915	120	594	5
JEdit	22	7	64	57	11	3
All	192	25	492	57	223	4

To our surprise, ST outperforms LSI in 63% of the cases and has a better median and average effectiveness considering data from all systems. The only system where LSI performs slightly better is Art Of Illusion, but the difference in median effectiveness is small (5 positions), compared to the improvement ST brings in the case of other systems, such as Eclipse 3.5 (601 positions). According to the Wilcoxon test, ST performs significantly better than LSI ( $p\text{-value} < 0.05$ ).

Lucene outperforms both ST and LSI. The Wilcoxon test on the effectiveness values reveals again that the difference in performance of the three approaches is statistically significant ( $p\text{-value} < 0.05$ ). However, there are cases where using ST leads to better results even than Lucene. The systems with most such cases are JEdit, where ST performed better in 28% of the cases, Eclipse 3.5, where this percentage was 22%, and Eclipse 2.0, where ST performed better than Lucene in 22% of the cases. For some of the bugs, the difference in effectiveness is striking, as in the case of bugs 304784 and 29950 in Eclipse 3.5, where the improvement over Lucene was of 7549 and 4960 positions, respectively. These outliers in Eclipse 3.5 explain also the better average effectiveness obtained by ST for this system, compared to Lucene.

Further work is needed to explain these results, which is beyond the scope of a short paper. Nevertheless, the results allow us to conclude that

The number of shared terms between bug reports and classes supports bug location better than LSI, yet not as well as Lucene.

#### IV. RELATED WORK

Related to our study is the linguistic and statistical analysis of the text found in bug reports, which includes: the analysis of the parts of speech of the words used in bug titles [6, 7]; word frequency analysis of bug report titles [7]; distribution of bug title words across bugs having different severity levels [7]; topic analysis [8]; and coherence analysis [9] of the bug reports. While our study also performs an analysis of the text found in bug reports, it does so with respect to the source code, aiming to determine the relationship between the bug reports vocabulary and the code vocabulary.

Recent work in bug localization [2] has proposed boosting the terms in the queries that appear in certain locations within code methods. In this paper we lay the foundation for extending such work to classes. In addition, we consider an extensive list of code locations that includes method and class names, and comments.

#### V. CONCLUSIONS AND FUTURE WORK

We confirmed the assumption that the source code shares many terms with bug reports and, moreover, that the number of shared terms is higher for patched classes than for the rest of the classes in a system. We also established that class names are more likely to have common vocabulary with the bug reports than other code locations. On the other hand, more verbose code locations will have more terms in common with the bug descriptions. We believe that TR-based concept location techniques may benefit from this information. Future work will verify this statement.

We found that the shared terms alone can be used to locate the classes relevant to a bug report. While some TR techniques (i.e., Lucene) will work better, other ones (i.e., LSI) may perform worse.

The obvious next step in this work is to address some of the threats to validity that affect our conclusions, especially regarding the external validity. For that, we will replicate the study on larger data sets. We will also consider additional TR techniques and various configurations.

#### REFERENCES

- [1] A. Marcus, A. Sergeev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *IEEE Working Conf. on Reverse Engineering*, 2004, pp. 214-223.
- [2] B. Bassett and N. A. Kraft, "Structural Information Based Term Weighting in Text Retrieval for Feature Location," in *IEEE Int'l. Conf. on Program Comprehension*, 2013, pp. 133-141.
- [3] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic Query Reformulations for Text Retrieval in Software Engineering," in *IEEE/ACM Int'l. Conf. on Software Engineering*, USA, 2013, pp. 842-851.
- [4] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," in *IEEE Int'l Conf. on Program Comprehension*, 2011, pp. 1-10.
- [5] M. Collard, J. I. Maletic, and A. Marcus, "Supporting Document and Data Views of Source Code," in *ACM Symposium on Document Engineering*, 2002, pp. 34-41.
- [6] A. J. Ko, B. A. Myers, and D. H. Chau, "A Linguistic Analysis of How People Describe Software Problems in Bug Reports," in *IEEE Conf. on Visual Language and Human-Centric Computing*, 2006, pp. 127-134.
- [7] A. Sureka and K. Varma Indukuri, "Linguistic Analysis of Bug Report Titles With Respect to the Dimension of Bug Importance," in *Third Annual ACM Bangalore Conference*, 2010.
- [8] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs," in *Working Conf. on Reverse Engineering*, 2012, pp. 83-92.
- [9] E. Linstead and P. Baldi, "Mining the Coherence of GNOME Bug Reports with Statistical Topic Models," in *IEEE Int'l. Working Conf. on Mining Software Repositories*, 2009, pp. 99-102.

# Database-aware Fault Localization for Dynamic Web Applications

Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen  
 Electrical and Computer Engineering Department  
 Iowa State University  
 Email: {hungnv,hoan,tung,tien}@iastate.edu

**Abstract**—Localizing and fixing software faults is an important maintenance task. In a dynamic Web application, localizing the faults is challenging due to its dynamic nature and the interactions between the application and databases. The faults could occur in the statements in the host program or inside the queries that are sent from the application to be executed in the database engines. This paper presents SQLook, a novel database-aware fault localization method that is able to locate output faults in PHP statements of a dynamic Web application as well as in SQL queries. In SQLook, a PHP interpreter is instrumented to execute an SQL query and to monitor the evaluation of those SQL predicates to determine if they affect the output process of individual data records. It performs row-based slicing across PHP statements and SQL queries to record the entities that are involved in the output of each data row. Our empirical evaluation shows that SQLook can achieve higher accuracy than the state-of-the-art database-aware fault localization approach.

**Keywords**—Fault Localization; Database-aware; Dynamic Web

## I. INTRODUCTION

A dynamic Web application is often written in a language such as PHP or ASP that communicates with the databases to retrieve data, processes and displays them on the client-side browsers. In such a program, there exist program statements that are responsible for interacting with the databases, which are called *database-interaction points*. Before an interaction point, the program constructs a string *query* from literals, variables, functions' returned values, etc. That query is written in a query language supported by the database (e.g. SQL).

As in other types of application, dynamic Web applications have failures as well. It was reported that there are common failures in a dynamic Web application that are caused by the interaction and passing of data between the application and the database [2]. Many researchers developed automated fault localization methods for traditional, non-database applications [1], [4], [5] and for data-centric applications (i.e. single-language database programs) [3], [6]. However, little attention has been paid to *database-aware* fault localization, i.e. taking into account the interaction between the applications and databases via queries, in dynamic Web applications.

Clark *et al.* [2] introduce an approach for database-aware fault localization in dynamic Web applications. It uses Tarantula [4] to assign each statement a suspiciousness score computed based on the percentage of passing/failing test cases executing that statement. However, it has a key restriction on its effectiveness. It requires users to provide passing/failing

test cases that must produce *different unique queries* at runtime, i.e. produce different unique SQL structures in which one structure is exercised by all failing test cases. This is too strict since all passing/failing tests often create SQL queries with the same structure and only literal values vary for each query.

This paper presents SQLook, a novel method for database-aware fault localization in PHP-based Web applications with SQL support. We focus on the output errors caused by incorrect queries with erroneous WHERE clauses or by the manipulation of the queries' result in PHP. We use a *row-based test case technique* in which instead of using the entire output of data records as a test case, we leverage the presence/absence of individual data rows and their expected values to create more test cases, called *row-based test cases*. Specifically, an input of a PHP program and a present/absent data row in the output forms a row-based test case. If a presence/absence is expected, the test case is a passing one, and vice versa. Instead of passing the control to the database engine to run a command as in Clark *et al.* [2], we instrument into a PHP interpreter the code to execute the SQL query and to *monitor* the evaluation of the predicates of a WHERE clause to determine if they affect the output of individual data records. Based on whether a WHERE part is exercised frequently by passing/failing tests, it is assigned with a Tarantula suspiciousness score [4]. Since our row-based test cases are for individual data rows, to compute the scores for PHP statements, SQLook performs *row-based slicing* across PHP statements and SQL parts to record the PHP statements that are exercised in the output of a row. It returns a ranked list of suspicious PHP entities and SQL queries.

## II. MOTIVATION

Web applications often use data stored in databases, such as in the table shown in Table I. The Users database table contains four rows (*data records*). The five columns of the table indicate five *attributes* associated with each record. For example, the first record has the information on a user with the attributes ID=1, Name=Alice, Age=20, Gender=Female, and Country=USA. This data can then be accessed or updated from the Web application through SQL queries. For instance, the SQL query "SELECT Name FROM Users WHERE Age = 20" returns a *result set* containing the names of all users whose age is 20.

As a Web application interacts with a database through SQL queries, database-specific failures can occur. Figure 1 shows an example of a PHP function that produces incorrect output

TABLE I  
THE USERS DATABASE TABLE

ID	Name	Age	Gender	Country
1	Alice	20	Female	USA
2	Bob	20	Male	Canada
3	Carol	25	Female	Canada
4	Daniel	30	Male	USA

A PHP function with an SQL query error on line 3:

```
function displaySearchResults($age, $gender, $country) {
1 $con = mysql_connect('localhost', 'admin', 'password');
2 mysql_select_db('my_database', $con);
3 $sql = "SELECT Name FROM Users WHERE Age >= $age
  AND Gender = '$gender' OR Country <> '$country' ";
4 $result = mysql_query($sql);
5 while($row = mysql_fetch_array($result)) {
6   echo $row['Name'] . '<br />';
}
```

Expected SQL query on line 3:

```
3 $sql = "SELECT Name FROM Users WHERE Age >= $age
  AND Gender = '$gender' OR Country = '$country' ";
```

Fig. 1. An PHP function with an SQL query error

values due to an error in its SQL query. The purpose of the function is to display the names of the users from the Users table (Table I) that satisfy a searching criteria (by age, gender, or country). First, the connection to the database is established (lines 1-2, Figure 1). The \$sql variable (line 3) contains the SQL query that retrieves the users' names for a given search input. This query is then sent to the database server to be executed via the PHP function `mysql_query` (line 4), and the returned result set is stored in the variable \$result. Finally, the code on lines 5-7 is used to loop through the records in the result set and display the corresponding names of the users found via the search. Note that this function contains an SQL fault: on line 3, the operator in the last predicate `Country <> '$country'` of the SQL query should be '=' instead of '<>'. Due to that error in the SQL query, the function does not display the expected results. Figure 2 shows the actual and expected output values given the search input `$age=25, $gender='Female', $country='USA'`. In the correct SQL query, the three-predicate condition determining the values in the returned result set is: `Age >= 25 AND Gender = 'Female' OR Country = 'USA'`. Since the actual query has a fault in the last predicate, there is a mismatch between its actual and expected outputs. As seen, Alice's and Daniel's names are expected to be found in the result, but are not displayed. Meanwhile, Bob's name is included in the actual output although it must not.

Due to that error in the SQL query, the function does not display the expected results. Figure 2 shows the actual and expected output values given the search input `$age=25, $gender='Female', $country='USA'`. In the correct SQL query, the three-predicate condition determining the values in the returned result set is: `Age >= 25 AND Gender = 'Female' OR Country = 'USA'`. Since the actual query has a fault in the last predicate, there is a mismatch between its actual and expected outputs. As seen, Alice's and Daniel's names are expected to be found in the result, but are not displayed. Meanwhile, Bob's name is included in the actual output although it must not.

Given such mismatch, it is not obvious which part of the program accounts for the error. In this example, the fault lies at a predicate of an SQL query, whereas the Web application is written in PHP. Moreover, the actual execution of the SQL query occurs at the database server, which is separate from the main PHP program where the SQL query is created. Thus, the process of localizing this type of fault needs to be *database-*

Search Input: `$age=25, $gender='Female', $country='USA'`

Actual SQL query:

```
SELECT Name FROM Users WHERE
Age >= 25 AND Gender = 'Female'
OR Country <> 'USA'
```

Actual output:

```
Bob
Carol
```

Expected SQL query:

```
SELECT Name FROM Users WHERE
Age >= 25 AND Gender = 'Female'
OR Country = 'USA'
```

Expected output:

```
Alice
Carol
Daniel
```

Fig. 2. Output of the PHP function in Figure 1

	Test Cases	Sus.
function displaySearchResults(\$age, \$gender, \$country) {	1 2 ... n	
...	• • ... •	0.5
4 \$result = mysql_query(\$sql);	• • ... •	0.5
4* SELECT...Age>=? AND Gender=? OR Country<>?	• • ... •	0.5
5 while(\$row = mysql_fetch_array(\$result)) {	• • ... •	0.5
6   echo \$row['Name'] . ' ';	• • ... •	0.5
Pass/Fail Status	F P ... F	

Fig. 3. Suspiciousness scores of the PHP statements and SQL queries in Figure 1 computed using the technique by Clark et al. [2] and Tarantula metric

*aware*, namely taking into account the interaction between the Web application and the database via queries.

The state-of-the-art tool in Clark *et al.* [2] for database-aware fault localization in Web applications monitors SQL queries generated at runtime and computes the *suspiciousness* scores for those queries and their associated attributes, as well as the statements in the main program. Similar to other statistical fault localization methods (e.g. Tarantula [4], Ochiai [1]), the idea in computing such scores is that if a program entity is exercised by more failing tests than passing ones, it more likely contributes to the failure and assigned with a higher score.

Compared with the previous fault localization approaches, Clark *et al.*'s method [2] is database-aware in that it considers SQL queries or SQL attributes as program entities and also computes their suspiciousness scores. For example, Figure 3 illustrates the score computation for the program entities in Figure 1 including SQL queries. Line 4\* shows the query executed by the PHP statement `mysql_query` on line 4 at runtime, with the question marks indicating literal values (numbers/strings). If the `mysql_query` statement executes multiple *unique* SQL queries (each with a different set of attributes) in different runs, the scores of individual queries will be computed.

That computation is based on the idea that if some of unique SQL queries are executed by more failing test cases than passing ones, they will have higher scores. This strategy is useful when there are multiple unique SQL queries that expose different behaviors in passing and failing test cases. However, in practice, a PHP `mysql_query` statement often executes only one SQL query, in which the set of attributes is fixed, and the concrete SQL queries in different executions have the same structure and vary only at the literal values. For example, the unique query in this example (line 4\* of Figure 3) is "SELECT Name FROM Users WHERE Age >= ? AND Gender = ? OR Country <> ?", with the set of attributes {Name, Age, Gender, Country}.

Since there is one unique SQL query executed by the PHP `mysql_query` statement, the coverage of the SQL query in the passing and failing test cases is the same as the `mysql_query` statement, and therefore, its suspiciousness score does not provide further information about the location of the fault. Let us describe the score computation to illustrate this limitation. In Figure 3, the bullets indicate the program entities that are exercised by a given test case. At the bottom row, the letters P and F specify a passing and failing test case, respectively. Column Sus. shows the suspiciousness score  $S(e)$  for a program entity  $e$  using the Tarantula ([4]) metric:

$$S(e) = \frac{\frac{Failed(e)}{TotalFailed}}{\frac{Passed(e)}{TotalPassed} + \frac{Failed(e)}{TotalFailed}} \quad (1)$$

where  $Passed(e)$  is the number of passing test cases that execute  $e$ ,  $Failed(e)$  is the number of failing test cases that execute  $e$ , and  $TotalPassed$  and  $TotalFailed$  are the respective total numbers of passing and failing test cases.

Although there is a fault in the SQL query, its suspiciousness score (0.5) is the same as the PHP `mysql_query` statement that executes it. In fact, all the program entities always have the same score even when a different suspiciousness metric is used, since the same set of program entities is executed in every passing or failing test case (regardless of the test suite). From those suspiciousness scores, no further information on the fault is gained. This limitation motivated us to develop SQLook, a method to better localize database-specific faults.

### III. LOCALIZING FAULTY SQL QUERIES

In a PHP Web application, program faults may be found in regular PHP statements or those that interact with the database, called *database-interaction points* (e.g. line 4 of Figure 1). The goal of this step is to localize these faults, specifically to decide if a database-interaction point contains a fault(s) at its WHERE clause or not. To do that, SQLook uses Tarantula [4] to compute the suspiciousness scores for all entities in PHP and in SQL WHERE clauses. To avoid the issues as in Clark *et al.* [2]’s approach, we have following key design strategies:

(1) *Row-based test cases.* Instead of viewing the input and entire expected output from the database as one test case, we analyze individual data records in the actual and expected outputs to create row-based test cases. For example, in Figure 2, we have 4 row-based test cases: 1) the input  $\$age=25$ ,  $\$gender='Female'$ ,  $\$country='USA'$  and the output of Alice’s record, 2) that input and the absence of Bob’s, 3) that input and the output of Carol’s, and 4) that input and the output of Daniel’s.

(2) *Monitoring the execution of PHP and SQL entities via instrumentation.* The suspiciousness scores are given to PHP statements and SQL WHERE clause(s). Instead of passing the control to the database engine to execute an SQL command as in [2], SQLook instruments into an PHP interpreter the code to execute the SQL command (Figure 4) and to observe the evaluation of the WHERE clause with respect to every row-based test case. Because the WHERE clause’s value decides if the output of a row-based test case is present or not, SQLook needs to record if that clause is evaluated to True or False.

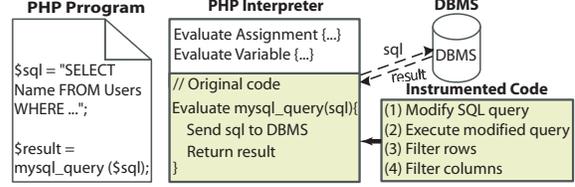


Fig. 4. Instrumented PHP interpreter to monitor the execution of SQL queries

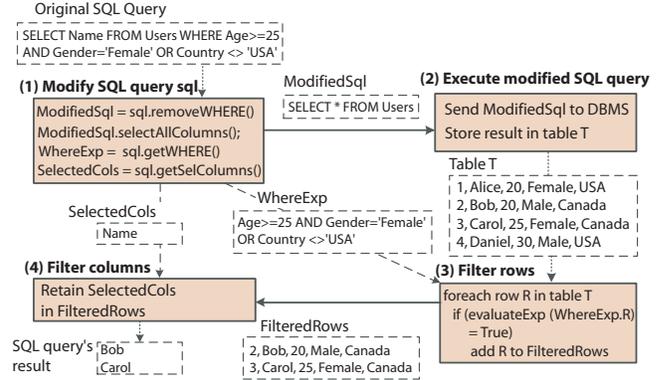


Fig. 5. Instrumented code to monitor the execution of SQL queries

Specifically, the operations that fulfill an SQL SELECT query consist of the following: (a) retrieving data from the database table(s) specified by the FROM part of the SQL query, (b) extracting the data records that satisfy the criteria specified by the WHERE condition of the SQL query, and (c) projecting the set of columns (attributes) given in the SELECT part of query into the final result set. As an example, the SQL query `SELECT Name FROM Users WHERE Age >= 25 AND Gender = 'Female' OR Country <> 'USA'` retrieves the names of all the users from the 'Users' table that meet the condition `Age >= 25 AND Gender = 'Female' OR Country <> 'USA'`. Our instrumented interpreter re-implements these three operations in four steps (see the instrumented code in Figure 4). Detailed instrumentation execution is in Figure 5.

From that, it knows which part (True or False) of a WHERE clause is exercised in a row-based test case. Thus, the suspiciousness metric can now be applied not only to PHP code, but also to the deeper level of the two parts of a WHERE clause.

(3) *Row-based Slicing across PHP and SQL.* Since SQLook uses row-based test cases, it needs to record the PHP statements that are exercised in the execution of such a test case, i.e. the PHP statements that are involved in the output of a data record (i.e. a row). To do that, during the monitoring in step (2), it computes the forward slice corresponding to each data row across the PHP statements and the two parts of WHERE.

(4) *Computing Suspiciousness Scores.* Based on the monitoring results, SQLook computes the suspiciousness scores for all program entities, including the WHERE conditions of SQL queries and other statements in the PHP program. Figure 6 illustrates the score computation for the example in Figure 1. A test case is marked as Passed(P) if the presence (or absence)

	Row-based test case				Sus.
	1-Alice	2-Bob	3-Carol	4-Daniel	
function display...(\$age, \$gender, \$country) {	•	•	•	•	0.5
...					
4 \$result = mysql_query(\$sql);	•	•	•	•	0.5
4a     WhereExp* = True	•	•	•		0.25
4b     WhereExp* = False	•			•	1.0
5 while(\$row = mysql_fetch_array(\$result)){	•	•	•	•	0.5
6     echo \$row['Name'] . ' ';		•	•		0.25
Pass/Fail Status	F	F	P	F	

Test case with \$age = 25, \$gender = 'Female', \$country = 'USA'  
 \* WhereExp: Age >= 25 AND Gender = 'Female' OR Country <> 'USA'

Fig. 6. Suspiciousness scores computed by SQLook for Figure 1

TABLE II  
 DATABASE-AWARE FAULT LOCALIZATION RESULTS

System	Ver	Files	LOC	Query	SQL faults		PHP faults	
					Mutants	%Rank	Mutants	%Rank
AddressBook	6.2.12	103	19K	52	30	100%	9	98%
SchoolMate	1.5.4	63	50K	295	54	100%	15	86%
ZenCart	1.3.9	1,118	156K	2,171	91	100%	24	90%

of the corresponding record in the actual output is as expected; otherwise, it is marked as Failed(F). For example, Alice does not appear in the actual output, which is not expected; thus record 1 is a Failed test case. The only Passed test case is record 3, where Carol is output as expected. The bullets for the statements indicate whether the entities are included in the slice for the corresponding test case, which is established when SQLook monitors the execution trace of row-based test cases.

As seen in Figure 6, the False part of the WHERE expression on line 4b has a high suspiciousness score (1.0), indicating that the program state when the WHERE of the SQL query is evaluated to False is likely incorrect (i.e., the result set does not contain the corresponding record whereas the record is expected to be included). Also, the suspiciousness score of the WHERE expression corresponding to the False case is higher than the score of any other program entity, which suggests that the predicates in the WHERE clause of the SQL query are most likely to contain an error. In this example, the fault is located at the last predicate of the WHERE condition (the operator in Country <> '\$country' should be '=' instead of '<>'). Thus, the scores are useful in localizing faulty SQL queries.

#### IV. EMPIRICAL EVALUATION

In this experiment, for each subject system, we manually seeded two types of database-related faults: (1) SQL faults in the WHERE clause of SQL queries, and (2) PHP faults that affect the output data retrieved from a database query. For (1), we mutated the operators of the predicates in the WHERE clauses, while for (2), we used the same mutation strategy as in the experiment in Clark *et al.* [2]. Each mutant program has a single fault. Then, we created a failed test case that was resulted from the fault in either an SQL query or the PHP code. Table II shows the result. Column Mutants shows the number of created mutants. Column SQL faults/% Rank shows

TABLE III  
 SQL QUERIES WITH UNIQUE SET OF ATTRIBUTES

#predicates	0-1	2-3	4-7	8-10	Checked	Unique
AddressBook	17	2	15	0	34	29
SchoolMate	181	25	3	0	36	36
ZenCart	755	431	135	8	36	36
Total					106	101

the percentage of statements in the execution trace that a fixer need not examine by using SQLook's ranked list of suspicious statements. For example, for PHP faults, (s)he does not have to examine 86-98% of the statements in the execution trace. Since the faulty SQL query is ranked at the top by SQLook, column SQL faults/% Rank shows 100% for all three systems.

**Comparison.** Since SQLook uses information about individual rows in the test case, SQLook is able to rank the likelihood of faulty entities with one test case only. In contrast, the state-of-the-art approach, Clark *et al.* [2], was designed to require multiple test cases to localize faults. For comparison, we took 106 randomly sampled SQL queries and manually examined if a query involves a unique set of attributes and varies only at the literal values. To do that, we first divided the queries into groups according to the numbers of their predicates. The number of sampled queries in each group is proportional to its size. In Table III, the first columns show the numbers of SQL queries with the corresponding numbers of predicates. Columns Checked and Unique show the numbers of queries that were checked and have unique sets of attributes, respectively. As seen, among 106 random query samples, 101 of them involve a unique set of attributes. Clark *et al.* [2] could not give those SQL statements higher suspicious scores even with multiple test cases, thus, could not locate those 101 faults. In contrast, SQLook was able to rank all of them at the highest.

#### V. CONCLUSIONS

SQLook is able to detect output faults in PHP statements and in SQL queries. The key solutions include (1) an instrumented PHP interpreter to monitor the execution of a query and the evaluation of predicates to see if they affect the output of records, and (2) row-based slicing across PHP and SQL to record entities in the output process of each row. Our empirical evaluation shows that SQLook can achieve high accuracy.

#### ACKNOWLEDGMENT

This project is funded in part by US National Science Foundation (NSF) CCF-1018600 and CNS-1223828 awards.

#### REFERENCES

- [1] R. Abreu, P. Zoetewey, and A. J.C. van Gemund. On the accuracy of spectrum based fault localization. In TAICPART-MUTATION'07, IEEE.
- [2] S. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. Localizing SQL Faults in Database Applications. ASE'11. IEEE, 2011.
- [3] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, D. Weiss. Customization change impact analysis for erp professionals via prog. slicing. ISSTA'08.
- [4] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. ASE, pp. 273-282. ACM, 2005.
- [5] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. PLDI'05, pages 15-26. ACM, 2005.
- [6] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, S. Chandra. Fault localization for data-centric programs. FSE'11. ACM, 2011.

# On the Personality Traits of StackOverflow Users

Blerina Bazelli, Abram Hindle, Eleni Stroulia

Department of Computing Science

University of Alberta

Edmonton, Alberta, Canada

{bazelli,hindle1,skoulia}@ualberta.ca

**Abstract**—In the last decade, developers have been increasingly sharing their questions with each other through Question and Answer (Q&A) websites. As a result, these websites have become valuable knowledge repositories, covering a wealth of topics related to particular programming languages. This knowledge is even more useful as the developer community evaluates both questions and answers through a voting mechanism. As votes accumulate, the developer community recognizes reputed members and further trusts their answers. In this paper, we analyze the community’s questions and answers to determine the developers’ personality traits, using the Linguistic Inquiry and Word Count (LIWC). We explore the personality traits of StackOverflow authors by categorizing them into different categories based on their reputation. Through textual analysis of StackOverflow posts, we found that the top reputed authors are more extroverted compared to medium and low reputed users. Moreover, authors of up-voted posts express significantly less negative emotions than authors of down-voted posts.

## I. INTRODUCTION

Question and Answer (Q&A) websites have gained significant ground as the preferred forums for developer interactions, in part due to support they offer for users to up-vote/down-vote questions and answers, accept answers as correct and edit the posts of others [1], [2]. StackOverflow.com<sup>1</sup> (StackOverflow) is one of the most popular Q&A websites focused mainly on questions related to programming languages.

Since StackOverflow posts are written in natural language, comprehensibility metrics and textual context analysis tools can potentially provide us with valuable information on what makes a posting perceived as trustworthy by the community. Thus, to take into account the semantics of the text, we have also applied a text analysis tool, the *Linguistic Inquiry and Word Count* (LIWC), which uses an embedded contextual dictionary [3], [4]. The LIWC tool has been used to identify the personality traits of Facebook users [5] as well as the anti-social personality types [6].

This study is a partial replication of Rigby and Hassan’s work [7] (Questions 1 and 2). They analyzed the Apache web-server developers’ personalities using their development mailing-list. They compared the personalities of the top 4 developers with two categories of developers having sent more than 30 messages and less than 30 messages respectively. The extra features of Q&A websites (compared to mailing-lists) allow us to also examine the personality types of the authors from different perspectives (Questions 3 and 4). The main goal of our study is to analyze and compare the authors’ personality types, based on the five most common personality traits [3]. Furthermore, we compare our results with Rigby and Hassan’s work to determine differences and commonalities of personality traits among mailing-list developers and StackOverflow authors.

<sup>1</sup><http://stackoverflow.com/>

Our research questions are:

- 1) Do the top reputed authors’ personality types differ?
- 2) Do author personalities vary by reputation?
- 3) Do authors of posts belonging to the same topics share similar personality types?
- 4) Do personality traits have an impact on the process of up-voting/down-voting questions and answers?

## II. RELATED WORK

There is much research based on StackOverflow’s open data-set [8]. Mamykina et al. [1] analyzed StackOverflow data in order to find the reason why StackOverflow has grown so rapidly since its inception. They found that most of the questions asked were quickly answered. Via interviews they found that factors such as reputation scores and badges motivated participation. Treude et al. [2] categorized the questions being asked on StackOverflow. They found that questions belonging into the categories of “review”, “conceptual” and “how-to” are most likely to be answered. Barua et al. [9] analyzed the LDA topics of StackOverflow users’ posts. They discovered topic trends of increasing popularity (e.g., Android, iPhone development) and decreasing popularity (e.g., Perl, Blackberry development).

Pennebaker et al. [3], [4] tried to associate words with one’s personality traits based on a study performed on 1,203 introductory psychology students. They developed the LIWC tool, which includes a dictionary and 72 language dimensions. Each dimension has a value which is calculated based on the frequency of words related to this particular dimension. However, only some of these are related to the “Big Five Personality Traits” (Neuroticism, Extroversion, Openness, Agreeableness and Conscientiousness). LIWC was used in a study by Summer et al. [6] to analyze and predict anti-social traits (“Dark Triad Personality Traits”) of Twitter users. Kramer et al. [5] used LIWC to study emotional expressions on Facebook and found evidence of emotional contagion (spreading of similar sentiment) between users.

This study aims to replicate the work done by Rigby and Hassan [7] on the Open Source Software (OSS) developers, by applying it to StackOverflow developers. Rigby and Hassan gathered data from the Apache server mailing list and used LIWC to determine the personality traits of OSS developers and whether their personality changes as they become more active. Moreover, they aimed to discover what personality traits are associated with the developers’ emotional state before and after an Apache version is being released. According to their results, 2 out of the top 4 developers had similar personality traits and differ from the general population. Moreover, a decrease in the developers’ positive emotions correlated with their imminent departure from the project. Finally, the textual analysis of the developers’ e-mails revealed that before an Apache version was released, their e-mails were composed of words that expressed mostly optimistic feelings.

### III. DATA COLLECTION AND TEXT ANALYSIS

We have analyzed the data of the six XML files provided by the 2013 MSR challenge [8], [10]. They include all the questions and answers posted on StackOverflow from August 2008 to August 2012, along with their authors' details.

The way a person writes and the different kind of words they select can reveal, to some extent, their personality types [3]. We use a text-analysis tool, LIWC [3], to analyze the posts and consequently to define the personality traits of the authors. LIWC includes a dictionary of 2700 words and word stems. The dictionary is divided into several categories (such as social processes, affective processes, cognitive mechanisms etc.) and sub-categories (such as insight, causation, discrepancy and tentativeness). At this point, it should be noted that when we refer to an author as neurotic, extroverted, open, agreeable or conscientious, we refer to their LIWC scores corresponding to their text corpora and not the authors themselves.

According to Pennebaker et al. [3], [4], some of the LIWC measures are correlated with the "Big Five Personality Traits":  
a) "Neuroticism": is associated with negative emotions such as anxiety, anger or envy and therefore is correlated to the presence of negative emotional words.

b) "Extroversion": expresses an emotional state where the person feels the need to be more sociable and interactive with others. Therefore, "Extroversion" depends on the presence of social and positive LIWC measures as well as the absence of tentative and negative emotional measures.

c) "Openness": characterizes people who are open to new ideas and is positively correlated with tentativeness and negatively with the causation LIWC measure.

d) "Agreeableness": describes people who tend to agree with others and it was found that the dimension of articles is the most significant factor that determines this personality trait.

e) "Conscientiousness": is negatively correlated with negations and negative emotional LIWC measures.

Pennebaker et al. [4] found that some LIWC factors are correlated with each one of the personality traits. The correlation between a language dimension and a personality trait may be positive or negative. For example, the personality trait of extroversion is linearly modeled as:  $Extroversion = -Tentativity - Negations + Social + PositiveEmotion$ . The full models can be found in Pennebaker's et al. work [4].

### IV. METHODOLOGY

In order to answer our research questions, we have divided the authors into several groups according to their reputation (top 10, high, medium and low). We also have categorized the authors based on their posts' content (defined by the keywords with which they have been tagged) and also by the posts' votes.

The default reputation when a user registers on StackOverflow is 1. The data revealed that nearly 35% (457 627) of the users have neither asked any question nor written any response to existing questions. Since we lack information about these users, we do not include them in our analysis. For each StackOverflow question we applied the LIWC on each post separately. More specifically, our analysis method involves three steps:

*First*, as the posts within the XML file are in HTML format, we discard the HTML tags from the posts before applying LIWC. We also discard code snippets (text between `<code>` and `</code>`).

*Second*, we apply the LIWC tool on the text from the previous step. Based on the LIWC language dimensions, we compute the values of the "Big Five Personality Traits".

*Third*, we compare the distributions of the personality traits' values by using the ANOVA and Tukey's HSD test.

### V. RESULTS

#### A. Do the top reputed authors' personality types differ?

We selected the top 10 reputed users in order to see if they share similar personality traits. The reputation of the top reputed users ranges from 214 774 to 465 166. Next, we select all the posts belonging to these users (both questions and answers). By applying the LIWC tool to each post separately, we are able to calculate the values of the "Big Five Personality Traits". In order to compare the means of the distributions, we apply a one-way ANOVA test. The two hypotheses are:

- $H_0$ : The means among top authors are equal
- $H_1$ : The means among top authors are not equal

According to the ANOVA test, we reject the Null Hypothesis ( $Pr(> F) < 2e^{-16}$ ) for all the personality traits tested. To further analyze these results we use the Tukey's test (a post-hoc test that compares all the possible combinations of the means). See Figure 1a.

Based on the TukeyHSD test with respect to neuroticism, the range of the 95% confidence intervals of neuroticism for all authors was from -0.67 to 1.02 (0 being none). This indicates there is a mild difference among authors in terms of neuroticism. The results of Tukey's test for the rest of the personality traits reveal that there are several combinations of authors who indeed share similar personality traits in terms of extroversion, openness, agreeableness and conscientiousness.

#### B. How do the personalities vary by reputation?

Are there differences in the personality traits of top, medium and low reputed users? We decided to investigate the relationship between personalities and reputation. We found out that the distribution of the authors' reputation was following a skewed distribution, similar to many countries distribution of wealth. There are few authors (the top reputed authors) who have very high reputation. On the other hand, the vast majority of StackOverflow users have a low reputation score. Therefore, we distinguish the authors into three main categories as follows: 1% top reputed authors, 10% medium reputed authors and the remaining 89% low reputed authors. Our hypotheses for the ANOVA test are:

- $H_0$ : The means of top-medium-low reputed authors are equal
- $H_1$ : The means of top-medium-low reputed authors are not equal

The ANOVA test exposed that there are statistically significant differences among top, medium and low reputed users ( $Pr(> F) < 2e^{-16}$ ). See Figure 1b. A further analysis with Tukey's HSD test exposed that there are differences among all groups as we expected. More specifically, the highest difference is between low and top reputed users. The score of neuroticism of low reputed users is much higher than top reputed users. This may suggest that authors who express more neuroticism through their text corpus are not being "awarded" by other users in terms of reputation points. Furthermore, we observe that there is a difference between medium and low reputed users. Again, the more reputed an author, the lower their measured neuroticism. Finally, despite the fact that there is a statistically significant difference between top and medium reputed users (top users less neurotic than medium reputed users), the mean is not much higher.

In terms of extroversion the highest discrepancy occurs between top and low reputed users. Furthermore, top reputed users exhibit more extroversion compared to medium reputed users who exhibit more extroversion compared to the less reputed ones. These results and the neuroticism results, hint that highly reputed authors exhibit more extroversion and lower neuroticism.

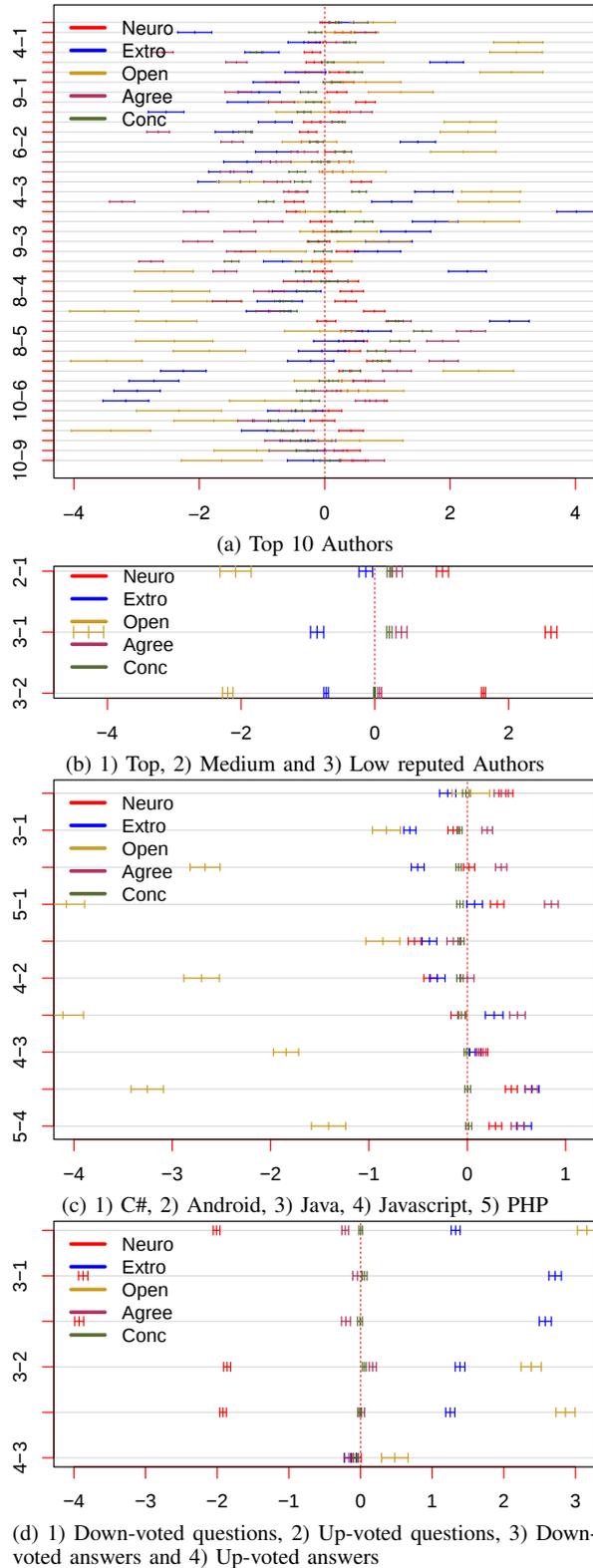


Fig. 1: 95% Confidence Intervals of difference of means between Top 10 Authors, Reputed Authors, Tags, Down/Up Questions/Answers. 2-1 means the difference of means between 2) and 1) (e.g. C# and Java or Author 1 and 2).

As mentioned above the personality trait of *openness* characterizes one who is open to new ideas and/or shares their ideas with others. Prior to this study we theorized that top reputed authors could exhibit more openness as they were the most experienced users on StackOverflow. Our results show that top reputed authors are more open compared to both, medium and low reputed users with the highest difference among top and low reputed authors.

All the three categories also differ significantly in terms of agreeableness. Top reputed authors are more agreeable compared to the other two categories with the less agreeable authors to be the low reputed ones.

Finally, medium and low reputed users tend to have similar degree of conscientiousness according to Tukey's HSD test. Nonetheless, there is a difference among top and medium reputed authors with the top reputed authors appearing more conscientious whereas medium reputed authors appeared more conscientious than low reputed users.

### C. Do authors of posts belonging to similar topics share similar personality types?

A tag is a label that describes the content of the question being asked and thus, helps categorize questions by topics such as "Web Development", "JavaScript", "Authentication" etc. When a user asks a question, StackOverflow forces users to add at least 1 tag. We examined the case of tags to be misleading; as StackOverflow allows most of the users to edit others' posts, in such cases "moderators" edit the vast majority of posts by replacing misleading tags with appropriate tags. Moreover, new tags are automatically removed if they are not used by at least one other question in a 6-month period. We found that the first 5 most common tags in descending order are: C#, Java, PHP, JavaScript and Android.

We found all the posts tagged as each one of the above tags for each author. Then, we computed the values of the "Big Five Personality Traits" for each post. As long as an author has multiple posts that correspond to one of the five most popular tags, we compute the mean of these values per author; therefore, we end up with a distribution composed of the mean values for the personality traits for each user. For instance, for an author who has written 5 posts tagged as "Java", we compute the mean of the personality traits' values and associate the author with the "Java" category.

Before applying the ANOVA test, we define the following hypotheses:

- $H_0$ : The means of the authors' personality traits tagged for each of the most popular 5 tags are equal
- $H_1$ : The means of the authors' personality traits tagged for each of the most popular 5 tags are not equal

The ANOVA test applied on these distributions showed that we should reject the null hypothesis ( $Pr(> F) < 2e^{-16}$ ). See Figure 1c. With that being said, there are statistically significant differences among the authors belonging in different tag categories. The Tukey's HSD test exposed that authors with posts tagged as "Android" tend to be slightly more neurotic compared to authors who had posted posts related to "Java", "JavaScript" and "PHP". Also, according to Tukey's HSD test, authors with posts tagged as "C#" are less neurotic than authors having posted posts related to "Android" and "PHP". Finally, there is no statistically significant difference among owners of posts associated with "C#" and "JavaScript".

Furthermore, we explore the rest of the personality traits in order to see if the tags follow a pattern similar to the authors belonging into the three categories according to their reputation (top, medium and low). As we can see in Figure

3b, the personality traits of extroversion varies among different tags. More specifically, authors of posts tagged as “C#” exhibit more extroversion than those with posts related to “Android”, “Java” and “JavaScript”. Authors related to “C#” programming language, follow the same pattern with authors belonging to the top reputed users (Less Neurotic-More extroverted). On the other hand, authors of “PHP” related posts are more extroverted when compared with “Java”, “JavaScript” and “Android” related posts.

Although there is no statistically significant difference among authors who have been written posts related to “C#” and “Android”, the former are more open to new ideas than authors of posts related to “Java”, “JavaScript” and “PHP” while “PHP” related authors are less open compared to “JavaScript”, “Java” and “Android” related ones.

As for the personality trait of Agreeableness, the less agreeable authors are those related to “C#” and “Java” posts. The most interesting personality trait is “Conscientiousness”, as authors belonging in several tag categories appear to be equally conscientious (“C#”-“Android” and “Java”-“JavaScript”-“PHP”). Although, Tukey’s test showed that there is statistically significant difference among the other combinations, the difference between the means is very small.

#### D. How much do personality traits differ between up-voted and down-voted authors?

Since the personality types among authors of questions and answers may not be similar, we divided the posts into questions and answers based on their votes. We came up with 4 distinct categories: *Down-voted questions*, *Up-voted questions*, *Down-voted answers* and *Up-voted answers*.

We noticed that 46% and 37% of all answers and questions respectively did not have any votes. After excluding the “vote-free” posts, questions and answers with at least 1 positive vote are called Up-voted, while posts with less than 0 votes are called Down-voted. We compare the authors belonging in these categories as follows: Down-voted questions vs. Up-voted questions and Down-voted answers vs. Up-voted answers as the nature of a question being asked is different from an answer. It should be noted that some authors may be double counted as they may have both Up-voted posts and Down-voted posts.

- $H_0$ : The means of the authors’ personality traits for each category to be compared are equal
- $H_1$ : The means of the authors’ personality traits for each category to be compared are not equal

The ANOVA test results in  $Pr(> F) < 5.3e^{-6}$  for all the personality traits; therefore, we reject the Null Hypothesis.

As mentioned above we focus on comparing authors of questions and answers separately. Based on Figure 1d, authors belonging to the category of down-voted questions expressed more neuroticism compared to the up-voted ones. We cannot assume the same result for the answers, as according to Tukey’s HSD test there is no significant difference among up-voted and down-voted answers.

Despite the fact that we expected authors of down-voted questions to be less extroverted, these questions exhibit more extroversion than up-voted ones. Authors of up-voted answers exhibited more extroversion than the down-voted answers.

We theorized that authors of up-voted posts might be more open than authors of down-voted posts. Authors of up-voted questions and answers had higher openness means, compared to authors of down-voted questions and answers respectively. However, they exhibit less agreeableness and conscientiousness.

## VI. CONCLUSIONS

In this paper we analyzed the personality properties of top, medium and low reputed authors, authors of most popular tags and most Up-voted and Down-voted posts on StackOverflow by replicating Rigby and Hassan’s work [7], who analyzed the personality traits of the top contributors of the Apache web-server project using the development mailing-list. According to our results, some of the top reputed authors share similar personality traits which matches Rigby and Hassan’s pattern (2 out of 4 top developers within Apache shared similar personality traits). We also found out that the top, medium and low reputed authors differ in Neuroticism, Extroversion, Openness, Agreeableness and Conscientiousness. Top reputed authors are less neurotic, more extroverted and open compared to medium and low reputed users who may just have entered the StackOverflow community. This difference may imply that posters who exhibit less neuroticism and more extroversion gain more popularity and reputation. Rigby and Hassan’s [7] conclusion differ; they report similar Extroversion and Openness measures between top authors and the general population.

Furthermore, Tukey’s HSD test shows that authors related to posts tagged as “Android” exhibit more neuroticism than authors with posts tagged as “Java”, “JavaScript” and “PHP”. Authors related to “C#” follow the same pattern as the top reputed users: less neuroticism and more extroversion. Yet authors of “PHP” related posts exhibited more extroversion than authors of “Java”, “JavaScript” and “Android” posts.

These results could serve as a measure that managers can use to hire programmers who can ask and answer questions effectively. In the future, we will focus on analyzing the variation of StackOverflow authors’ personalities over time. Furthermore, we would like to investigate the different kind of the personalities of question askers and question respondents and if they relate to teamwork abilities.

## REFERENCES

- [1] L. Mamykina, B. Manoim, M. Mittal, G. Hripesak, and B. Hartmann, “Design lessons from the fastest q&a site in the west,” in *Proceedings of the 2011 annual conference on Human factors in computing systems*. ACM, 2011, pp. 2857–2866.
- [2] C. Treude, O. Barzilay, and M.-A. Storey, “How do programmers ask and answer questions on the web?: Nier track,” in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011.
- [3] J. W. Pennebaker, M. E. Francis, and R. J. Booth, “Linguistic inquiry and word count: Liwc 2001,” *Mahway: Lawrence Erlbaum Associates*, 2001.
- [4] J. W. Pennebaker, L. A. King *et al.*, “Linguistic styles: Language use as an individual difference,” *Journal of personality and social psychology*, vol. 77, no. 6, pp. 1296–1312, 1999.
- [5] A. D. Kramer, “The spread of emotion via facebook,” in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*. ACM, 2012, pp. 767–770.
- [6] C. Sumner, A. Byers, R. Boochever, and G. J. Park, “Predicting dark triad personality traits from twitter usage and a linguistic analysis of tweets,” in *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, vol. 2. IEEE, 2012, pp. 386–393.
- [7] P. C. Rigby and A. E. Hassan, “What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list,” in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 23.
- [8] J. Atwood, “Stack overflow creative commons data dump,” June 2009, <http://blog.stackoverflow.com/2009/06/stack-overflow-creative-commons-data-dump/>.
- [9] A. Barua, S. W. Thomas, and A. E. Hassan, “What are developers talking about? an analysis of topics and trends in stack overflow,” *Empirical Software Engineering*, pp. 1–36, 2012.
- [10] A. Bacchelli, “Mining challenge 2013: Stack overflow,” in *The 10th Working Conference on Mining Software Repositories*, 2013.

# Towards Identification of Software Improvements and Specification Updates By Comparing Monitored and Specified End-User Behavior

Tobias Roehm, Bernd Bruegge  
Technische Universität München  
Munich, Germany  
{roehm, bruegge}@in.tum.de

Tom-Michael Hesse, Barbara Paech  
University of Heidelberg  
Heidelberg, Germany  
{hesse, paech}@informatik.uni-heidelberg.de

**Abstract**—Support of end-user needs is an important success factor for a software application. In order to optimize the support of end-user needs, developers have to be aware of them and their evolution over time. But a communication gap between developers and users leads to ignorance of developers about how users use their application. Also, developer assumptions about user behavior are rarely tested and corrected if they are wrong. Consequently, many software applications have a mediocre support of user needs and user problems as well as changes in user needs are detected rather late.

In this paper, we present a research agenda addressing this problem by comparing use case descriptions to monitored user actions. More specifically, we propose to monitor user actions using instrumentation, detect the current use case of a user using machine learning, and compare use case steps to monitored user actions. By detecting differences between both, we identify mismatches between user behavior and developer assumptions reflected in use case descriptions. Those mismatches can serve as starting points to identify software improvements, to test the use case specification and identify updates, and to revise training programs. Finally, we sketch a plan to evaluate our approach.

**Keywords**—User needs, User monitoring, Use case detection, Comparison of observed and specified behavior, Specification testing, Reverse modeling, Machine learning, Software evolution

## I. INTRODUCTION

In a competitive software market end-users can choose among several rival applications and pick the application with the best support of their needs. The success of a software system in such a situation is determined by the acceptance of its end-users. Consequently, developers should cooperate with end-users to understand user needs, develop software that supports those needs, and ensure the continuous support during software evolution.

Unfortunately, a communication gap between end-users and developers exists [12] that hinders cooperation between developers and users. Developers might not have access to end-users, end-users might not be willing or able to express their needs and problems, or developers and end-users might misunderstand each other. The result of this gap is ignorance at the developers' side about how end-users use the application [19]. Also, developer assumptions about end-user behavior are seldom tested and corrected if they are wrong.

Consequently, many software applications have a mediocre support of user needs and user problems as well as changes in user needs are detected rather late.

In this paper, we propose an approach to address this problem. More specifically, we propose to instrument software applications to monitor user actions. Further, we propose to detect the use case a user is currently performing and compare its flow of events to monitored user actions. We propose to consider use cases as they document the assumptions of developers about how a user will use an application in a fine-grained, detailed way. We argue that detected differences between user actions and use case steps can be exploited in several ways. First, a difference can serve as starting point for further exploration and finally identify software improvements in terms of functionality or usability. Second, the use case specification can be tested based on the observation (specification testing) and updates can be identified if the use case specification is incorrect or incomplete (reverse modeling). Third, training programs for users can be designed or refined based on knowledge about application usage and difficulties users are facing. As developer assumptions about application usage are reflected and documented in the use case description, our approach allows comparing the assumptions of developers about application usage to actual usage obtained from monitored user actions. Our approach is complementary to existing approaches such as feedback mechanisms, on-site user observations, usability labs or user workshops.

The contributions of this paper are the following. First, we propose to compare monitored user actions to the use case specification of an application and sketch a corresponding approach. Second, we discuss how these differences can be exploited to identify software improvements, to test and update the use case specification, and to revise training programs for users.

This paper is organized as follows: In Section II we review related work. In Section III we describe our approach in detail. In Section IV we sketch an evaluation strategy for our approach and finally sum up in Section V.

## II. RELATED WORK

Maalej et al. [12], [13] described the problem of communication gaps between developers and end-users and proposed to consider user input as important information. We instantiate their generic framework and additionally propose to compare specified and monitored user behavior. Kim et al. [10] describe TRUE, an approach to collect user actions by instrumentation and exploit this information to improve video games. While their approach and goal is similar to ours, they do not compare monitored and specified behavior automatically.

*a) Use Case Mining:* El-Ramly et al. [3], [5], [6] developed an approach to recover use cases from monitored user actions and exploit that knowledge in user interface reengineering. Similarly, Antonio et al. [1] and Li et al. [11] developed approaches to recover a use case model from runtime information. While these approaches overlap with ours by the use case detection, they do not compare monitored user actions with specified use cases. We plan to reuse their work for use case detection.

*b) Comparison of Monitored and Specified Behavior:* Comparing monitored user behavior to a specification of expected user behavior has been studied by other researchers. Paternò et al. [15] propose an approach using task models, i.e. a hierarchical decomposition of a task, to capture expected behavior of users and compare them to monitored behavior. Feuerstack et al. [7] use UI models to generate user interfaces automatically and evaluate their usability by comparing monitored user behavior to the original models. We are currently evaluating whether and how we can reuse their work. Robinson [17], [18] proposes an approach similar to ours. He monitors user and system behavior and compares it to user goals specified by OCL-like statements. While Robinson focuses on abstract goals, we monitor and compare lists of user actions.

*c) Automated Usability Testing:* Several approaches have been proposed to monitor user actions and exploit them to (semi-) automate usability testing. Ivory and Hearst [8] review the state of the art of automated usability evaluation. Tao [21] proposes an approach to capture user interactions and analyze usability in early development phases. Several approaches have been proposed to evaluate the usability of web applications [2], [15] as well as mobile applications [9], [16]. Those approaches focus on usability evaluation while we do not only target usability problems but also discuss specification testing and updating.

## III. OUR APPROACH

In this section we describe our approach in detail.

### A. Motivating Example

In order to motivate our approach, we start with an example that is depicted in Figure 1. We consider the development of online banking software. Before the implementation of the software, developers talked to banking customers and identified the use case steps (see the left hand side of Figure 1). Now the software is implemented as a web application, deployed

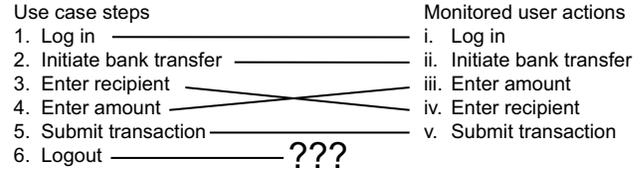


Figure 1. Bank Transfer Example

and end-users use it to do their online banking activities. A sensor within the web server monitors HTTP requests and identifies user actions from the URLs requested. For example, the initiation of a new bank transfer can be detected when a user initiates a HTTP request to the url /BankTransfer.do. All actions of a user are monitored (see the right hand side of Figure 1). When comparing use case steps and user actions, we detect that entering recipient data and amount are switched in order and the user forgot to logout, e.g. by closing the browser or surfing to another web page. These differences are presented to the developers of the banking application and they have to decide how to handle them. They decide to ignore the switched order of data entry and to add an automatic logout feature after a certain inactivity time of a user - a standard feature of online banking applications today.

### B. Research Questions

We address the following research questions.

- How can semantically meaningful user actions be obtained (RQ 1)?
- How can the current use case a user is performing be identified based on traces of monitored user actions (RQ 2)?
- How can use case steps and monitored user actions be compared (RQ 3)?
- How can differences between use case steps and monitored user actions be exploited (RQ 4)?

Regarding RQ1, monitoring of low-level user actions such as mouse clicks and text entered is easy, but we need semantically meaningful user actions on a similar level of abstraction as use case steps to be able to compare both. Further, we argue that detecting differences as such is not a worthwhile goal in itself but the exploitation of the knowledge gained to support evolution decisions and test developer assumptions about application usage (RQ 4).

### C. General Framework

In this section we sketch a framework that is necessary to monitor user actions and compare use case steps to monitored user actions. Figure 2 gives an overview of our framework. A user interacts with an application that is instrumented with one or more sensors. Sensors can be implemented using different implementation approaches - framework hooks, log file monitors, special monitoring code, or byte code instrumentation. Also, they can target different frameworks and hence exhibit a varying degree of application independence and reusability -

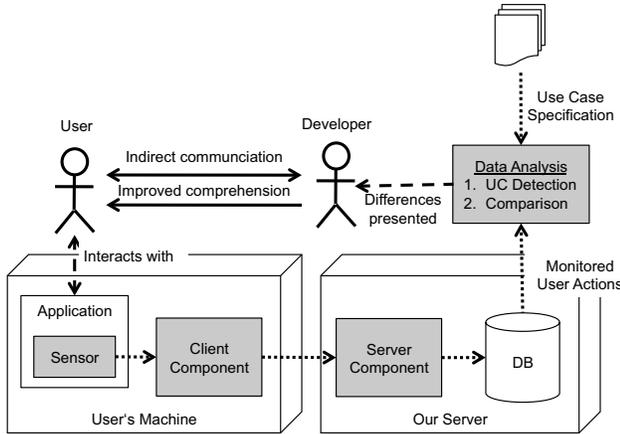


Figure 2. Framework Overview  
Grey boxes denote components of our framework.

from being completely independent of the application (e. g. a sensor monitoring log files), via enabling reuse in the same framework (e. g. a sensor for RCP framework) to being completely application dependent (e. g. a sensor integrated in the application source code). A sensor monitors user actions and sends them to a client component running on the user’s machine, too. The client component processes and aggregates the user actions and sends them to a server component, which stores them in a database. This architecture provides the possibility to process user actions on the client side and collect additional context information while performing the main processing on the server side to minimize the overhead on a user’s machine. The monitored user actions from the database are analyzed in two ways. First, the use case performed by the user among the set of specified use cases is detected. Second, the flow of events of the current use case is compared to monitored user actions and detected differences are presented to developers. Overall, we achieve an indirect, automated communication which is effortless for the user.

#### D. Detection of the Current Use Case

We are exploring machine learning algorithms to detect the current use case a user is performing based on his or her actions. As we can observe user actions directly but use cases not, Hidden Markov Models seem to be appropriate for our case. Saito et al. [20] developed an interesting, related approach targeting user tasks. Further, El-Ramly et al. [4]–[6], Antonio et al. [1], and Li et al. [11] did work on analyzing user actions and mine use cases models that we plan to reuse.

#### E. Comparison of Use Cases and Monitored User Actions

In this section we describe the steps necessary to compare the flow of events of a use case to monitored user actions.

1) *Abstracting Monitored User Actions:* In order to be able to compare use case steps and monitored user actions directly, they have to have a similar level of abstraction. We are experimenting with three strategies to abstract monitored

user actions. First, our sensors monitor user actions already at a high level of abstraction, i.e. not every mouse or key action but manipulation actions that consist of several mouse and key actions. Second, we use sequential pattern mining to detect frequent patterns in traces of monitored user action. When a frequent user action pattern is detected, the sequence of user actions forming this pattern can be replaced in the traces by a new, single, and more abstract user action. Third, we use a taxonomy of user actions in order to reason about user actions and abstract user action types.

#### 2) Mapping between Use Case Steps and User Actions:

In order to be able to compare use case steps and monitored user actions a mapping between both has to be established. This mapping establishes a relationship from use case steps to types of user actions. It denotes the relationship “This use case step can be represented by this type(s) of user action”. We are experimenting with two strategies to accomplish this mapping. In the first strategy, a developer assigns each use case step to one or more user action types that correspond to it. In the second strategy, we compare the similarity of the textual description of a use case step to the textual description of a user action in our user action taxonomy. If the textual similarity exceeds a threshold parameter, we map the corresponding user action to a use case step.

3) *Comparing Use Case Steps and User Actions:* Finally we compare a trace of monitored user actions to the flow of events of the current use case based on the mapping described above. We expect the differences to be additional/ missing steps and a different order of steps. Hence, we iterate over the steps of the current use case and check for each step whether a corresponding user action or user action pattern was monitored. Also, we check if there is a difference in the order.

#### F. Exploitation of Detected Differences

We argue that differences between use case steps and user actions are not good or bad in itself. Developers have to analyze them, determine how to handle them, and decide if any of the following situations applies. First, a detected difference can be the starting point for further analysis and finally trigger an improvement of the application. Examples of improvements are additional functionality, changed functionality, or removal of a usability problem. Second, if the monitored user behavior reflects a valid use of the application that is not documented in the current specification, the specification should be updated based on the observation. Third, a detected difference can trigger changes and tailoring of user training by taking information about user behavior and problems users are facing into account. Overall, a detected difference reveals a wrong assumption of developers (who wrote the use cases) about application usage and might trigger further investigation. Thereby, it helps developers to comprehend the behavior of users and improve the support of user needs within their application.

#### IV. EVALUATION STRATEGY

In this section we outline our plan to evaluate our proposed approach. We already implemented the monitoring part of the framework described in Section III. We implemented sensors that track user actions in Eclipse RCP applications and J2EE-based web applications [14]. Using these sensors we collected user action traces of five users that worked with an instrumented application for several weeks.

We plan to evaluate our approach with two case studies. In the first case study, we will monitor user actions using our sensors and ask users to create a protocol of the tasks/use cases they are performing. The obtained dataset allows us to learn predictors of the current use case and evaluate their prediction accuracy. In the second case study, we will monitor user actions using our sensors, compare them to the flow of events of the current use case, and show detected differences to developers. We will evaluate our approach in three directions. First, we evaluate its correctness, i.e. if the detected differences are real differences. Second, we evaluate its helpfulness for developers, i.e. if the detected differences help developers to improve their application or update the use case specification. Third, we evaluate its performance overhead, i.e. how much overhead our sensors introduce and if the overhead disturbs users in their daily work.

In the case studies we will use two real-world applications and their users and developers. The first application will be UNICASE<sup>1</sup>, an Eclipse-based CASE tool that supports UML modeling, rationale-based software engineering, and capture and use of project knowledge. The second application will be an application developed by a partner software company.

#### V. SUMMARY

In this paper we proposed an approach to detect the current use case a user is performing and to compare its flow of events to monitored user actions. Detected differences can be exploited to identify software improvements, to test the use case specification and identify updates, and to revise training programs for users. Overall, we aim to improve user comprehension on the developer's side and thereby help developers to maximize the support of user needs in their applications.

Our next steps will be to continue implementation of our approach and evaluate it by case studies with two real-world applications and their users and developers. Further, we will investigate the impact of privacy issues on our approach.

#### ACKNOWLEDGEMENTS

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution and the European Commission (FastFix project, grant FP7-258109).

<sup>1</sup><http://code.google.com/p/unicase>

#### REFERENCES

- [1] G. Antonio, D. Lucca, A. R. Fasolino, U. D. Carlini, N. Federico, and V. Claudio. Recovering use case models from object-oriented code: a thread-based approach. In *Proc. of the Seventh Working Conf. on Reverse Engineering*, pages 108–117. IEEE, 2000.
- [2] R. Atterer, M. Wnuk, and A. Schmidt. Knowing the user's every move - User activity tracking for website usability evaluation and implicit interaction. In *Proc. of the 15th Int. Conf. on World Wide Web*, pages 203–212. ACM, 2006.
- [3] M. El-Ramly and E. Stroulia. Mining software usage data. In *Proc. of the 1st Int. Workshop on Mining Software Repositories*, MSR 2004, 2004.
- [4] M. El-Ramly, E. Stroulia, and P. Sorenson. From run-time behavior to usage scenarios: An interaction-pattern mining approach. In *Proc. of the Eighth ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, KDD'02, pages 315–324. ACM, 2002.
- [5] M. El-Ramly, E. Stroulia, and P. Sorenson. Mining system-user interaction traces for use case models. In *Proc. of the 10th Int. Workshop on Program Comprehension*, pages 21 – 29. IEEE, 2002.
- [6] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In *Proc. of the 14th Int Conf. on Software Engineering and Knowledge Engineering*, SEKE '02, pages 447–454. ACM, 2002.
- [7] S. Feuerstack, M. Blumendorf, M. Kern, M. Kruppa, M. Quade, M. Runge, and S. Albayrak. Automated usability evaluation during model-based interactive system development. In *Engineering Interactive Systems*, volume 5247 of *LNCSE*, pages 134–141. Springer, 2008.
- [8] M. Y. Ivory and M. a. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516, 2001.
- [9] D. Kim and K.-p. Lee. Development of interactive logger for understanding user's interaction with mobile phone. In *Human-Computer Interaction. Interaction Platforms and Techniques*, volume 4551 of *LNCSE*, pages 394–400. Springer, 2007.
- [10] J. H. Kim, D. V. Gunn, E. Schuh, B. C. Phillips, R. J. Pagulayan, and D. Wixon. Tracking real-time user experience (TRUE): A comprehensive instrumentation solution for complex systems. In *CHI 2008 Proceedings*, pages 443–451. ACM, 2008.
- [11] Q. Li, S. Hu, P. Chen, L. Wu, and W. Chen. Discovering and mining use case model in reverse engineering. In *Proc. of the 4th Int. Conf. on Fuzzy Systems and Knowledge Discovery*, FSKD'07. IEEE, 2007.
- [12] W. Maalej, H. Happel, and A. Rashid. When users become collaborators: Towards continuous and context-aware user input. *Proc. of the 24th ACM SIGPLAN Conf. Comp. on Object Oriented Programming Systems Languages and Applications*, pages 981–990, 2009.
- [13] W. Maalej and D. Pagano. On the socialness of software. In *Ninth IEEE Int. Conf. on Dependable, Autonomic and Secure Computing*, DASC, pages 864–871. IEEE, 2011.
- [14] D. Pagano, M. Juan, A. Bagnato, T. Roehm, B. Bruegge, and W. Maalej. FastFix: Monitoring control for remote software maintenance. In *ICSE 2012 Proceedings*, pages 1437–1438. IEEE, 2012.
- [15] F. Paternò, A. Piruzza, and C. Santoro. Remote web usability evaluation exploiting multimodal information on user behavior. In *Computer-Aided Design of User Interfaces V*, pages 287–298. Springer, 2007.
- [16] F. Paternò, A. Russino, C. Santoro, and V. G. Moruzzi. Remote evaluation of mobile applications. In *Task Models and Diagrams for User Interface Design*, volume 4849 of *LNCSE*, pages 155–169. Springer, 2007.
- [17] W. N. Robinson. Seeking quality through user-goal monitoring. *IEEE Software*, 26(5), 2009.
- [18] W. N. Robinson. A roadmap for comprehensive requirements monitoring. *IEEE Software*, 43(5):64–72, 2010.
- [19] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *ICSE 2012 Proceedings*, pages 255–265. IEEE, 2012.
- [20] R. Saito, T. Kuboyama, Y. Yamakawa, and H. Yasuda. Understanding user behavior through summarization of window transition logs. In *Databases in Networked Information Systems*, volume 7108 of *LNCSE*, pages 162–178. Springer, 2011.
- [21] Y. Tao. Capturing user interface events with aspects. In *Human-Computer Interaction. HCI Applications and Services*, volume 4553 of *LNCSE*, pages 1170–1179. Springer, 2007.

# An Empirical Illustration to Validate a FLOSS Development Model using S-shaped Curves

Ana Erika Camargo Cruz, Hajimu Iida

Graduate School of Information Science and Technology  
Nara Institute of Science and Technology  
Nara-ken, Ikoma-shi, Takayama-cho 8916-5, 630-0192, Japan  
camargo@is.naist.jp, iida@itc.naist.jp

Norbert Preining

Research Center for Software Verification  
Japan Advanced Institute of Science and Technology  
Ishikawa-ken, Nomi-shi, Asahidai 1-1, 923-1292, Japan  
preining@jaist.ac.jp

**Abstract**—Open source software (OSS) or Free/Libre OSS (FLOSS) has become an interesting source of research in software engineering. However, it has been criticized that FLOSS development is often considered as a homogeneous phenomenon grounded by assumptions rather than empirical evidence. Proper empirical methods that can shed light into FLOSS development are desirable. In this paper, we propose an empirical method to validate a software development model for FLOSS, the Adapted Staged Model for FLOSS. We mined some selected metrics from Apache Ivy and study their evolution using S-shaped curves. Our results indicate that S-shaped curves can model software evolution well for Ivy. Moreover, we demonstrated that our method can be used to identify successfully different stages of its development, validating part of the Adapted Staged Model for FLOSS.

## I. INTRODUCTION

Research on software engineering (SE) has been limited due to the lack of industry projects available for research. Open source software (OSS) or Free/Libre OSS (FOSS / FLOSS) has become an interesting source of research. However, despite the growing successful solutions (methods, models, etc.) derived from this type of software, it remains questionable their application to software that follows a more traditional development, such as industry projects. Furthermore, it has been criticized that SE research describes OSS development as a homogeneous phenomenon grounded by assumptions rather than empirical evidence [1]. On the other hand, according to [2] there are conflicting views on what the OSS phenomenon actually is and there is not even consensus on which label to use on the phenomenon, OSS, FOSS/FLOSS. In this paper, we treat them as synonyms, and focus on how to validate empirically FLOSS development.

There are a few works which have attempted to model the very unique dynamic development of FLOSS, one of which is the *Adapted Staged Model for FLOSS* (ASMF) proposed by Capiluppi et al. [3]. In this paper, we study the evolution of a FLOSS to validate empirically this model, the reason behind of our choice is explained in a later section.

Most of the research on software evolution concerns empirical tests of the well-known laws of evolution of Lehman and Belady (result of earlier studies on the evolution of the IBM OS/360 in the late 1960's). Initially, this research reported studies involving industrial software, till around 2000 when some studies on some FLOSS were reported. While the majority of industry software showed a linear growth [4], a super-linear growth (non-linear) was observed for some FLOSS, not

supporting one of the laws that states the incremental growth rate of software is constant [3], [4], [5]. Furthermore, studies have reported that software evolution may be more accurately modeled as a non-linear function [4], [5], for which we propose to model software growth utilizing logistic curves of natural growth, S-shaped curves (to the best of our knowledge this is the first study employing these curves for software evolution).

Our contributions can be summarized as follows. We proposed a methodology to validate empirically the ASMF, based on modeling the evolution of some selected metrics using S-shaped curves. We present our initial results using as a case study four released versions of Apache Ivy, which indicate that S-shaped curves can model software evolution of Ivy well, because all the derived models fit well the data. Moreover, we demonstrated that our method can be used to identify successfully different stages of its development, validating part of the ASMF.

The remaining of this paper is organized as follows. In Section II, we present some background and related work. In Section III, we describe our methodology. In Section IV, we discuss our results and observations. Finally, we conclude this paper in Section V.

## II. BACKGROUND AND RELATED WORK

### A. Software Development Models for FLOSS

FLOSS development differs from traditional software development in many aspects. The most significant is perhaps that FLOSS development is volunteer driven and repositories are available to the public even before official releases. The continuous feedback from not only end users, but also developers, results in many more new releases and patches in comparison to traditional development. This dynamism is considered as a vitality factor [3], and could be the cause of the continuous evolution observed in many FLOSS, making them referred as long-lived software [3] or continuous growing software in a perpetual development [5].

There are a few works which have attempted to model this dynamic development of FLOSS, one of which is the *ASMF* of Capiluppi et al. [3] and more recently, the *Perpetual Model* of Feitelson [5]. Both are an adaptation of the *Staged Model* of Rajilich and Bennett, which describes the software life cycle taking into account various maintenance tasks (adaptive, corrective and preventive) [6].

The ASMF was built upon the observations from various case studies of FLOSS and traditional software development. Their authors first analyzed when and how differences and commonalities arise between the two types of development, from this analysis their model was derived (Figure 2). The details of each of its stages are provided in a later section.

On the other hand, the perpetual model was inspired on Linux. This model adds another dimension to the Staged model considering a continually growing development backbone, from which stable production versions branch out at intermittent release points.

We think that the latter model could be embedded into the former. For example, the continually growing development backbone, as referred in the perpetual model, could be described by [Initial Development] → [Evolution Loop] in the ASMF; and branches development by [Initial Development] → [Evolution] → [Servicing] → [Phase Out]. Although this would require a more strict validation, this paper only focuses on validating empirically the ASMF.

### B. S-shaped curves

The different stages of the ASMF are characterized by different activities or tasks. We are interested to know how differently these are carried out throughout the software development. Selection of metrics describing them and their analysis via growth rates can aid us in such a differentiation. Frequent models used for software evolution include linear, quadratic and in general of polynomial form. Unfortunately, except for the linear model, growth rates cannot be analyzed directly. Thus, we are not only interested in models that can model software evolution well, but also enable us to provide clear interpretations of how the selected metrics evolve. As mentioned earlier, because previous research has suggested that software evolution may be more accurately modeled as a non-linear function, we chose to apply S-shaped curves, which are also characterized for their small number of parameters, clear interpretation and easy application.

S-shaped curves have been applied for describing the evolution of various types of systems including technological, economical, social and software reliability [7], [8]. Its function is defined by the equation:

$$G(m) = \frac{\kappa}{1 + e^{-(\alpha + \beta t)}} \quad (1)$$

Where,  $G(m)$  is the growth function of an attribute of interest  $m$  overtime, measured in  $t$  units,  $\kappa$  is the asymptotic limit of growth or ceiling,  $\alpha$  is a growth rate parameter that specifies the steepness or width of the S-curve; and  $\beta$  helps us to estimate the time  $t_{midpoint}$  when the curve is symmetric and reaches 0.5  $\kappa$ , by calculating  $-\alpha/\beta$ . It is often helpful to replace the use of  $\alpha$  for a variable that specifies the time required for  $m$  to grow from 10% to 90% of  $\kappa$ . This period is called the characteristic duration, or  $\Delta t$ . Through simple algebra,  $\Delta t = \ln(81)/\alpha$ [9]. An example of a S-shaped curve is shown in Figure 1.

The essential meaning of this function is that the rate of growth is proportional to both the amount of growth already accomplished and the remaining to be accomplished. S-shaped

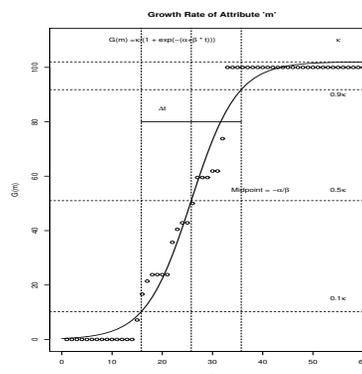


Fig. 1: A S-Shaped Curve

curves can be also applied for (short-term) forecasting of the values of the ceiling and steepness of the growth [7].

## III. METHODOLOGY

Our approach to evaluate empirically the ASMF consisted in collecting several metrics describing different activities carried out during the development of a FLOSS, Apache Ivy, and model their evolution using s-shaped curves. In this section, we first describe briefly the ASMF according to [3], then we discuss which metrics can help us to assess the different stages of this model. After that, we present our results of applying our method to Ivy in Section IV.

### A. The Adapted Staged Model for FLOSS

The different stages of the ASMF in Figure 2 are described as follows.

**Initial Development [I].** Traditionally this stage includes the well-know phases: Design, first coding and testing; and no releases are considered. However, in the case of FLOSS, releases may be available even before their completion. Some projects may never leave the initial stage, as documented for a large majority on the projects hosted by SourceForge, the initial stage can turn into a Bazaar stage (large increase of developers). In Figure 2, this is represented with a highlighted box for [I].

**Evolution [E].** Continuous Feedback is provided by users (bugs, changes, new requirements), as a consequence enhancements and new functionalities are made. New releases and patches are available more often (vitality factor). The loop in Figure 2 characterizes long-lived software such as Windows, Free-BSD and Open BSD. Long-live software are reported to have linear and super-linear growth rates.

**Servicing [S].** Typically the system is considered mature, new functionalities are not added, whilst fixes are still performed to the code base. FLOSS shows stabilization points and its size overall does not change. Albeit, several releases are available. After Servicing, either a new evolution period or complete abandonment can occur.

**Phase Out [P].** Typically no implementation of new functionalities, nor fixes occur. For FLOSS, new developers may

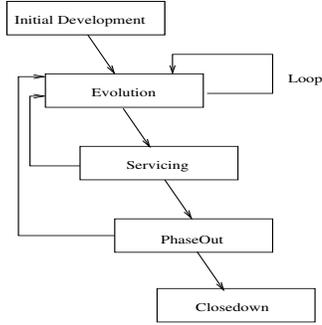


Fig. 2: The Adapted Staged Model for FLOSS [3]

TABLE I: FLOSS Stages and Proposed Metrics' Behavior

Stage	Size	Fixed Bugs	Improvements	New Features
Initial [I]	↑	↑	↑	*
Evolution [E]	↑	↑	↑	↑
Servicing [S]	↔	↔	↔	*
Phase Out [P]	*	*	*	*

↑(Fast growth), ↗(Slow growth), ↔(Signs of Stabilization), \*(No growth)

take over the existing system, in consequence, a new evolution phase can take place.

### B. Selected metrics

According to the previous description, we propose the following metrics to differentiate the different stages of the ASMF: Size in LOC, number of fixed bugs, number of improvements and number of new features (functionalities/requirements). LOC would grow increasingly during [I], starting to stabilize at the end of [E] or some time during [S]. A major number of bugs would be fixed during [E], starting to decrease at [S]; the same would occur with number of improvements. New features would be developed only during [E]. A summary of these hypotheses are shown in Table I.

The selected metrics were collected from logs and source code of the code repository (trunk) of Ivy, and reports of its issue tracking system, where issues are classified into bugs, improvements, new features and others. The collection was performed for every single Java file of a selected package of Ivy at all different times in which a commit in the code repository was performed. The resultant dataset was a daily set of measures of each of the files. If in any given day, no commit existed, the latest measures recorded were duplicated only for LOC, for the rest of the cases 0 was recorded. Four different time frames were explored which correspond to four releases of Ivy (2.0, 2.1, 2.2 and 2.3<sup>1</sup>).

Using S-shaped curves, we modeled growth of the selected metrics; cumulative values were used for all of them, but LOC, and measures were normalized from 0 to 100%.

## IV. RESULTS AND OBSERVATIONS

All the coefficients of our models were highly significant, to an alpha value smaller to the traditional used (0.05). We

<sup>1</sup>Versions prior to 2.0 were released by a different organization, and are not endorsed by the Apache Software Foundation.

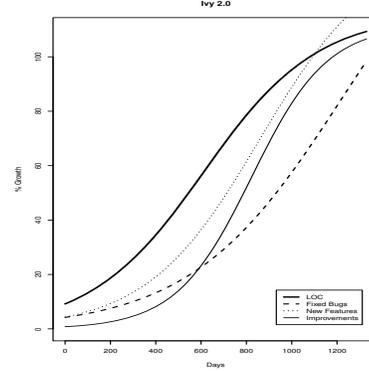


Fig. 3: Regression Curves of Ivy 2.0

calculated the overall fitness of our models using  $R^2$ , the results ( $R^2 > 9.0$ ) suggest that all models fit well the data under exploration.

Table II shows the values of the regression coefficients, as well as the most important parameters describing each of the curves: The characteristic duration ( $\Delta t$ ) and the midpoint (50%) of the growth of the curve ( $-\alpha/\beta$ ). These parameters were also used to calculate 90% and 10% of the total growth of the curves.

### A. Release 2.0 (Jun-01-2005 to Jan-20-2009, 1330 Days)

LOC reached the fastest 90% of its total growth at day 1302 approximately, followed by number of improvements, which reached 90% of its total growth at day 1243 approximately. Number of new features and number of fixed bugs grew to a slower rate, reaching only 50% of its total growth at day 955 and 1167 respectively. An interesting observation is that these curves obtained ceiling values ( $\kappa$ ) much higher than 100%, which was interpreted as a possible immediate growth after Day 1330. Figure 3 shows a representation of these regression curves.

Moreover, LOC reached the fastest 10% of its total growth at day 71 approximately, followed by number of improvements, number of fixed bugs and then new features at day 345, 467 and 478 respectively.

Upon these findings, we concluded that the initial development of this release describes characteristics of [I], and the last part characteristics of [E], as defined by the ASMF.

### B. Release 2.1 (Jun-01-2005 to Oct-08-2009, 1591 Days)

LOC and number of improvements remained almost stable. As expected, number of fixed bugs and new features grew continuously, although to a slower rate. The former reached 90% of its total growth almost at the end of this period, approximately at day 1549, and the latter at day 1339 approximately.

The ceilings of number of fixed bugs and new features were still higher than 100%, for which we expected an immediate growth of these two factors after day 1591.

Because almost no new features were implemented (1.57%, 4 in 261 days) and number of total fixed bugs and improvements did not increment much either (8.73% and 4.17%), we

concluded that the development of this release describes mostly characteristics of [S].

### C. Release 2.2 (Jun-01-2005 to Oct-04-2010, 1952 Days)

As expected, number of fixed bugs and new features kept growing, yet to a slow rate. Number of fixed bugs incremented the most, 8.4%, followed by new features with an increment of 5.2%.

In general, all curves showed signs of stabilization, including number of fixed bugs, with a reduced ceiling of  $\kappa = 102.11$ . No immediate growth of any of the curves was expected after day 1952.

Similarly to the previous release development, because increments were not high, we concluded that the development of this release also describes mostly characteristics of [S].

### D. Release 2.3 (Jun-01-2005 to Jan-21-2013, 2792 Days)

All curves mainly remained stable, reaching 90% of its total growth before the beginning of the development of this release at days 1621, 1316, and 1178, for number of fixed bugs, new features and improvements respectively. Yet, more number of improvements were performed during the development of this release.

By exploring the data, we noticed that number of improvements started to increase at day 2080, 551 days earlier than number of fixed bugs and new features. As expected, before day 2080, no changes were observed. These sudden increase of all factors towards the end of the period can explain why the  $\kappa$  values of all our curves were below 100%.

Upon these findings, we concluded that the initial development of this release entered into a temporary [P] stage, followed by a new [S] period, again, for the same reasons explained before.

### E. Validity

The obvious threat to validity of this study concerns external validity for which replication of our approach on other systems is required, taking into account different settings, aging, their branches, etc. Concerning construct validity, we rely on the classification of issues by the issue tracking system of Ivy, and in the good practice of developers at indicating the corresponding issue identification number when the code is modified. As for internal validity, our statistical analysis indicates that all of the derived models fit well the data explored in this study.

## V. CONCLUSIONS

We conclude that S-shaped curves can model software evolution of Ivy well. Moreover, applying the proposed method, we could identify successfully different stages of the implementation of Ivy and their transitions, validating in part the ASMF. According to our observations, two characteristics of the development of Ivy did not comply with the ASMF; first, the transition [P]  $\rightarrow$  [S] in R 2.3 is not considered by the model; and, although no implementation of new features is considered during a [S] stage, we observed that during stages that described more characteristics of [S] than [E], some new

TABLE II: S-shaped Curves' Parameters of Ivy

Statistic	LOC	Fixed Bugs	New Features	Improvements
R 2.0 (1330 Days)				
Midpoint	686.3495	1166.952	955.283	860.3483
$\Delta t$	1231.323	1398.38	1221.308	765.2002
$\kappa$	115.505	169.892	139.578	111.666
$\alpha$	-2.45	-3.667	-3.437	-4.941
$\beta$	0.004	0.003	0.004	0.006
Maxy	9938	209	251	253
R 2.1 (1591 Days)				
Midpoint	644.807	967.9876	825.3971	840.5341
$\Delta t$	1137.147	1161.862	1026.785	718.2805
$\kappa$	104.208	111.892	110.236	102.263
$\alpha$	-2.492	-3.661	-3.533	-5.142
$\beta$	0.004	0.004	0.004	0.006
Maxy	10355	229	255	264
%Incrementy	4.03	8.73	1.57	4.17
R 2.2 (1952 Days)				
Midpoint	636.053	965.8333	801.6298	838.3393
$\Delta t$	1110.937	1159.69	974.5137	711.6687
$\kappa$	100.83	102.112	100.383	99.498
$\alpha$	-2.516	-3.66	-3.615	-5.177
$\beta$	0.004	0.004	0.005	0.006
Maxy	10561	250	269	270
%Incrementy	1.95	8.4	5.2	2.22
R 2.3 (2792 Days)				
Midpoint	644.19	1000.43	814.4722	858.3211
$\Delta t$	1143.92	1241.02	1001.883	639.403
$\kappa$	96.24	96.831	98.609	96.301
$\alpha$	-2.47	-3.543	-3.572	-5.899
$\beta$	0.004	0.004	0.004	0.007
Maxy	11193	277	280	335
%Incrementy	5.65	9.75	3.93	19.4

features were performed. These observations might suggest an additional transition in the ASMF and redefinition of the [S] stage, which would require more research and experimentation.

## REFERENCES

- [1] T. Østerlie and L. Jaccheri, "A critical review of software engineering research on open source software development," in *Proc. of the 2nd AIS SIGSAND, Gdansk, Poland, 2007*, pp. 12 – 20.
- [2] Ø. Hauge, C. Ayala, and R. Conradi, "Adoption of open source software in software-intensive organizations - a systematic literature review," *Inf. and Softw. Tech.*, vol. 52, no. 11, pp. 1133 – 1154, 2010.
- [3] A. Capiluppi, J. M. González-Barahona, I. Herraiz, and G. Robles, "Adapting the "staged model for software evolution" to free/libre/open source software," in *9th Intl. Workshop on Principles of Softw. Evol.*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 79–82.
- [4] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "How software process automation affects software evolution: a longitudinal empirical analysis: Research articles," *J. Softw. Maint. Evol.*, vol. 19, no. 1, pp. 1–31, Jan. 2007.
- [5] D. G. Feitelson, "Perpetual development: A model of the linux kernel life cycle," *J. of Systems and Softw.*, vol. 85, no. 4, pp. 859 – 875, 2012.
- [6] V. T. Rajlich and K. H. Bennett, "A staged model for the software life cycle," *IEEE Computer*, vol. 33, no. 7, pp. 66–71, Jul. 2000.
- [7] D. Kucharavy and R. D. Guio, "Application of s-shaped curves," *Procedia Engineering*, vol. 9, no. 0, pp. 559 – 572, 2011.
- [8] S. Yamada, M. Ohba, and S. Osaki, "s-shaped software reliability growth models and their applications," *IEEE Transactions on Reliability*, vol. R-33, no. 4, pp. 289–292, 1984.
- [9] P. S. Meyer, J. W. Yung, and J. H. Ausubel, "A primer on logistic growth and substitution: The mathematics of the loglet lab software," *Technological Forecasting and Social Change*, vol. 61, no. 3, pp. 247 – 271, 1999.

# Understanding Schema Evolution as a Basis for Database Reengineering

Maxime Gobert, Jérôme Maes and Anthony Cleve  
 PReCISE Research Center  
 University of Namur  
 Namur, Belgium  
 Email: {gobertm,maesj,acl}@info.fundp.ac.be

Jens Weber  
 Department of Computer Science  
 University of Victoria  
 Victoria, BC, Canada  
 Email: jens@uvic.ca

**Abstract**—Software repositories can provide valuable information for facilitating software reengineering efforts. In recent years, many researchers have started to follow a holistic approach, considering diverse software artifacts and the links existing between them. However, when analyzing data-intensive systems, comparatively little attention has been devoted to the analysis of an important system artifact: the database. Even fewer approaches attempt to uncover facts about the evolution history of database schemas. We have developed a tool-supported method for analyzing and visualizing database schema history. This paper reports early results of applying and validating this method. We discuss our experiences to date and point out several novel research perspectives in this domain.

**Index Terms**—data reengineering, mining software repositories, schema evolution

## I. INTRODUCTION

Understanding the evolution history of a complex software system can significantly aid and inform maintenance and reengineering. Software repositories such as configuration management systems and issue trackers provide opportunities for historical analyses of system evolution. Most research work in this area has concentrated on program code, design and architecture. Fewer studies have focussed on database systems and schemas. This is an unfortunate gap as databases are often at the heart of today’s information systems.

We have developed a tool-supported method for analyzing the evolution history of database schemas in order to improve program comprehension and inform software reengineering efforts. This paper describes our method, its implementation, and its application to a real-world case study.

## II. APPROACH

Historical data about the evolution of database schemas can be helpful in software comprehension, but is rarely considered in current methods. For example, database schemas of long-term projects often contain “deprecated structures” that have been semantically superseded by other structures. Nevertheless, the deprecated structures are kept in order to maintain old data. Identification and documentation of such deprecated structures is important for program comprehension and reengineering. Other uses of historical schema analysis may provide insight into questions such as why some parts of the schema are apparently missing important constraints (such

as foreign key declarations), while other parts comprise such constraints. (Usually, older parts of database schemas are less constrained as database management systems at that time may not have been able to compile and enforce such constraints.)

These two concerns are merely examples on how historical schema analysis may provide important clues in information system reengineering projects. The overall question for our research is thus: *How can we extract, represent and exploit the history of a database schema?*

Our general approach consists of extracting and comparing the successive versions of the database schema from the versioning system, in order to produce the so-called *global historical schema*. The latter is a visual and browsable representation of the database schema evolution over time. It contains all database schema objects (e.g., tables, columns and constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime, which in turn serve as a basis for the visualization of the schema and its further analysis. This historical schema can be queried in order to derive valuable information about the evolution of the database, potentially raising other interesting system-specific questions to investigate.

The global process that we follow to build the historical database schema of a system consists of several steps:

- 1) *SQL code Extraction & Cleaning*: We first extract all the SQL files corresponding to each system version, by exploiting the versioning system used. Those files may then need to be slightly edited, in case the SQL syntax used does not match with the SQL parser considered<sup>1</sup>.
- 2) *Schema Extraction*: We extract the logical schema corresponding to each SQL file obtained so far.
- 3) *Schema comparison*: We compare the successive logical schemas while incrementally building the resulting historical schema.
- 4) *Visualization & Exploitation*: The historical schema can then be visualized and further analyzed, depending of the project-specific needs.

Each historical schema object (e.g., table, column) is annotated with several meta-attributes:

<sup>1</sup>We use the schema extractor of DB-MAIN (<http://www.db-main.eu>)

- *NbVersions*: the total number of versions of the schema where the object can be found.
- *creationSchema*: references the first (oldest) schema version where the object appears.
- *creationDate*: the date the oldest schema version where the object appears (the date of *creationSchema*).
- *lastAppearanceSchema*: the last (most recent) schema version where the object appears.
- *lastAppearanceDate*: the date of the *lastAppearanceSchema*.
- *severalLives*: true when the object has existed, has been removed, before being restored (false otherwise.)
- *isOpen*: is a helper attribute needed for the algorithm, meaning that the object has already a value in *lastAppearanceSchema* and *creationSchema*.

---

**Algorithm 1** Deriving the global historical schema from  $n$  successive schema versions.

---

**Notations.**

- Let  $S_i$  be a database schema version, defined as a set of schema objects (including a set of tables and their respective columns).
- Let  $date(S_i)$  be the release date of schema version  $S_i$ .

**Require:**  $S_1, S_2, \dots, S_n$ :  $n$  successive schema versions.

**Ensure:**  $S_H$ : the corresponding global historical schema.

```

// initializing  $S_H$ 
1:  $S_H \leftarrow S_n$ 
2: for all  $o \in S_H$  do
3:    $lastAppearanceSchema(o) \leftarrow S_n$ 
4:    $lastAppearanceDate(o) \leftarrow date(S_n)$ 
5:    $nbVersions(o) \leftarrow 1$ 
6:    $isOpen(o) \leftarrow true$ 
7: end for
// iterating from the last version to the initial version
8: for all  $i \in \{n-1, \dots, 1\}$  do
9:   // objects  $o$  appearing in more than one version
10:  for all  $o \in S_i \cap S_H$  do
11:     $nbVersions(o) \leftarrow nbVersions(o) + 1$ 
12:    if  $isOpen(o) == false$  then
13:       $severalLives(o) \leftarrow true$ 
14:       $isOpen(o) \leftarrow true$ 
15:    end if
16:  end for
17:  // objects  $o$  deleted before the last version
18:  for all  $o \in S_i \setminus S_H$  do
19:     $S_H \leftarrow S_H \cup o$ 
20:     $lastAppearanceSchema(o) \leftarrow S_i$ 
21:     $lastAppearanceDate(o) \leftarrow date(S_i)$ 
22:     $nbVersions(o) \leftarrow 1$ 
23:  end for
24:  // objects  $o$  created after the initial version
25:  for all  $o \in S_H \setminus S_i$  do
26:    if  $isOpen(o) == true$  then
27:       $creationSchema(o) \leftarrow S_{i+1}$ 
28:       $creationDate(o) \leftarrow date(S_{i+1})$ 
29:       $isOpen(o) \leftarrow false$ 
30:    end if
31:  end for
32: end for
33: // Final step
34: for all  $o \in S_H$  do
35:  if  $isOpen(o) == true$  then
36:     $creationSchema(o) \leftarrow S_1$ 
37:     $creationDate(o) \leftarrow date(S_1)$ 
38:  end if
39: end for

```

---

Algorithm 1 formalizes our procedure for deriving a historical database schema  $S_H$  from  $n$  successive schema versions. This derivation algorithm is based on a pairwise comparison of all those schema versions in reverse chronological order.

The *initialization* step of the algorithm (lines 1-7) consists of considering the most recent schema  $S_n$  (augmented with its meta-attributes) as the initial historical schema  $S_H$ . We then iterate on all the previous schemas in reverse chronological order (lines 8-32), while comparing the current schema  $S_i$  with the current historical schema  $S_H$ . The comparison is made by iterating on each schema object of both schemas. Several situations may occur for a given schema object  $o$ :

- 1)  $o$  belongs to  $S_i$  and belongs to  $S_H$  (i.e.,  $o \in S_i \cap S_H$ ). In this case,  $o$  has appeared in more than one schema version. For such an object, we increment the *nbVersion* meta attribute. If its meta-attribute *isOpen* is *false*, this means that  $o$  has *several lives*: it had been removed after version  $i$  before reappearing later on, and version  $i$  corresponds to the end of (one of) its *previous life*. Therefore, we set its *severalLives* meta-attribute to true.
- 2)  $o$  belongs to  $S_i$  but does not belong to  $S_H$  (i.e.,  $o \in S_i \setminus S_H$ ). In this case,  $o$  has been deleted after version  $i$ , and it is the first time we encounter it. We then add  $o$  to the global historical schema together with its initial meta-attribute values.
- 3)  $o$  belongs to  $S_H$  but does not belong to  $S_i$  (i.e.,  $o \in S_H \setminus S_i$ ). In this case, we can derive that  $o$  has been created in version  $i+1$ . We set its *isOpen* meta-attribute to *false*, in order to be able to identify its previous lives, if any.

The *Final* step is performed once all schema versions have been compared to the current historical schema. This step considers all schema objects of the historical schema for which the *isOpen* meta-attribute is still *true*. All those objects have actually been created in the initial schema version  $S_1$ . One therefore needs to initialize their *creationSchema* and *creationDate* meta-attributes accordingly.

### III. APPLICATION AND VALIDATION

We implemented our algorithm as a Java plugin to DB-MAIN and applied it to a case study of significant complexity: the OSCAR system [13]. OSCAR is an Electronic Medical Record (EMR) system for primary care clinics. One problem we faced was a lack of documentation and the sheer size of the database schema (over 465 tables and many thousands of columns). Moreover, at the time of conducting our study, the database schema of OSCAR contained little information on relationships between tables (foreign keys) and no documentation was available.

We analysed the history of the OSCAR database schema during a period of 9 years and 3 months (21/08/2003-21/11/2012). During this period, a total of 532 different schema versions can be found in the project's GitHub repository. However, we decided to consider only one version per month in our historical analysis. We therefore analysed 112 successive schema versions. Among these 112 versions,

8 schema versions proved to be identical to their previous version. The earliest schema version analyzed (21/08/2003) includes 88 tables, while the latest schema version considered (21/11/2012) comprises 460 tables.

Once the historical schema was derived, we applied a procedure that colourizes each historical schema object, depending on its age and its liveness. Fig. 1 shows a colourized version of the OSCAR historical schema, as derived by our tool, and that can be browsed and queried using DB-MAIN. All schema objects depicted in green constitute the tables and columns that are present in the latest schema version. All red schema objects have been deleted. The colour shade corresponds to the age of the objects. A dark red schema object is a table or column that has been deleted a long time ago. A light red object is an object that has recently been removed from the schema. An object depicted in green corresponds to a column or a table that is still present in the latest schema version. The darker the green, the older the corresponding table or column is, and vice versa. A schema object coloured in orange is a deleted object that had several lives.

The resulting historical schema was effective in helping us to answer important semantic questions about the database schema. For example, we found multiple seemingly unrelated schema structures covering the same semantic issue in the database. In one case, one schema structure revolved around tables entitled “*immunizations*” and “*configimmunization*” while another schema structure revolved around tables entitled “*preventions*” and “*preventionsexi*”. As immunizations are essentially disease preventions, we felt that there should be a relationship between these structures, or that one structure may in fact be deprecated and superseded by another. The historical schema allowed us to conclude the latter and identify the deprecated structure.

We also provide the user with a historical schema querying tool, allowing the extraction of interesting statistics regarding the evolution of the schema of interest. Some of those statistics are given below. Fig. 2 (left) provides an aggregate information about the creation and deletion of tables. One can easily notice that OSCAR tables are rarely removed. The evolution of the schema consists (most of the time) of adding tables, while not replacing or splitting them up. The total number of deleted tables is around 30, and we can again quickly identify the major release time periods. The observation is similar for the ratio of created and deleted columns, as shown in Fig. 2, right. The number of column creations is, indeed, often greater than the number of column deletions. During the last 10 releases, however, the removal of columns becomes more intensive: 867 columns were deleted between schema version 106 and schema version 107.

#### IV. RELATED WORK

Research on software evolution has become popular thanks to Lehman’s laws of software evolution [9]. This has inspired many researchers to validate these laws on open source software systems [7], [5], [14]. Today, it has given rise to an empirical research branch on software repository mining,

investigating a wide variety of topics such as test and production code co-evolution [16], code cloning analysis [6], bug prediction techniques [4], change-proneness and fault-proneness [8], and many more.

While the literature on database schema evolution is very large [12], few authors have systematically *observed* how developers cope with database evolution issues in practice. Existing work in this domain [2], [10] analyzed the evolution of rather *small* database schemas. Curino *et. al* [2] present a study of the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. The authors propose, in addition to a schema evolution statistics extractor, a tool that operates on the differences between subsequent schema versions and semi-automatically extracts the set of possible schema operations that have been applied. For instance, simultaneously deleting an existing column and creating a new column may actually translate the renaming of the existing column. In this study, a period of 4 years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. The latter is rather limited in size: from 17 to 34 tables depending on the schema version considered. The total number of columns in the schema does not exceed 250, whatever the version. Lin *et. al* [10] study the co-evolution of database schemas and application programs in two open-source applications, namely Mozilla and Monotone. The number of schema changes reported is very limited. In Mozilla, 20 table creations and 4 table deletions are reported in a period of 4 years. During 6 years of Monotone history, only 9 tables were created while 8 tables were deleted.

In this paper, we introduce the concept of global historical schema, and propose a scalable tool allowing to derive such a schema from the versioning system, in a browsable and queryable format. The concept of the global historical schema and its coloured visualization is analogous to the approach of visualizing program change sets at the architectural level with UML class diagrams [11], and is inspired from a myriad of other software visualization tools supporting the understanding of large-scale software systems evolution, including CodeCity [15], the Evolution Radar [3] and ExtraVis [1].

#### V. CONCLUSIONS

Analyzing the evolution history of database schemas aids understanding of the current schema structure and informs maintenance and reengineering efforts. The historical schema is useful in answering questions about schema structures, for example as in the case of the “preventions” table structure replacing the “immunizations” structure (see OSCAR case study above), while the aggregate (macroanalysis) allows us to infer general trends, such as the tendency of not deleting schema structures, even if they are superseded. To some degree, the creation of new schema structures to semantically replace existing ones without removing them is similar to the well-known process of “cloning” in program code. Therefore, the OSCAR preventions tables and the immunization tables could be considered *database clones*. However, subtle differences

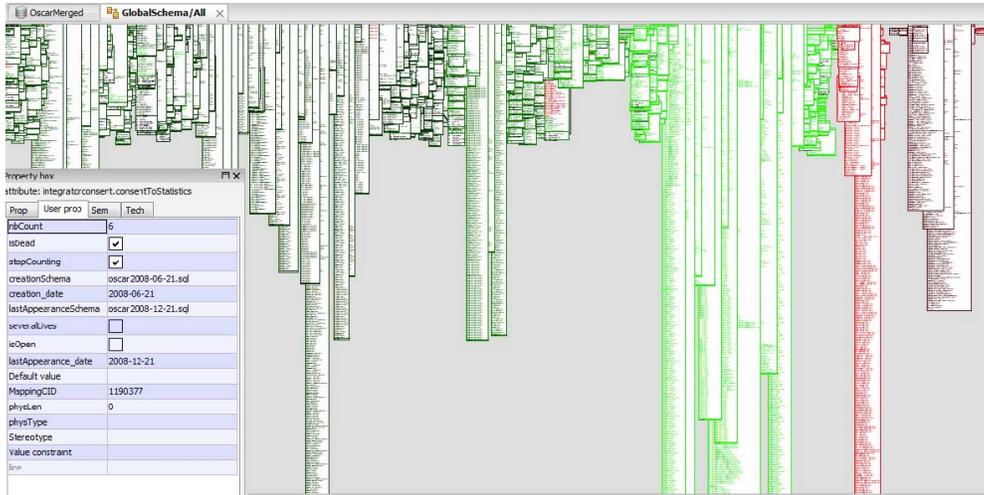


Fig. 1. The OSCAR historical schema, as it can be viewed in DB-MAIN with the property box containing the meta attributes

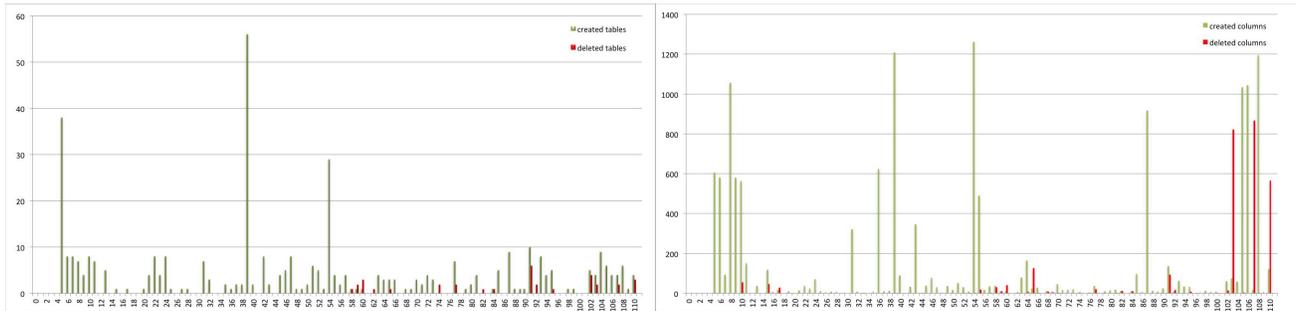


Fig. 2. Intensity of creation and deletion of OSCAR tables (left) and columns (right) over time.

exist to the concept of program clones. For example, program clones are usually still made up of functional code (as opposed to dead code), while the superseded database clone may indeed be considered “dead schema” from the point of view of at least a newer installation of the information system software, i.e., an installation that does not have to deal with legacy data that uses the superseded database structure. Nevertheless, our method may be a first step towards analysing database schema history for the purpose of schema clone detection.

## REFERENCES

- [1] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *TSE*, 37(3):341–355, 2011.
- [2] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In J. Cordeiro and J. Filipe, editors, *ICEIS (1)*, pages 323–332, 2008.
- [3] M. D’Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *TSE*, 35(5):720–735, 2009.
- [4] M. D’Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.
- [5] J. Fernández-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical studies of open source evolution. In *Software Evolution*, pages 263–288. Springer, 2008.
- [6] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of ICSE ’11*, pages 311–320, New York, NY, USA, 2011. ACM.
- [7] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. ICSM’00, ICSM ’00*, pages 131–, IEEE CS, 2000.
- [8] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.
- [9] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, Sept. 1984.
- [10] D.-Y. Lin and I. Neamtii. Collateral evolution of applications and databases. In *Proc. of IWPSE-EVOL’09*, pages 31–40. ACM, 2009.
- [11] A. McNair, D. M. German, and J. Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Proc. of WCRE’07*, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.
- [12] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, Dec. 2006.
- [13] J. Ruttan. *The Architecture of Open Source Applications, Volume II*, chapter OSCAR. Lulu.com, 2012.
- [14] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi. Assessing architectural evolution: a case study. *ESE*, 16(5):623–666, Oct. 2011.
- [15] R. Wetzel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *ICSE*, pages 551–560. ACM, 2011.
- [16] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *ESE*, 16(3):325–364, 2011.

# SAMOA — A Visual Software Analytics Platform for Mobile Applications

Roberto Minelli and Michele Lanza

REVEAL @ Faculty of Informatics — University of Lugano, Switzerland

**Abstract**—Mobile applications, also known as *apps*, are dedicated software systems that run on handheld devices, such as smartphones and tablet computers. The apps business has in a few years turned into a multi-billion dollar market. From a software engineering perspective apps represent a new phenomenon, and there is a need for tools and techniques to analyze apps.

We present SAMOA, a visual web-based software analytics platform for mobile applications. It mines software repositories of apps and uses a set of visualization techniques to present the mined data. We describe SAMOA, detail the analyses it supports, and describe a methodology to understand apps from a structural and historical perspective.

The website of SAMOA, containing the screencast of the tool demo, is located at <http://samoa.inf.usi.ch/about>

## I. INTRODUCTION

Mobile applications, or *apps*, are custom software systems running on handheld devices, *i.e.*, smartphones and tablet PCs. The world of apps is variegated: Each vendor imposes a number of constraints (*e.g.*, the programming language and development environment to be used), provides specific design guidelines, and offers its own distribution channel (*e.g.*, Android’s Google Play, Apple’s App Store). The market of apps is remarkable: Apps generated a revenue of \$4.5 billion USD in 2009 [1], and the business is expected to be worth \$25 billion USD [2] a few years from now.

The Apple and Google stores provide ca. one million apps for download. With their increasing popularity, apps are becoming an important software engineering domain. Apps represent a new phenomenon but, as any software system, they will inevitably face evolution, maintenance, and comprehension problems. It is unclear whether existing approaches for program comprehension and maintenance [3], [4], [5] can be ported to apps, since they were devised before apps existed.

We devised a novel approach to analyze apps [6] and implemented SAMOA, a web-based software analytics platform for apps<sup>1</sup>. SAMOA mines software repositories of apps and uses a set of visualization techniques to present the mined data. SAMOA offers a catalogue of custom views to understand the structure and evolution of apps. Both analysts and developers interested in comprehending apps can benefit from these visualizations.

We used SAMOA to investigate part of the F-Droid repository<sup>2</sup>. We discovered, for example, that inheritance is essentially unused in apps, that apps heavily rely on 3<sup>rd</sup>-party APIs, and that most apps are short-lived single developer projects.

<sup>1</sup>See <http://samoa.inf.usi.ch>

<sup>2</sup>See <http://f-droid.org>

**Related work.** Since the first apps were developed only a few years ago, there is little directly related work. Ruiz *et al.* focused on software design aspects of apps, namely on reuse by inheritance and class reuse [7]. They divided apps in categories (*e.g.*, casino, personalization, photography), and found that more than 60% of all classes in each category appear in more than two other apps. Hundreds of apps were entirely reused by other apps in the same category. Harman *et al.* introduced “App Store Mining” [8], a novel form of software repository mining. They mined the Blackberry app store and studied a number of correlations between different features of apps.

Differently from Harman *et al.* we want to focus on the source code of apps, rather than on app stores. Our goal is to understand the differences between apps and traditional software systems, and the implications for the maintenance and comprehension of apps. The novelty of apps explains the small amount of related work, but also calls for novel tools and techniques to analyze apps.

We present SAMOA, our visual web-based software analytics platform for mobile applications. SAMOA leverages three factors for the analysis: source code, usage of 3<sup>rd</sup>-party libraries, and historical data. SAMOA presents the data to the user by means of a catalogue of interactive visualizations. The views are enriched with traditional software metrics complemented by domain-specific ones. SAMOA provides a custom snapshot view to depict a specific revision of one app, a evolution view to present historical aspects of one app, and ecosystem views to depict several apps at once.

## II. SAMOA: A VISUAL SOFTWARE ANALYTICS PLATFORM FOR APPS

Figure 1 depicts the main user interface of SAMOA presenting a snapshot view of the ALOGCAT application. The UI is composed of five parts: a (1) *Selection panel* that allows the user to pick the app to be analyzed, and to switch between the different visualizations SAMOA provides; a (2) *Metrics panel* which summarizes a set of metrics in sync with the visualization, being a specific revision of an app (*i.e.*, snapshot) or global measurements about the apps ecosystem; a (3) *Revision info panel* that displays information about a specific revision of an app; an (4) *Entity panel* displaying additional details about the entity in focus; and the (5) *Main view*, the remaining surface dedicated to the interactive views.

Figure 1 illustrates also how we enhance the view using colors and metrics.

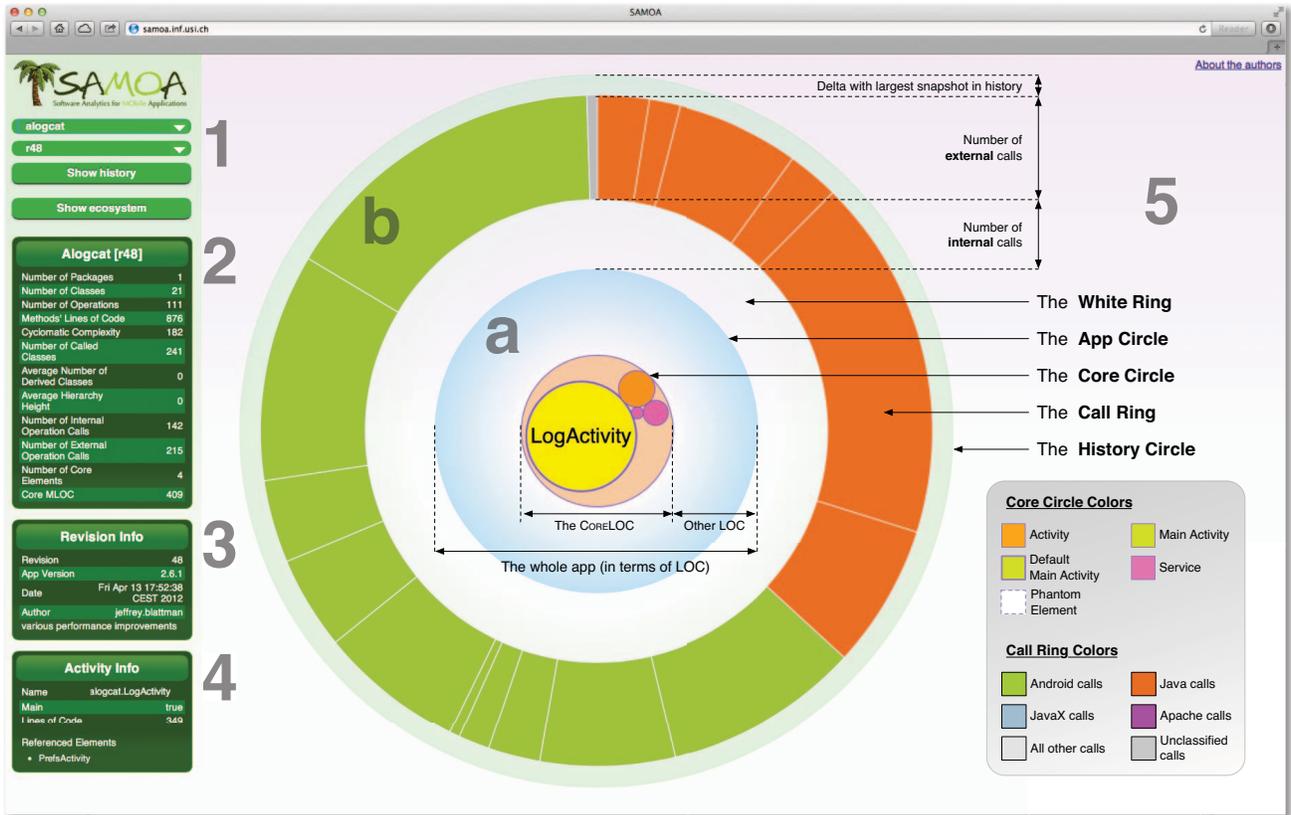


Fig. 1: SAMOA: our web-based software analytics platform for mobile applications, depicting a snapshot of ALOGCAT.

### A. Visualizations

To better comprehend mobile applications SAMOA provides visualizations at three different granularities: The “*snapshot view*” depicts a specific revision of an app; “*evolution views*” present the evolution of an app over its history; and “*ecosystem views*” depict more than one app at once.

a) *Snapshot view*: Figure 1 shows the 48<sup>th</sup> revision of the ALOGCAT app<sup>3</sup>. This view presents the most important structural properties of an app at hand. Two main parts compose our “*snapshot view*”: the central section (1.a) and a ring (1.b).

The central section of the view presents the entire app in terms of classes and lines of code (LOC). Among all classes, we define as “*Core Elements*” the entities specific to the development of apps (*i.e.*, which inherit from the mobile platform SDK’s base classes). This section is composed of an “*App Circle*” (*i.e.*, shaded blue) and a “*Core Circle*” (*i.e.*, light red). The former represents the entire app in terms of LOC, while the latter depicts the core of the app. Circles inside the “*Core Circle*” are “*Core Elements*” (*i.e.*, Java classes), where the radius of each circle is proportional to the value of LOC of the entity represented, and color & stroke provide additional information about the type of the entity (*e.g.*, Activity, Service, Main Activity).

<sup>3</sup>See <http://code.google.com/p/alogcat/>

The radius of the “*App Circle*” is proportional to the number of LOC of the app at the current revision while the radius of the “*Core Circle*” is proportional to the sum of the values of LOC of “*Core Elements*”.

The “*Call Ring*” (Figure 1.b) uses a circular layout to depict the 3<sup>rd</sup>-party APIs calls the app makes. Its thickness and total span are proportional to the number of external method calls. Each slice of the “*Call Ring*” represents calls to a distinct 3<sup>rd</sup>-party library (*e.g.*, Apache, JavaX), where colors distinguish calls to different libraries. We use specific colors to depict calls to the four most employed libraries and two tones of gray: one for all calls to other libraries and one for the calls SAMOA is not able to identify. The angle spanned by an arc is proportional to the number of method calls it represents. The “*Historical Circle*” (*i.e.*, green shaded) represents the maximum size of an app over its entire history, thus if there is a gap (as in Figure 1) we know that the currently visualized snapshot is not the largest in the history of the app. The number of internal calls (*i.e.*, calls implementing internal behavior of the app) is represented by the thickness of the “*White Ring*”. With this visual cue, we can infer the ratio between internal and external calls. The outer radius of the “*Call Ring*” indicates the size of a snapshot, considering both LOC (*i.e.*, the radius of “*App Circle*”) and method calls (*i.e.*, thickness of the “*Call Ring*” and the “*White Ring*”).

b) *Evolution view*: SAMOA uses stacked bar charts and line charts to present different types of evolutionary information (e.g., LOC, external calls, or core elements) about an app, considering all the snapshots available to SAMOA.

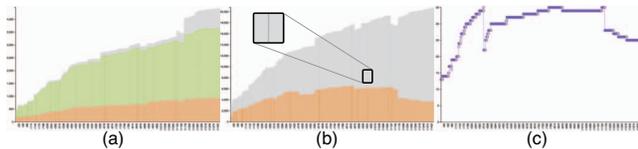


Fig. 2: Evolution views of the CSIPSIMPLE app in terms of (a) 3<sup>rd</sup>-party calls, (b) LOC, and (c) number of “Core Elements”.

Figure 2 depicts the *evolution views* of the CSIPSIMPLE app<sup>4</sup>, in terms of (a) 3<sup>rd</sup>-party calls, (b) LOC, and (c) number of “Core Elements”. In the bar chart views, each bar represents a snapshot of the app, divided into layers, according to the type of data presented, e.g., Figure 2.b depicts the evolution LOC, thus layers are CORELOC (i.e., red) and non-CORELOC (i.e., grey). The height of each bar represents the value of a specific software metric, in this example the value of LOC. As highlighted in Figure 2.b, we use opacity to denote an app’s release versions: Darker bars are snapshots whose release number changed. SAMOA uses line charts to depict data without a logical layer subdivision, e.g., the evolution of the number of “Core Elements” presented in Figure 2.c.

c) *Ecosystem view*: *Ecosystem views* depicts several apps at once, using stacked bar charts or a grid layout. Figure 3 depicts an ecosystem view of 12 apps, sorted according to their size (i.e., in terms of LOC), arranged using a grid layout.

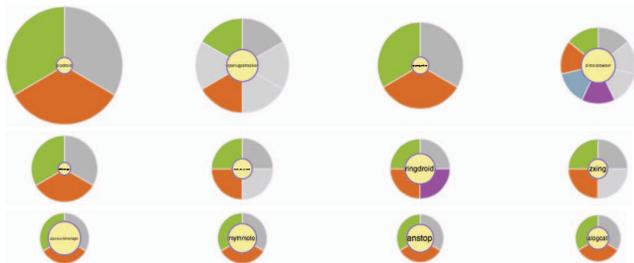


Fig. 3: An ecosystem view of 12 apps, sorted by total LOC.

Each shape is a simplified snapshot view of an app where the radius of the core (i.e., yellow) corresponds to the number of CORELOC. The radius is proportional to the total number of LOC, the span of the “Call Ring” shows the proportions of external calls (either with unary or proportional weights).

### B. User Interactions

All the visualizations offered by SAMOA are interactive. For example, by hovering on a shape, the entity is highlighted and the “entity panels” of SAMOA provides additional information about the shape in focus (see Figure 1.4). The user can freely zoom and pan the snapshot view. On clicking on a core element,

<sup>4</sup>See <http://code.google.com/p/csipsimple/>

SAMOA displays its source code. In the visualizations based on bar charts, the data can be re-ordered and the user can choose to display layers either grouped or stacked (i.e., default). In both the evolution and ecosystem view clicking on a shape leads to the corresponding snapshot view of the app.

### C. A Methodology to Understand Apps

SAMOA provides views at different granularities, with different purposes and applications, which we described in previous work [6]. The typical methodology is to use ecosystem views to get a “big picture” of several apps at once, and then drill down using the evolution view which, in turn, help us to understand where to use the snapshot view.

### D. Under the Hood: Architecture and Technologies

SAMOA is composed of a back-end and a front-end, as Figure 4 depicts. The back-end, entirely written in Java, is responsible for a number of tasks: (1) it mines software repositories of apps and extracts apps-specific data from different artifacts. Then (2) it processes the data by, extracting and parsing two different source code representations, namely the AST (Abstract Syntax Tree) and the MSE<sup>5</sup>. From these two representations, plus the Android manifest<sup>6</sup>, SAMOA (3) extracts a set of software metrics and (4) generates JSON files that are served to the front-end, implemented using PHP, HTML5, and JavaScript (i.e., views are generated using d3.js).

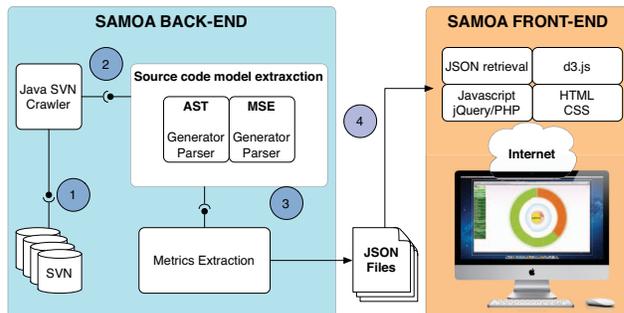


Fig. 4: An architectural overview of SAMOA.

## III. A CATALOGUE OF PECULIARITIES OF APPS

Following the methodology described in Section II-C, we devised a catalogue of peculiarities of apps [6]. For example, we can order our ecosystem of apps according to their number of revisions, and investigate on the app with the longest history.

**Example I:** Figure 5.a shows part the evolution of LOC of ZXING<sup>7</sup>, the app with the longest history in our apps ecosystem, with more than 2.2k commits. Android apps have a configuration file (i.e., called manifest) which identifies the “Core Elements” of the app. To work properly, an app requires the manifest file to be in sync with the source code.

<sup>5</sup>See <http://www.moosetechnology.org/docs/mse>

<sup>6</sup>See <http://goo.gl/Rt6GD>

<sup>7</sup>See <http://code.google.com/p/zxing/>

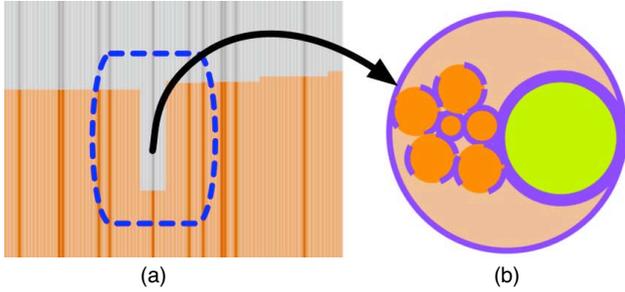


Fig. 5: (a) Part of the evolution of LOC of the ZXING app and (b) one of the snapshot in which the manifest is out of sync.

Android developers should maintain this file in sync manually (*i.e.*, reflecting the changes performed in the source code to the manifest file). During our app analysis we discovered that sometimes developers forgot to keep the manifest updated, introducing bugs in their apps, as in the case of ZXING. In the highlight of Figure 5, there is a time interval in which CORELOC drop and suddenly they increase again. With further investigation, we discovered that the authors have moved some functionalities into sub-packages, but they forgot to update the references in the manifest, preventing both the Android OS and SAMOA to recover links to the “Core Elements”. Figure 5.b shows that SAMOA is not able to correctly recover “Core Elements”, and depicts some of them as “phantom elements.”

**Example II:** Apps should conform to a set of sound guiding principle. The Android documentation, for example, recommends that apps should have one “Main Activity” and, if not so, they must have a single “Default Main Activity”: The real main activity to invoke. In our app analysis, we observed many apps have more than one main activity.

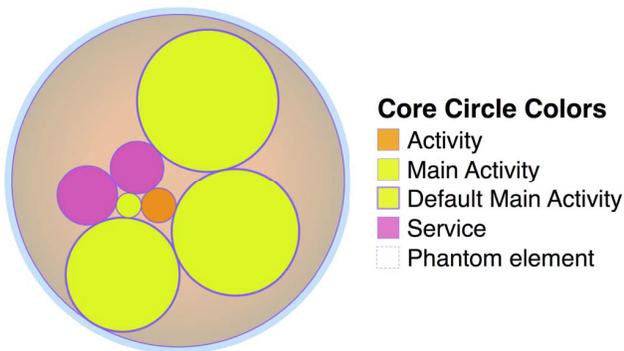


Fig. 6: The central section of the snapshot view of APP-SOUNDMANAGER app at revision 106.

Figure 6 depicts the 106<sup>th</sup> snapshot of APP-SOUNDMANAGER<sup>8</sup>. This apps lists 4 Main Activities (*i.e.*, yellow), out of which 3 “default” main activities (*i.e.*, thicker stroke in our snapshot view), violating the said Android guideline.

<sup>8</sup>See <http://code.google.com/p/app-soundmanager/>

**Example III:** In all Java systems, including Android apps, 3<sup>rd</sup>-party APIs are reused by including JAR files in the build path. Developers of apps have a tendency of directly importing the entire source code of 3<sup>rd</sup>-party libraries instead of adding the needed JAR files, which is a questionable practice from a legal point of view.

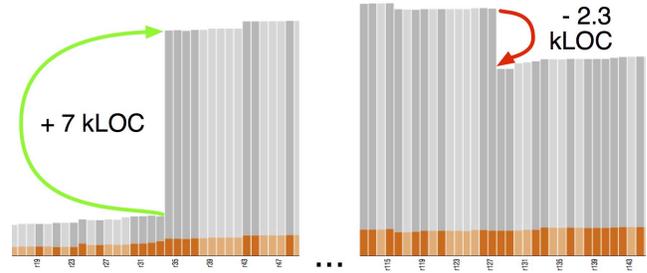


Fig. 7: Part of the evolution of LOC of APPS ORGANIZER.

Figure 7 depicts the evolution of the number of LOC of the APPS ORGANIZER<sup>9</sup> application. At some point the authors added the source code ( $\approx 7$  kLOC) of the Trove library<sup>10</sup>. Later on, they removed part of the same library ( $\approx 2.3$  kLOC).

#### IV. CONCLUSION

We presented SAMOA, a novel web-based software analytics platform, which supports our approach [6] to analyze apps from several points of view, using custom views tailored to the novel domain of mobile applications, offering several means to navigate and inspect information.

**Acknowledgements.** We gratefully acknowledge the Swiss National Science foundation’s support for the project “HI-SEA” (SNF Project No. 146734).

#### REFERENCES

- [1] R. Islam, R. Islam, and T. Mazumder, “Mobile application and its global impact,” *IJEST*, 2010.
- [2] Markets and Markets, “Global mobile application market (2010–2015),” 2010.
- [3] M. Lehman, D. Perry, and J. Ramil, “Implications of evolution metrics on software maintenance,” in *Proceedings of ICSM*, 1998, p. 208.
- [4] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, “Software evolution observations based on product release history,” in *Proceedings of ICSM*, 1997, pp. 160–166.
- [5] W. Turski, “The reference model for smooth growth of software systems revisited,” *TSE*, pp. 814–815, 2002.
- [6] R. Minelli and M. Lanza, “Software Analytics for Mobile Applications - Insights & Lessons Learned,” in *Proceedings of CSMR*, 2013, pp. 144–153.
- [7] I. Ruiz, M. Nagappan, B. Adams, and A. Hassan, “Understanding reuse in the android market,” *ACM-ICPC*, 2012.
- [8] M. Harman, Y. Jia, and Y. Zhang, “App store mining and analysis: MSR for app stores,” in *Proceedings of MSR*, 2012.

<sup>9</sup>See <http://code.google.com/p/appsorganizer/>

<sup>10</sup>See <http://trove.starlight-systems.com/>

# Towards A Scalable Cloud Platform for Search-Based Probabilistic Testing

Louis M. Rose\*, Simon Poulding\*, Robert Feldt† and Richard F. Paige\*

\*Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK  
{louis.rose,simon.poulding,richard.paige}@york.ac.uk

†School of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden  
robert.feldt@bth.se

**Abstract**—Probabilistic testing techniques that sample input data at random from a probability distribution can be more effective at detecting faults than deterministic techniques. However, if overly large (and therefore expensive) test sets are to be avoided, the probability distribution from which the input data is sampled must be optimised to the particular software-under-test. Such an optimisation process is often resource-intensive. In this paper, we present a prototypical cloud platform—and architecture—that permits the optimisation of such probability distributions in a scalable, distributed and robust manner, and thereby enables cost-effective probabilistic testing.

## I. INTRODUCTION

The probabilistic generation of test inputs by random sampling from an input profile (i.e. a probability distribution over the input domain of the software-under-test) has two significant advantages over deterministic test inputs. Firstly, if the software is repeatedly tested using the same deterministic test set, it can become ‘overfitted’ to the tests: while the software may operate correctly for the specific inputs specified by the deterministic test set, it may still exhibit faults when run using any other input. Using randomly-generated inputs reduces such overfitting. Secondly, it is relatively straightforward to construct a test set of any given size using a probabilistic approach—additional test inputs may be generated simply by taking further samples from the input profile—and so adapt the amount of testing to the time and budget available.

However, probabilistic test generation using an *arbitrary* input profile, such as a uniform distribution over the input domain, is unlikely to be cost-effective. For example, if a component of the software is only exercised when the software is executed with inputs from a small region of the input domain, test inputs sampled from a uniform distribution will rarely exercise that component. Therefore a large set may be required to detect faults in that component, but large test sets are not cost-effective: each test case requires the application of an ‘oracle’ to check the correctness of the observed output from the software and many oracles, such as a test engineer interpreting a specification document, are expensive. Instead, the input profile must be optimised to the specific software-under-test if probabilistic testing is to be cost-effective.

Poulding [1] has previously demonstrated an automated algorithm for optimising profiles in this way that uses a criterion developed by Thévenod-Fosse and Waeselynck [2] based on structural coverage: maximise the probability that any given part (e.g. statement or branch) of the software-under-test is exercised by one test input sampled at random from the

profile. Poulding’s algorithm uses a metaheuristic optimisation (or ‘search’) method: iterative hill-climbing. At each iteration, small changes are made to the current input profile to create a number of candidate profiles, the candidate profiles are evaluated against Thévenod-Fosse and Waeselynck’s criterion to determine their ‘fitness’, and the fittest of these candidates becomes the current input profile in the next iteration.

Since each candidate profile is a probability distribution, it is evaluated by executing the software-under-test with multiple inputs sampled from the distribution. These executions are not explicitly part of the testing processes, and the (potentially expensive) oracle is *not* applied to the results; only after the input profile has been optimised by the algorithm is a small test set generated from the profile and the oracle applied to the results from this test set. Nevertheless, Poulding’s search-based algorithm can be resource-intensive and to ensure practicality of this testing strategy, it is beneficial to derive an optimal input profile as quickly as possible.

It is this challenge that we address in this paper. We present a prototypical implementation of a cloud platform that optimises input profiles for probabilistic testing quickly and robustly, with the objective of scaling the testing approach to larger software. The paper makes the following contributions:

- A probabilistic testing platform that derives optimised input profiles, available at <http://coco.herokuapp.com>.
- The design of a service-oriented architecture for integrating a testing platform with the software-under-test in a robust and scalable manner.
- An evaluation of the platform with a realistic case study: testing a model transformation used in specifying the behaviour of a robot.

## II. DESIGN AND IMPLEMENTATION

We have designed and implemented a probabilistic testing platform, *coco*, that can quickly optimise input profiles. Our platform employs a service-oriented architecture to achieve scalability and robustness. Figure 1 outlines our approach. A user of the platform provides a URL for the software that they wish to test. The testing platform requests, via the user-provided URL, an initial input profile from a testing harness that wraps the software-under-test (SUT), and then proceeds to derive an optimised input profile by application of Poulding’s automated search-based algorithm. The process of evaluating the fitness of a candidate profile involves executing the SUT

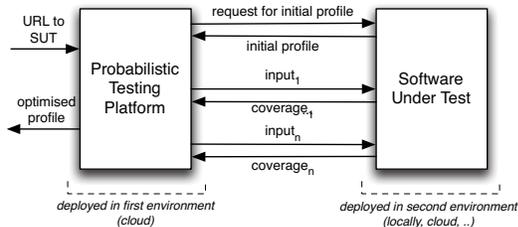


Fig. 1. An overview of our proposed architecture. Boxes indicate software components and arrows indicate communication via HTTP.

using multiple inputs sampled from the profile. The SUT is instrumented to determine which parts of the software are exercised by each input, and this coverage data is returned to the testing platform in order to calculate the candidate profile’s fitness.

Key to our architecture (Figure 1) is the decoupling of the testing platform from the SUT via HTTP, which provides several benefits. The testing platform and SUT can be implemented in different programming languages—allowing us to support the testing of programs written in arbitrary languages—and in separate environments (e.g. different machines)—allowing us to tune and scale each environment independently. In addition, the testing platform is more robust with respect to failures that arise in the SUT: candidate evaluation can be re-tried or aborted if the SUT fails to respond.

Further decoupling at the level of data representation is achieved through the use of a stochastic grammar to represent a distribution over the (potentially highly-complex) input domain of the SUT; this technique is described in [3]. The SUT wrapper specifies this grammar as the initial profile. The testing platform provides strings of tokens sampled from candidate profile grammars that are interpreted by the wrapper as inputs to the SUT. Consequently, the testing platform and SUT do not need to share the same data representation.

Internally, our testing platform comprises three components that communicate via message queues (Figure 2). The *HTML GUI* allows users of the platform to start new probabilistic testing tasks, to view the status of existing tasks, and to download the optimised input profile from which they can sample random test data. The *Searcher* encapsulates Poulding’s automated search-based algorithm and, during the optimisation process, identifies candidate input profiles to evaluate. The *Evaluator* encapsulates the logic required to communicate with the SUT to make this evaluation.

The *HTML GUI*, *Searcher* and *Evaluator* components are decoupled via message queues and a shared database. For example, when the *Searcher* requires an evaluation to be performed it creates a record for that evaluation in the shared database, places a request for evaluation work on the message queueing system and polls the database to determine when the evaluation has completed. Meanwhile, the *Evaluator* dequeues requests for evaluation work, interacts with the SUT, and updates the evaluation record in the database with the coverage data obtained from the SUT.

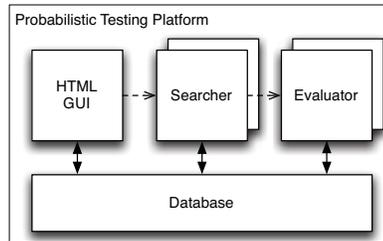


Fig. 2. The architecture of our testing platform. Boxes indicate software components, which communicate via message queues (dashed arrows) and with a database via HTTP (solid arrows). In the configuration depicted above, we have scaled out search and evaluation by deploying additional instances of those components.

Decoupling the components of the testing platform via message queues and a shared database provides benefits that are analogous to those described for the architecture as a whole: the components can be implemented in different languages, can be deployed in separate environments, and can be scaled out independently by instantiating additional instances of a component. Additionally, the use of message queuing increases the robustness of the testing platform as whole, because (search and evaluation) jobs that fail can be re-queued and re-tried.

Our testing platform has been implemented in Ruby and uses a number of off-the-shelf libraries. The *HTML GUI* uses the Ruby on Rails web framework and the *Evaluator* uses the HTTParty HTTP client for Ruby. The *Searcher* component is written in JRuby<sup>1</sup> and Java. Message queuing is implemented with the Resque library. We selected Ruby and Ruby libraries due to our familiarity with deploying and scaling out Ruby applications on the Heroku cloud platform. For the *Searcher* component, we selected JRuby to allow us to more readily experiment with alternative Java libraries for metaheuristic search algorithms (such as ECJ<sup>2</sup> and JMetal<sup>3</sup>) in future work.

### III. EXAMPLE AND EXPERIENCE REPORT

We report here our experiences of applying our probabilistic testing platform to derive an optimised input profile for a program written in a model transformation language (introduced below), and reflect on the benefits and drawbacks of our platform compared to the ad hoc and naïve architecture used in our previous work on the same program [4]. We present empirical evidence to demonstrate performance improvements over the testing performed in our previous work, and also reflect on our experiences with using the probabilistic testing platform for the preliminary experimentation presented below.

Model transformations are programs that consume a source model and produce a target model. The models on which model transformations operate must conform to a modelling language (metamodel) that specifies the structures from which valid instance models can be constructed. For example, UML is a metamodel, a class diagram is a model, and a program that consumes a class diagram and produces a database schema is a

<sup>1</sup>An implementation of Ruby that runs on the Java Virtual Machine

<sup>2</sup><http://cs.gmu.edu/~eclab/projects/ecj/>

<sup>3</sup><http://jmetal.sourceforge.net>

model transformation. Model transformation is key to model-driven engineering and is supported by model transformation languages such as XSLT [5], Query/View/Transformation [6] and the Atlas Transformation Language [7]. As Mottu et al. discuss in [8], when the output of the SUT is a model—as it is for a model transformation—applying an oracle can be particularly difficult: not only is the output itself complex in nature, but the observed and predicted models may be syntactically different even when they are semantically equivalent. There is, therefore, a particular motivation to use probabilistic testing to derive an *optimised* input profile that minimises the size of the test set required to exercise the model transformation and hence the costs involved in applying the oracle.

### A. The Software-under-Test

Our previous work applies probabilistic testing to a model transformation for a Lego Mindstorms robot [4]. For this transformation, the source and target models conform to domain-specific modelling languages that describe a high-level plan for the robot and a low-level strategy for executing the plan, respectively. The transformation automatically derives a strategy from a plan. Previously, we were able to derive an optimised input profile but experienced several issues with the ad hoc and naïve architecture we used at the time:

- Deriving the optimised input profile took almost 17 hours, averaging approximately 0.3 seconds per execution of the model transformation (which equates to 240,000 executions in total).
- For each concurrent instance of the technique that we ran for the empirical work, we had to provision dedicated (virtual) machines for the search-based algorithm and for the SUT. Provisioning the machines was a significant infrastructural overhead, and limited the extent to which we could conduct our empirical work.
- The search-based algorithm did not store its state in a persistent store. Consequently, failures on the machines running the algorithm were catastrophic, and destroyed any progress made towards deriving an optimised input profile.
- The search-based algorithm and SUT were tightly coupled, which was problematic when, for instance, the performance of the SUT degraded because it was impossible to re-provision the SUT without aborting the search. Performance degradation of the SUT is a particular risk in applying our approach because of the random nature of the input data, which might be very different to the kinds of input against which the software has previously been tuned.

The Lego Mindstorms model transformation was implemented in the Epsilon Transformation Language [9], a model transformation language written in Java. In our previous work, we constructed a light-weight adapter for the Lego Mindstorms transformation that facilitated probabilistic testing. The wrapper allowed us to deploy the Lego Mindstorms transformation as a web application on the Heroku cloud platform. Scalability was achieved by creating multiple copies of the application,

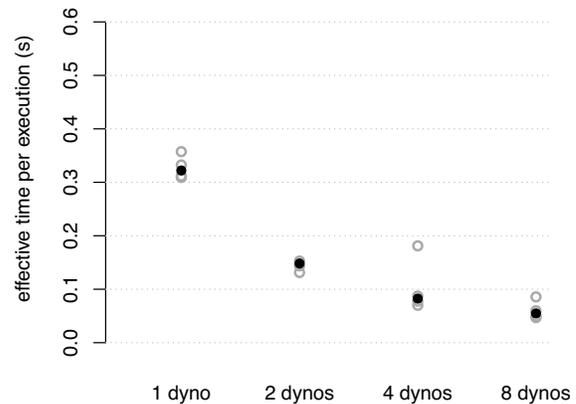


Fig. 3. Scatter graph showing the change in performance with the number of dynos. The open circles are the individual performance measurements, and the filled circles are the median performance of each configuration.

and having each instance of the search-based algorithm address a unique copy of the transformation application.

For this paper, we adapted the architecture presented in Figure 2 for the testing platform to provide a similarly scalable application for executing the Lego Mindstorms (or other similar) model transformation. This application fulfils the role of the wrapper of the software-under-test in the architecture of Figure 1.

### B. Empirical Evaluation

The empirical evaluation focuses on the major objective of our proposed testing platform: that of improved performance through horizontal scaling.

Both the testing platform and transformation application (the software-under-test) were deployed using the Heroku cloud platform. The testing platform was run in four different configurations which made available 1, 2, 4, and 8 dynos (Heroku virtual machines). Each dyno supports one *Evaluator* instance, and so the 2 dyno configuration, for example, could run two *Evaluators* concurrently during the search. In each configuration, the equivalent number of processing dynos were available to the transformation application to support the increased load from the *Evaluators*. The time taken for 5 iterations of the search algorithm was measured (a total of 7550 executions of the model transformation by inputs sampled from candidate profiles), and this time divided by 7550 to calculate an effective time per execution. In order to reduce the impact of wide variation in performance as a result of the time-shared Heroku platform (see below), the performance of each configuration was measured 4 times.

The results are shown in Figure 3. We note that the median performance (filled circle) for the 1 dyno configuration is similar to the 0.3 seconds achieved by the ad hoc architecture used in our previous work; this is to be expected given that the 1 dyno configuration employs similar computing resources to that previous architecture. However, as the number of dynos is increased, the algorithm performance improves. The median performance achieved with 8 dynos is 0.055 seconds

per execution: over 5 times faster than using the architecture of our previous work. This result illustrates the performance improvements that are possible due to the horizontal scaling of our testing platform’s architecture.

### C. Reflective Evaluation

In addition to the performance improvements described in the previous section, we now reflect on additional, unexpected benefits that we have experienced in applying our probabilistic testing platform to test the Lego Mindstorms transformation described above.

In performing our experimentation we observed non-trivial variation in execution time both for the testing platform and the software under test: see, for example, the very slow outlier for the 2 dyno configuration in Figure 3. We attribute this to deployment on the Heroku cloud platform which implements a time-sharing policy: our applications execute on a machine shared with the applications of other users. When other applications deployed on the same machine experience high load, our applications are likely to receive less CPU time. Using message queuing allows us to potentially distribute load over a number of threads deployed on different machines, reducing the overall chance of our applications being affected by contention for CPU time on a single machine.

Furthermore, the architecture that we propose in this paper is amenable to dynamic scaling in which the number of threads allocated to a task (e.g. evaluating candidate input profiles) can be automatically increased in response to a backlog. By monitoring the queue size, we can approximate the load on the system and instantiate additional—or re-allocate existing—computational resources to reduce bottlenecks.

Finally, a downside of the architecture that we proposed in this paper is that the asynchronous interface between the testing platform and the software under test is implemented using a simplistic polling mechanism which introduces additional load on the software-under-test (which must provide information about the status of executions) and increases latency (as there is often a delay between an execution completing and the testing platform sending a request to obtain the results). We will investigate introducing a callback mechanism to address these issues, and further reduce the time taken to derive an optimal input profile.

## IV. RELATED WORK

We briefly describe examples of other testing tools that have employed cloud architectures in order to scale automated algorithms to real-world software. Oriol and Ullah [10] describe a cloud-deployed version of the automated random testing tool YETI<sup>4</sup> motivated by the need to accommodate software-under-test that executes relatively slowly, and to isolate potentially damaging executions of the software from the main testing mechanism. Di Geronimo et al. [11] propose a search-based technique for generating JUnit test cases using a genetic algorithm parallelised using Hadoop MapReduce; the motivation is to enable the technique to leverage high-performance parallel computing environments such as cloud

computing and GPU cards. We note that in common with the search-based technique of Di Geronimo et al., our use of a cloud architecture is for the *generation* of test cases, in contrast to YETI where the cloud resources are applied to the large-scale *execution* of test cases.

## V. CONCLUSIONS

Poulding’s probabilistic testing algorithm, like many other techniques for automated testing, is a promising—but computationally expensive—approach. We argue that scalable, distributed and robust architectures are crucial for cost-effective application of automated testing techniques, particularly those based on search algorithms. We have presented a preliminary version of our probabilistic testing platform, described its underlying service-oriented architecture, and evaluated the platform by applying it to test a model transformation. Our initial experience with the platform and its architecture indicate that is more scalable and robust than our previous ad hoc and naïve implementation. In future work, we will seek to also demonstrate the generality of our platform by applying it to test further examples of model transformations, and to test programs written in other programming languages.

## ACKNOWLEDGEMENT

This work was funded in part by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering.

## REFERENCES

- [1] S. Poulding and J. A. Clark, “Efficient software verification: Statistical testing using automated search,” *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 763–777, 2010.
- [2] P. Thévenod-Fosse and H. Waeselynck, “An investigation of statistical software testing,” *J. Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5–26, 1991.
- [3] S. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, “The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing,” in *Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, 2013, (to appear).
- [4] L. Rose and S. Poulding, “Efficient probabilistic testing of model transformations using search,” in *Proc. CMSBSE workshop, co-located with Int’l Conf. Software Engineering*. IEEE / ACM, 2013, (in press).
- [5] W3C, “XSL Transformations (XSLT) Specification V1.0 [online],” [Accessed 21 June 2013] Available at: <http://www.w3.org/TR/xslt>, 1999.
- [6] OMG, “Query/View/Transformation Specification V1.1 [online],” [Accessed 21 June 2013] Available at: <http://www.omg.org/spec/QVT/1.1/>, 2011.
- [7] F. Jouault and I. Kurtev, “Transforming models with ATL,” in *Proc. Satellite Events at the Int’l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed., vol. 3844. Springer, 2005, pp. 128–138.
- [8] J.-M. Mottu, B. Baudry, and Y. Traon, “Model transformation testing: oracle issue,” in *Proc. Workshops at Int’l Conf. Software Testing Verification and Validation*, April 2008, pp. 105–112.
- [9] D. Kolovos, R. Paige, and F. Polack, “The Epsilon Transformation Language,” in *Proc. ICMT*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.
- [10] M. Oriol and F. Ullah, “YETI on the cloud,” in *Proc. Workshops of the Int’l Conf. on Software Testing, Verification, and Validation*, 2010, pp. 434–437.
- [11] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, “A parallel genetic algorithm based on Hadoop MapReduce for the automatic generation of JUnit test suites,” in *Proc. Int’l Conf. Software Testing, Verification and Validation (ICST)*, 2012, pp. 785–793.

<sup>4</sup>YETI differs from Poulding’s algorithm in that it targets faults identified by ‘automatic’ oracles such as precondition violations and unhandled exceptions.

# LHDiff: Tracking Source Code Lines To Support Software Maintenance Activities

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider Massimiliano Di Penta†

Department of Computer Science, University of Saskatchewan, Canada

†Department of Engineering, University of Sannio, Italy

{md.asad, chanchal.roy, kevin.schneider}@usask.ca, dipenta@unisannio.it

**Abstract**—Tracking lines across versions of a file is a necessary step for solving a number of problems during software development and maintenance. Examples include, but are not limited to, locating bug-inducing changes, tracking code fragments or vulnerable instructions across versions, co-change analysis, merging file versions, reviewing source code changes, and software evolution analysis. In this tool demonstration, we present a language-independent line-level location tracker, named LHDiff, that can be used to track lines and analyze changes in various kinds of software artifacts, ranging from source code to arbitrary text files. The tool can effectively detect changed or moved lines across versions of a file, has the ability to detect line splits, and can easily be integrated with existing version control systems. It overcomes the limitations of existing language-independent techniques and is even comparable to tools that are language dependent. In addition to describing the tool, we also describe its effectiveness in analyzing source code artifacts.

**Keywords**—differencing tools; line tracking; language-independent differencing tool

## I. INTRODUCTION

Software maintenance activities often require tracking source code lines across versions of a software system. One reason may be to separate changed lines from those that are deleted from an old version of a file or added to its new version during code review. If a bug has been identified in a software system, tracking lines containing the bug can assist us in locating the bug in subsequent versions. Mining version archives for co-changed lines can recommend line locations after a change. Source location tracking techniques can help in this regard to obtain an accurate estimation of changed lines. Results obtained from line tracking techniques can be used to track higher level language constructs. For example, the evolution of functions, methods or classes can be tracked across versions using location mapping data of changed lines. In addition, a line tracking tool can be used to track the evolution of other kinds of software artifacts, such as requirements, design documents or configuration files.

Versioning systems (such as CVS or Subversion) and many development environments rely on *Unix diff* or its variants to track lines between two versions of a text file. *Diff* reports the minimum number of added, deleted or changed lines between two file revisions. However, it has limitations in detecting reordered or changed lines that prevents it from becoming an ideal line tracking tool [3]. To address the problems, a number of line tracking techniques and tools have been developed [1].

For example, Canfora *et al.* developed the *ldiff* tool that uses a combination of information retrieval techniques and Levenshtein distance to track lines across versions of a file independent of source code language [2], [4]. In addition to describing various source location tracking techniques, Reiss recommends *W\_BESTI\_LINE* to track lines which uses context and content similarity to track lines [8]. Spacco and Williams developed another tool, *SDiff*, that accesses the syntactic structure of Java source files to track lines across versions [9]. All these techniques are sensitive to the degree and kind of source code changes. While techniques that use an Abstract Syntax Tree (AST) can provide fine grain change information, they require the source file to be parsed which may not always be feasible. For example, developers may issue a file differencing command in the middle of editing a file, which may not be possible to parse at that point in time.

In this paper, we describe the *LHDiff* tool that can track source code lines across versions of a software system. It does not consider the AST of source files, and thus can be applied to arbitrary text files. The tool takes two versions of a file as input and utilizes *Unix diff* to track lines other than those that are reported as either deleted from the old file or added to the new file by *diff*. To determine mapping between the set of added and deleted lines reported by *diff*, the tool uses both context and content of a line. While the line itself represents the content, the context is computed by combining its top and bottom four lines. However, to make the mapping process faster, it leverages the simhash technique [5], [7] to compute mapping candidates for each deleted line of the old file. Next, it computes the context and content similarity using source code lines to select one from each set of mapping candidates. Since the algorithm and its evaluation with other state-of-the-art line tracking techniques have been discussed in a separate paper [1], we briefly summarize the algorithm in this tool paper and explain tool syntax.

The remainder of the paper is organized as follows. Section II provides examples that motivate us developing the tool. Section III briefly describes the *LHDiff* tool. Section IV briefly summarizes the tool effectiveness, while Section V describes its syntax and usage. Finally, Section VI concludes the paper.

## II. MOTIVATING EXAMPLES

Fig. 1 shows four different examples, each one showing two different versions of a file. Due to space limitations, we

Fig. 1: Line tracking examples.

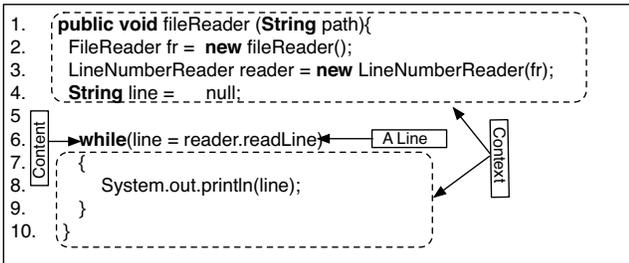


Fig. 2: An example of finding context and content of a line.

only show code fragments from the files (except Fig. 1-a). The highlighted lines are the ones we are interested in and the correct mapping of those lines are marked with arrows. Fig. 1-a shows an example of a line split. In the new version of the file in Fig. 1-b, the highlighted line is moved outside the `for` loop. Fig. 1-c shows an example of a method split, where lines from the `readFile` method are moved to another method in the new version. In the last example (Fig. 1-d), the variable `arrayTb` is replaced with `arrayType` and the condition part of `if` block is updated.

*Unix diff* fails to provide a correct mapping in all four examples. Lines that are changed, or moved are reported as either added or deleted by *diff*. Although existing state-of-the-art line tracking techniques provide better results than *diff*, they have their own limitations. While *SDiff* was able to detect line splits, it fails to detect lines that are moved to another method (see Fig. 1-c). *Ldiff* fails to detect line splits. The technique recommended by Reiss is also not an exception. This shows the limitation of existing source location tracking tools in detecting changed or moved lines and motivates us to develop an enhanced line tracking tool.

### III. LHDIFF: TRACKING SOURCE CODE LINES

Unlike *SDiff* that requires parsing source files, *LHDiff* is purely textual in nature and can be applied to arbitrary text files. It is a hybrid technique because it leverages findings

collected through analyzing incorrect mappings of existing state-of-the-art location tracking techniques. The tool works in five different phases and they are briefly summarized below (further details about the *LHDiff* approach/algorithm can be found in a related research paper [1]):

- **Step 1** involves normalization of input files where the objective is to remove editing differences that are only cosmetic in nature. For example, multiple spaces are replaced with only one.
- **Step 2** applies *Unix diff* to determine the set of unchanged lines. *Diff* reports list of lines that are deleted from the old file and added to the new file. We refer to them as the left and right lists. Since *diff* often fails in detecting changed and moved lines, we need to find the missing link between the lines of these two lists.
- **Step 3** acts as an intermediate step to speed up the mapping process without sacrificing accuracy. For each line in the left list we need to find a line from the right list, if there exists any mapping for that line. We leverage content and context of lines to determine the correct mapping. The line itself represents the content and context is calculated by concatenating its top and bottom four lines (see Fig. 2). To speed up the mapping process, we add an intermediate step instead of acting on source code lines in the first place. For each line in the left list we determine a small set of lines from the right list that are the probable mapping candidates. Instead of working on raw lines, we first apply a hash function to determine a 64 bit binary simhash value for both context and content. A simhash is nothing but a short binary representation of a much longer string with an important property that simhash values of two similar strings have a small Hamming distance. The content and context similarity scores are calculated by calculating the Hamming distance between their corresponding simhash values. If  $a$  and  $b$  are two binary strings, the Hamming distance is the number of 1s in  $a \oplus b$  (the bitwise exclusive

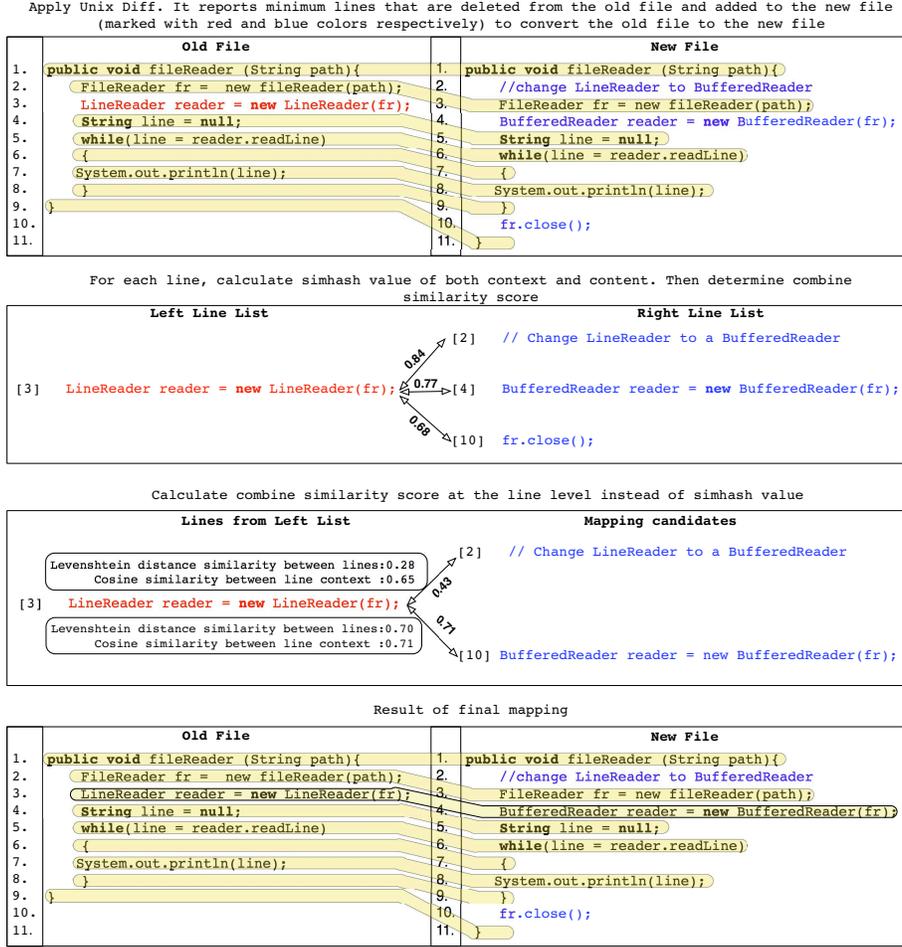


Fig. 3: Working steps of LHDiff.

OR between  $a$  and  $b$ ). We now calculate the combine similarity score by taking 0.6 times the content similarity and 0.4 times the context similarity for all pairs of lines between left and right lists. For each line in the left list, we then determine a set of lines that constitute the possible mapping candidates using a predefined threshold value.

- **Step 4** selects one line from each set of mapping candidates. For each line from the left list and its mapping candidates, we select the line pair that gives the highest similarity score, but this time the similarity scores are calculated on the raw lines instead of on the simhash values. The content similarity between a pair of lines is calculated by applying the Levenshtein edit distance, and the context similarity is calculated using cosine similarity. A combined similarity score is then calculated by combining both context and content similarity.
- **Step 5** deals with detecting line splits. This step can be enabled/disabled through a command line option or programmatically. To detect line splits, *LHDiff* does the following. For each unmapped lines of the old file, it repeatedly concatenates lines of the new file, one at a time

and starting from the top unmapped line of the old file, and calculates the Levenshtein edit distance similarity. We repeat the concatenating operation until the similarity value starts dropping. If such a similarity value crosses a predefined threshold value, we map all the concatenated lines of the new file to that line of the old file.

Fig. 3 shows an example that explains working steps of *LHDiff*. We omit the preprocessing step from the figure for the sake of simplicity.

#### IV. PERFORMANCE

This section briefly summarizes the performance of *LHDiff*. Details about evaluation procedure and detailed results can be found in a separate paper [1]. In the following, we also highlight the benefits of using the simhash technique, and also warn users about the limitation of the tool.

##### A. Correctness of tracking lines and time requirements

We compared *LHDiff* with other state-of-the-art line tracking tools using three different benchmarks (Reiss, Eclipse and NetBeans) where the lines are collected from real-world

TABLE I: LHDiff configuration options.

Option	Description	Default Settings
-i	Ignore case differences	disabled
-k	The size of mapping candidate set	15
-p	Context weight ( $0 \leq CXW \leq 1$ ) and the threshold value ( $0 \leq TH \leq 1$ ) for combine similarity score used in Step 4. Content weight will be automatically set to $1 - CXW$	0.4 and 0.6
-cnm	Line content similarity metric	Levenshtein
-cxm	Line context similarity metric	Cosine
-cxs	Context size	4
-ls	detect line split	disabled
-ob	Output both line number and content	display only line number

applications. Our evaluation results reveals that *LHDiff* outperforms all language dependent techniques and even a language dependent technique, *SDiff*. For example, while the tool correctly maps 82.8% of target lines in Eclipse benchmark, where the files underwent changes to a higher degree than other benchmarks, the closest score to ours is 74.1% by *SDiff*.

In terms of execution time, *LHDiff* is comparable with other state-of-the-art techniques. Although *LHDiff* may require slightly more time than some of the techniques, this small additional time gives the tool the ability to achieve higher accuracy.

### B. Performance gain from the simhash technique

The tool leverages the simhash technique to filter mapping candidates for lines from the left list because calculating similarity scores at the line level between all pairs of lines of left and right lists is computationally expensive, and this is particularly true for what concerns the Levenshtein edit distance. To better understand the effect of simhash, we ran *LHDiff* without enabling step-3 on the files in Reiss benchmark [8], which consists of 25 revisions of `ClideDataManagerManager.java` file. While the original *LHDiff* tool took 5.22 sec. to complete tracking line locations, the modified version took 18.06 sec. to complete running. This indicates more than three times improvement of running time from using simhashing, and it can be even more when differencing a large number of source code files.

### C. Limitations

*LHDiff* fails to track lines when both context and content of a line changes a lot. The combined similarity score gives more weight on the content similarity and this sometimes leads to an incorrect mapping for short lines. For example, even if the context suggests that there is no connection between line 13 of the old file to line 26 of the new file, content similarity gives a very high score, because of the matching keywords and the length of the keywords dominating the total line length which results in an incorrect mapping (see Fig. 5).

## V. TOOL IMPLEMENTATION

An implementation of the *LHDiff* tool is available in Java as a command line tool [6]. The tool supports the use of various

```

muhammad-asaduzzaman@macbook-pro:~/Desktop/parvez$ java -jar lhdiff.jar -ob ./old.txt ./new.txt
LHDiff version: 1.0
[1]public void fileReader (String path){ -->[1]public void fileReader (String path){
[2]
[3]//file should exist -->[2]//file should be present in the system
[4]fileReader fr = new fileReader(); -->[3]fileReader fr = new fileReader(path)
[5]lineNumberReader reader = new lineNumberReader(fr); -->[4]BufferedReader reader = new BufferedReader(fr);
[6]String line = null; -->[5]String line = null;
[7]while(line=lr.readLine){ -->[6]while(line=br.readLine)
[8]System.out.println(line); -->[7]System.out.println(line);
[9]
[10]
[11]
[12]
[13]
muhammad-asaduzzaman@macbook-pro:~/Desktop/parvez$
  
```

Fig. 4: Tool output example.

```

. . .
11. protected int add(int x, int y){
12.     int z = x+y;
13.     return z;
14. }
15. . . .
16.
17.
. . .
21. protected int sum(int input){
22.     int s = 0;
23.     for(int i=1;i<=x;i++){
24.         s = s + i;
25.     }
26.     return s;
27. }
28. public int add(int n1,int n2){
29.     int sumOfNumbers = x+y;
30.     return sumOfNumbers;
31. }
. . .
  
```

Fig. 5: An example of incorrect mapping.

similarity/distance metrics (Levenshtein, Cosine, Jaccard, and Dice) and Table I summarizes the configuration options *LHDiff* has available. Users can also type help in the command line to learn details about different available options. Fig. 4 shows an example of running the tool from the command line, where the input files are the same ones we used in Fig. 3.

## VI. CONCLUSION

This paper describes *LHDiff*, a line tracking tool that can be applied to any text files, can easily be integrated with existing version control systems, and can even detect line splits. The demonstration will show how *LHDiff* can be used to analyze source code or other text files, track buggy lines, and examples from real world applications where *LHDiff* succeeds in mapping lines but *Unix Diff* or other state-of-the-art line tracking techniques fail(s). As a work-in-progress, we are investigating how to reduce the degree of false mappings, and developing visualizations to aid comprehending the results from the tool.

## REFERENCES

- [1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, M. Di Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines", accepted to be published in Proc. ICSM, 2013.
- [2] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach", in IEEE Softw., pp. 50-57, 2009.
- [3] G. Canfora, L. Cerulo, M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories", in Proc. MSR, pp.14, 2007.
- [4] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool", in Proc. ICSE, pp. 595-598, 2009.
- [5] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in Proc. STOC, pp. 380-388, 2002.
- [6] "The LHDiff Tool", <http://asaduzzamanparvez.wordpress.com/Research>
- [7] G. S. Manku, A. Jain and A. D. Sarma, "Detecting Near Duplicates for Web Crawling", in Proc. WWW, pp. 141-150, 2007.
- [8] S. P. Reiss, "Tracking source locations", in Proc. ICSE, pp. 11-20, 2008.
- [9] J. Spacco and C. Williams, "Lightweight Techniques for Tracking Unique Program Statements", in Proc. SCAM, pp. 99-108, 2009.

# gCad: A Near-Miss Clone Genealogy Extractor to Support Clone Evolution Analysis

Ripon K. Saha\*    Chanchal K. Roy†    Kevin A. Schneider†

\*The University of Texas at Austin, USA

†University of Saskatchewan, Canada

rip@utexas.edu, chanchal.roy@usask.ca, kevin.schneider@usask.ca

**Abstract**—Understanding the evolution of code clones is important for both developers and researchers to understand the maintenance implications of clones and to design robust clone management systems. Generally, a study of clone evolution starts with extracting clone genealogies across multiple versions of a program and classifying them according to their change patterns. Although these tasks are straightforward for exact clones, extracting the history of near-miss clones and classifying their change patterns automatically is challenging due to the potential diverse variety of clone fragments even in the same clone class. In this tool demonstration paper we describe the design and implementation of a near-miss clone genealogy extractor, gCad, that can extract and classify both exact and near-miss clone genealogies. Developers and researchers can compute a wide range of popular metrics regarding clone evolution by simply post processing the gCad results. gCad scales well to large subject systems, works for different granularities of clones, and adapts easily to popular clone detection tools.

**Index Terms**—Type-3 clones; clone genealogy; clone evolution

## I. INTRODUCTION

After a decade of active research, it is evident that code clones have both a positive [3] and a negative [5] impact in the maintenance and evolution of software systems. Code cloning is inevitable in software development, and in order to exploit the advantages of clones while lowering their negative impact, it is important to understand the evolution of clones and manage them accordingly.

Generally, a clone evolution study starts with detecting clones in multiple versions of a program, and constructing genealogies by mapping clones across the different versions. A clone genealogy tells us how the code fragments of a clone class change through versions during the evolution of a subject system. Researchers have proposed and implemented a number of clone genealogy extractors (CGEs) to study the evolution of clones. However, most of the tools were designed focusing on some particular tasks of interests and for only Type-1 and Type-2 clones. Thus CGEs rarely meet the current diverse requirements such as fast construction and classification of near-miss clone genealogies and adaptation/integration of a third party clone detection tool. Furthermore, most of the reported tools are not publicly available.

The recently developed incremental clone detection methods [2] improve the genealogy construction time considerably by integrating clone mapping with clone detection. However, these methods have their own set of limitations. First, they

are unable to utilize the results obtained from a classic non-incremental clone detection tool as the detection of clones and their mapping is tightly integrated. Since most existing clone detection tools are non-incremental, they restrict developers and researchers to using a limited number of clone detection tools. For flexibility it is important to have a CGE that is independent of the clone detection tools. Second, with each new revision or release of the subject system, the entire detection and mapping process needs to be repeated, because clones are detected and mapped concurrently. Since clone management is likely being conducted on a changing system, it is a disadvantage for an approach to require detecting clones for all versions each time a new revision/version is produced. Third, the incremental approach is fast enough for both detecting and mapping for a given set of revisions. However, it might not be as beneficial for the release level because there might be significant differences between releases.

In this tool demonstration paper, we describe the design and implementation of a near-miss CGE, gCad (evolved from our earlier research [6]) that can extract both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies across multiple versions of a program, and identify their change patterns automatically. Genealogies are constructed incrementally by merging current mapping results with previously stored genealogies to give a complete result. gCad scales well to large subject systems, works for different granularities of clones, and adapts easily to popular clone detection tools. Developers and researchers also can compute many popular metrics of clone evolution by simply post processing the gCad results.

## II. APPROACH AND IMPLEMENTATION

This section describes the overall design and implementation of gCad. Usually gCad accepts  $n$  versions of a program and their clones, maps clone classes between the consecutive versions, and extracts how each clone class changes throughout an observation period. Therefore, it is expected that users will have detected clones in all  $n$  versions of the program before running gCad. A version may be a release or a revision. gCad mainly works in the following four steps to construct and classify genealogies: (1) Function Mapping, (2) Clone Mapping, (3) Automatic Identification of Change Patterns, and (4) Constructing Genealogies. The first three steps are performed for each consecutive version pairs. Finally, gCad

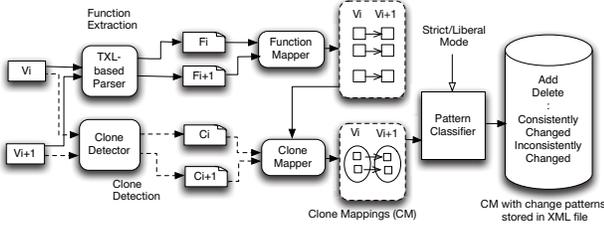


Fig. 1. The First Three Steps of the gCad Framework

merges all the results to construct genealogies for the  $n$  versions. Figure 1 shows the overall structure of gCad for two versions of a program. A more detailed description of the approach including the analysis of the time complexity can be found elsewhere [6].

### A. Function Mapping

For two given consecutive versions,  $v_i$  and  $v_{i+1}$  of a software system, first gCad extracts all the function signatures from both the versions. For extracting functions gCad uses TXL [1], a special-purpose programming language that supports lightweight agile parsing techniques. We exploit TXL's extract function operator (denoted by  $[ \hat{ } ]$ ) to enumerate all the functions. For each function gCad stores the function signature, class name, file name, the start and end line number of the function in the file, and its complete directory location in an XML file. A function is the smallest unique element of a software project if we consider the signature of a function along with its class name and complete file path. Therefore, gCad uses these attributes as a composite key to map functions between two versions, which is computationally very fast. However, in practice some functions are renamed, or could move to different files or directories during the evolution. In those cases, gCad uses the longest common subsequence count (LCSC) similarity metrics of function name and (comment-free pretty-printed) body to find the origin of a function.

### B. Clone Mapping

Typically a clone detection tool reports results as a collection of clone classes where each clone class has two or more clone fragments. A clone fragment could be of any granularities such as function, structural block, or arbitrary block. Let  $CC^i = \{cc_1^i, cc_2^i, \dots, cc_n^i\}$  be the reported clone classes in  $v_i$  where  $cc_j^i = \{CF_{j1}^i, CF_{j2}^i, \dots, CF_{jm}^i\}$ . Here  $CF_{jk}^i$  refers to the clone fragments of the clone class  $cc_j^i$  where  $1 \leq k \leq m$ . In order to map clones between two versions, gCad first maps each clone fragment  $CF^i$  to its contained (parent) function,  $F^i$  in  $v_i$  using the following algorithm.

```
boolean isContained(Block CF, Function F) {
    return (CF.FileName == F.FileName)
        AND (CF.BeginLine >= F.BeginLine)
        AND (CF.EndLine <= F.EndLine)
}
```

At this point, since all the functions are already mapped (from Section II-A), the problem of mapping clones between two versions of a program reduces to the mapping of clones between two versions of a function. Therefore, mapping clones in gCad is computationally very fast. If  $F^i$  contains only one

clone fragment, gCad can easily map that clone fragment in  $v_{i+1}$  since it already knows the corresponding mapped function  $F^{i+1}$  in  $v_{i+1}$ . The mapped clone fragment  $CF^{i+1}$  will be found in  $F_{i+1}$  if it has not been removed. If  $F^i$  has more than one clone fragment, gCad uses the LCSC similarity score to map corresponding clone fragments in  $F_{i+1}$ . However, there might be still some clones that have not been mapped yet. They might be file clones, clones that span more than one functions, clones in declarations, or clones in C preprocessor code. gCad maps such clones using the LCSC similarity metric and file name. Once the clone mapping is completed at the fragment level, gCad maps clones at the class level. A clone class in  $v_i$  can be split into two or more clone classes in  $v_{i+1}$  due to inconsistent changes. For each clone fragment of a given clone class  $cc_j^i$ , gCad checks if they map into single or different clone classes in  $v_{i+1}$ . If all the clone fragments are mapped to the same class,  $cc_x^{i+1}$ , gCad maps  $cc_j^i \rightarrow cc_x^{i+1}$ . On the other hand, if they are mapped to multiple classes,  $\{cc_x^{i+1}, cc_y^{i+1}, \dots\}$ , which usually indicates a split, gCad keeps track of them as  $cc_j^i \rightarrow \{cc_x^{i+1}, cc_y^{i+1}, \dots\}$ .

### C. Automatic Identification of Change Patterns

Automatic and accurate identification of change patterns is one of the important features of a CGE. Although identifying whether a Type-1 clone class changed consistently or inconsistently is straightforward, it is challenging for near-miss (Type-2 and Type-3) clones due to the diverse variety of clone fragments in the same clone class. gCad applies a multi-pass computationally efficient method to identify the change patterns of both exact and near-miss clones.

In the first pass, gCad identifies the clone classes that did not change in the next version (*Static*), and those clone classes that have split. The program identifies the split clone classes as an inconsistent change because it is evident that their fragments changed inconsistently, and thus they are part of two or more clone classes in the next version.

In the second pass, gCad makes a decision for Type-1 and those Type-3 clone classes where modifications of different fragments of the same clone class are only limited to line additions or deletions but do not have any variable renaming. If  $cc_j^i \rightarrow cc_{j'}^{i+1}$  is such a mapping, gCad computes the differences between each of the clone fragments of  $cc_j^i$  with the corresponding clone fragments of  $cc_{j'}^{i+1}$  using *diff*. It should be noted that gCad uses comment-free pretty-printed lines of source code to ignore any formatting or commenting differences. If the differences for each of the fragment pairs ( $CF_{jk}^i, CF_{j'k'}^{i+1}$ ) are the same, then the clone class is classified as a consistent change, otherwise as an inconsistent change.

In the third pass gCad considers the rest of the clone classes (Type-2 clones and Type-3 clones with identifiers renaming). Since the clone fragments of these clone classes have variations in their identifiers, gCad cannot exploit *diff* directly because the differences will not be the same even if the fragments changed consistently. In order to deal with this issue, gCad consistently renames the identifiers of the clone fragments using TXL. For example, the first identifier and all

```

<mappings>
  <version1>
  <version2>
  <mapping>
    <id>
    <clone type>
    <change pattern>
    <change of fragment>
    <clone class of v1>
    <id>
    <number of nlines>
    <number of fragments>
    <source>
      <file name>
      <start line>
      <end line>
      <source fragment id>
    </source>
    <clone class of v1>
    <clone class of v2>
    ....
  </clone class of v2>
</mapping>
<mapping>
  .....
</mapping>
.....
</mappings>

```

Fig. 2. XML Structure for Storing Clone Mappings between Two Versions

its occurrences in a fragment is replaced by  $x_1$ , the second identifier and all of its occurrences will be replaced by  $x_2$  and so on. gCad then computes the differences. As before, if the differences are the same, gCad classifies the change pattern as a consistent change, otherwise as an inconsistent change. gCad also reports add and delete patterns based on the number of fragments in a clone class in the previous and next version. All the identified mappings and their change patterns are stored in an XML file (Figure 2) for future use.

#### D. Genealogy Construction

At this point, gCad has all the clone mappings between each consecutive versions  $\{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)\}$  stored in XML files. In this step, gCad combines all of the results of each version-pair by matching clone class ids. For example, if a clone class  $cc_1^1$  in  $v_1$  maps to a clone class  $cc_2^5$  in  $v_2$ , which again maps to  $cc_3^4$  in  $v_3$ ,  $\{cc_1^1 \rightarrow cc_2^5 \rightarrow cc_3^4\}$  forms a clone genealogy. Now if a clone genealogy has any inconsistent change patterns during the evolution, it will be classified as an *Inconsistently Changed Genealogy*. If a genealogy has any consistent change patterns but does not have any inconsistent change patterns, it will be classified as a *Consistently Changed Genealogy*. If a genealogy has any *Add* or *Delete* change patterns, it will be classified accordingly.

### III. TOOL FEATURES

1) *Clone Coverage*: gCad can construct genealogies for both exact and near-miss clones and classifies their change patterns automatically. gCad also works for different clone granularities (e.g., function clones and block clones) and clone relationships (clone pairs, clone classes, and RCF [4]).

2) *Adaptability*: From the clone mapping phase, it should be noted that gCad uses only file name and line numbers to find the parent function of a clone, and then uses only clone code fragments for mapping purposes. Therefore, gCad is easily adaptable to any clone detectors that report their results

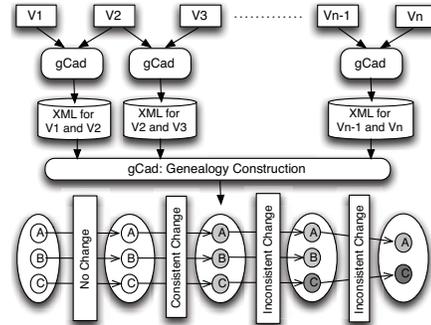


Fig. 3. Genealogy Construction

with this information. We have successfully tested gCad for NiCad, CCFinderX and iClones with high accuracy [6].

3) *Supporting Languages*: gCad uses TXL for function extraction and normalizing source code to classify change patterns of near-miss clones. Therefore, gCad can support the wide range of programming languages that TXL supports. We have tested gCad for C, C#, and Java programming languages.

4) *Operating Modes*: An issue with classifying change patterns of Type-3 clones is that whether the changes in the gap (dissimilar lines between clone fragments) should be considered in determining the change pattern. One might argue that the changes in the gaps should not be considered because those lines are already different and thus cannot change consistently. However, sometimes although gapped lines are textually different, they are semantically similar. Figure 4 shows such a real world example in dnsjava. Therefore, developers may miss some unintentional inconsistent changes if the CGE ignores gaps. gCad supports both ways in the form of two modes.

**Liberal Mode**: In this mode, gCad ignores changes that occur in the gaps of each clone pair in the same clone class. Therefore, a change will be identified as inconsistent change only when similar lines of any clone pair in the same clone class change differently with respect to one another.

**Strict Mode**: In this mode, gCad does not consider the gaps as a special case. If the changes to the clone fragments are not the same, it will be considered as an inconsistent change.

5) *Flexibility of changing the scope of study*: In the genealogy construction phase, we described how gCad constructs genealogy incrementally. Therefore, users can add a new version (e.g.,  $v_{n+1}$ ) anytime easily. She just needs to run gCad for mapping clones between  $v_n$  and  $v_{n+1}$  and gCad can add the new results to form the genealogies from version  $v_1$  to  $v_{n+1}$ . Users can also insert a version in the middle (e.g.,  $v_i$ ) by running gCad for mapping clones of  $(v_{i-1}, v_i)$  and  $(v_i, v_{i+1})$ .

6) *Scalability*: gCad has almost a linear time complexity with respect to the number of functions in the subject system. Therefore, gCad scales well to very large systems. We successfully ran gCad for 45 releases of the Linux kernel. gCad took only 2 minutes 35 seconds per release to construct and classify all the clone genealogies [6].

7) *Extensibility*: gCad is extensible. We have already developed an extension of gCad that populates gCad results into a MySQL database to support SQL query on clone evolution

```

public Object sendAsync(final Message query, final ResolverListener listener) {
    final Object id;
    :
    String name = this.getClass() + ": " + query.getQuestion().getName();
    WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
    return id;
}

public Object sendAsync(final Message query, final ResolverListener listener) {
    final Object id;
    :
    String name = getClass() + ": " + query.getQuestion().getName();
    WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
    return id;
}

```

Fig. 4. A Semantically similar but textually different change in a Type-3 clone class in dnsjava

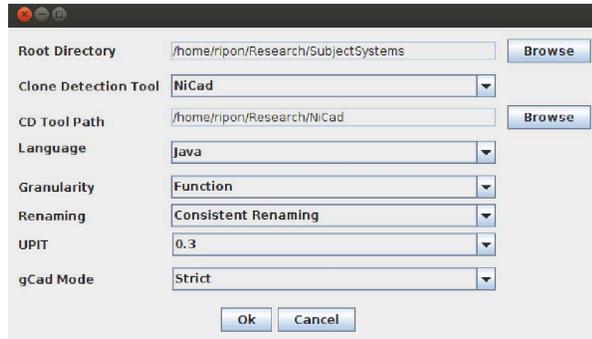


Fig. 5. GUI for gCad Settings

data. Development of a comprehensive clone evolution visualization tool is in progress on the top of gCad.

8) *Working with gCad:* gCad is very easy to configure and use. Currently gCad provides a set of GUIs for initiating various operations. Users just need to set some parameter values through a simple GUI (shown in Figure 5) as necessary to run gCad for a given system. gCad reports various results through a set of XML, HTML, and text files. The gCad tool, a sample result set for ArgoUML, and all other supporting documents (a live demonstration and user manual) are available at <https://webspace.utexas.edu/rks848/www/miscellaneous.html>.

#### IV. APPLICATION

Understanding the evolution of code clones is important to manage clones efficiently and gCad can be a useful tool for developers to make timely decisions regarding clones by collecting relevant information regarding clone evolution without additional efforts. For example, developers can understand the changing nature or maintenance effort of clones by observing the proportion of consistently and inconsistently changed genealogies. They can also choose a set of clone classes for refactoring that change consistently and frequently, or can manually check if an inconsistent change was intentional.

Researchers can use gCad to analyze the clone evolution from various perspectives to design new techniques to manage clones. We already performed two empirical studies using gCad to understand the evolution of Type-3 clones and to evaluate the conventional wisdom in clone removal. In the first study [7], we used gCad to compute the proportions of consistently and inconsistently changed genealogies, change frequencies, and ages of Type-1, Type-2, and Type-3 clones

separately to understand if the evolution of Type-3 clones is different from that of Type-1 and Type-2. We also used gCad's strict and liberal modes to understand the extent of changes in gaps for Type-3 clones. We found that the proportion of consistently changed Type-3 clone genealogies increases considerably if we ignore the changes in gaps. In the second study [8], we used gCad to find the clone classes that were removed from systems. Then we investigated different attributes of the code clones, such as number of clone fragments, their distributions in different files, change patterns, change frequencies, and so on, in relationship to clone removal.

#### V. SUMMARY

This tool demonstration paper describes the design and implementation of a near-miss clone genealogy extractor, gCad to support the analysis of clone evolution. gCad can extract both exact and near-miss clone genealogies, classify their change patterns automatically, and provide various important information regarding clone evolution through a number of widely used metrics. Our evaluation results from a previous study [6] show that gCad is accurate, scales well to very large systems (e.g., Linux Kernel releases) and is adaptable to different clone detectors, which makes it useful for understanding the various evolutionary phenomena of code clones. We believe gCad would be helpful not only for clone researchers but also for developers or maintenance engineers in making decisions for reducing the negative impacts of code clones.

#### REFERENCES

- [1] J. R. Cordy, "The TXL Source Transformation Language," *Sci. of Com. Prog.*, 61(3):190–210, 2006.
- [2] N. Göde and R. Koschke, "Studying Clone Evolution using Incremental Clone Detection," *JSME*, 25:165–192, 2010.
- [3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An Empirical Study of Code Clone Genealogies," *Proc. ESEC-FSE*, 2005, pp. 187–196.
- [4] J. Harder and N. Göde, "Efficiently Handling Clone Data: RCF and cyclone," *Proc. IWSC*, 2011, pp. 81–82.
- [5] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that Smell?," *Proc. MSR*, 2010, pp. 72–81.
- [6] R. K. Saha, C. K. Roy, and K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies," *Proc. ICSM*, 2011, 293–302.
- [7] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the Evolution of Type-3 Clones: An Exploratory Study," *Proc. MSR*, 2013, 139–148.
- [8] M. Z. Zibran, R. K. Saha, C. K. Roy and K. A. Schneider, "Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study," *Proc. SAC*, 2013, pp. 1223–1230.

# eCITY: A Tool to Track Software Structural Changes using an Evolving City

Taimur Khan\*, Henning Barthel†, Achim Ebert\* and Peter Liggesmeyer\*

\* University of Kaiserslautern

Gottlieb-Daimler-Str. 67663

Kaiserslautern, Germany

Email: {tkhan, ebert, liggesmeyer}@cs.uni-kl.de

† Fraunhofer IESE

Fraunhofer-Platz 1 - 67663

Kaiserslautern, Germany

Email: Henning.Barthel@iese.fraunhofer.de

**Abstract**—One of the main challenges in the maintenance of large-scale software systems is to ascertain the underlying software structure and to analyze its evolution. In this paper we present a tool to assist software architects and developers in not only understanding the software structure of their system but more importantly to track the insertion, removal, or modification of components over time. The tool is based on the idea that the above-mentioned stakeholders should have an intuitive, efficient, and effective means to detect when, where, and what structural changes took place. The main components include an interactive visualization that provides an overview of these changes. The usefulness of this approach is highlighted through a summary of a user study we conducted.

**Index Terms**—software architecture visualization; software maintenance; software evolution

## I. INTRODUCTION AND RELATED WORK

Software systems nowadays undergo continuous changes to meet new requirements, adapt to new technology, and to repair errors [1]. Such changes often lead to a situation where the original architectural design decays unless proper maintenance is performed. In order to monitor the design and to remedy undesirable artifacts, software teams rely on software visualization tools and techniques. However, the development of such tools and techniques is hindered mainly due to the following two factors: 1) performing visually supported maintenance is convoluted due to the complex, abstract, and difficult to observe nature of software systems [2], and 2) dealing with the complexity that emerges from the huge quantity of evolution data [3]. The former task can be eased through the careful selection and implementation of a visual representation of the software structure that aids in forming a precise and comprehensible mental map of the system [4]. Further, with respect to software evolution analyses, it is necessary to track this mental map over time to explain how a system has evolved to its present state and to predict its future development [5].

There exist only a small number of visualizations that deal with the above challenges and represent structural changes of a system architecture over time. One such visualization is the work of Holten et al. that presents a technique to compare

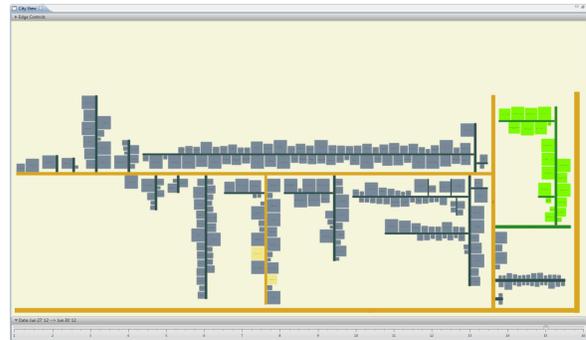


Fig. 1. eCITY City View

the software hierarchies of two software versions [6]. The algorithm positions matching nodes opposite to each other and utilizes shading in order to better compare the two versions. Another approach can be found in the work of Collberg et al. [7] that imposes a temporal component on standard graph drawing techniques. They employ force-directed layouts to plot call graphs, control-flow graphs, and inheritance graphs of Java programs. Changes that the graphs have gone through since inception are highlighted through the use of color.

A recent research focus is to utilize and extend intuitive metaphors to aid in the visualization of software systems so that users may have access to representations that are both easy to grasp and comfortable to work with. It is for this reason that we found inspiration for our work from the original contribution of Steinbrückner et al., where they propose the idea of stable city layouts for evolving software systems [4]. Our work is an adaptation of their underlying graphing model that employs a combination of animated transitions and color interpolations to grow or shrink the city and highlight modification statuses respectively. While the implementation details are provided later, a snapshot can be seen in Fig. 1.

The goal of the *eCITY tool*<sup>1</sup> is to provide a visualization that

<sup>1</sup>eCITY Website: <http://daddi.informatik.uni-kl.de/~khan/eCITY/>

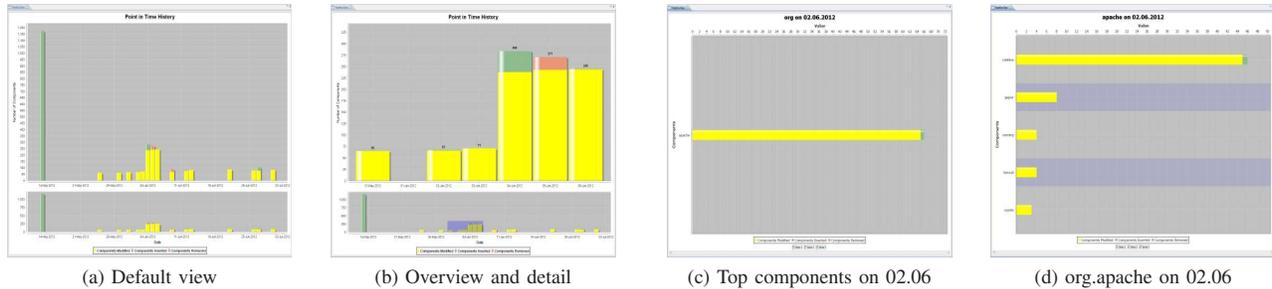


Fig. 2. Timeline View

assists in the daunting task of large-scale software architecture evolution analyses. To realize this goal, we provide views that assist in the tracking of hierarchical structural changes over time. As such, the main contributions of this work are:

- the *Timeline View*; a managerial view of the evolution data that uses charts to highlight changes (components added, removed, modified) over time.
- the *City View*; a dynamic city layout - one that employs both animated transitions to grow or shrink the city as components are added or removed and color interpolations to highlight the evolution of components.
- summary of a controlled experiment that integrates eCITY with a conventional architecture maintenance tool.

This paper is organized as follows: the eCITY tool is described in Section II, evaluation results are reported in Section III, and concluding remarks are found in Section IV.

## II. THE ECITY TOOL

Typically, conducting an analysis of the software evolution is a two-part process: 1) extracting historical recordings through the use of Software Configuration Management systems that track activities performed by developers, testers, and users and 2) reconstruction of the time-varying data in a visually intuitive manner. Due to the fact that there are a number of tools that perform the former task adequately [8], the focus of this work is on the latter. The eCITY tool was consciously designed as an eclipse plug-in so that it may be interfaced with a traditional software maintenance tool. As such, the maintenance tool extracts the above-mentioned historical recordings and interfaces with eCITY to have additional views on the evolution of the software structure.

In this section, we first look at how to implement the generic graph model interface and subsequently focus on the two main views of the tool; namely the *Timeline View* and the *City View*.

### A. Implementation of a Model Interface

A pre-requisite to incorporate the eCITY views into a software maintenance tool is the implementation of a model interface that accesses the software systems historical data. In order to maintain a flexible and easy-to-implement approach, this interface is to be a generic JGraphT [9] graph implementation - a free Java graph library that provides mathematical graph theory objects and algorithms.

This graph interface should capture the software components and all their dependencies as follows: packages and classes of the software system should be customized vertices that contain the name of the software component and a collection of date and modification status (inserted, removed, or modified) attributes. Similarly, all dependencies should contain the type (i.e. containment, inheritance, call, etc.) and a collection of date and modification status attributes.

### B. Timeline View

The main idea behind the Timeline View is to provide the user with a managerial overview of changes made to the software system over time. This is realized through a combination of interactive bar charts that represent the number of insertions, removals, and modifications made to software components such as packages and classes over time. These changes are depicted through a typical traffic-light color scheme; removals are represented using the red color, modifications using the yellow color, and insertions using the green color.

There are several ways in which the user may utilize this view. One of which is to customize the underlying timeline depending on the frequency of the changes made to the software system. Fig. 2 shows an example with a default timeline that is set at a daily resolution. We choose this case to highlight the overview-and-detail approach of the primary Timeline view. As shown in Fig. 2a, this view contains a combination of plots. The plot at the bottom of the drawing canvas serves as an overview where the user can interactively create and/or manipulate a rectangular selection, while the plot at the top provides details of this selection (Fig. 2b).

Additionally, the user may want to dig deeper into the changes made at a particular point-in-time, in which case he/she may select the entry for that date and update the Timeline View. The view then updates itself by switching the combined plots with a singular plot that provides details of the top level component at the chosen time stamp (Fig. 2c). Once this interaction has taken place, the user may want to recursively explore the distribution of changes over the hierarchy (Fig. 2d) by selecting the relevant component.

### C. City View

As the main view of the eCITY tool the purpose of the City View is to add context to the evolution of the software



Fig. 3. Initial and updated city layout for suburb of org.apache.tomcat

structure, i.e. when, where, and what components were inserted, removed, or modified (Fig. 1). To achieve this, eCITY implements a city metaphor to provide an overview of the entire system at a particular point in its evolution process and offer the user a means to interactively explore these changes.

As mentioned earlier, we took inspiration from the work of Steinbrücker et al. [4]. In their work, they describe a three-staged visualization approach; where they capture the software systems structure and evolution into a primary model, creates a secondary model that adds geometric information including elevation levels to depict a software systems development history, and a third stage that imposes symbols or diagrams to the secondary model. Instead, our approach adapts their secondary model to highlight the evolution of components via animation - we only follow their methodology to calculate the representation of the software structure at the initial point-in-time (Fig. 3a). Further, instead of geometrically mapping different development history details onto the city layout we allow the user to interactively update the layout itself and the modification status of its components to highlight structural changes over time.

During the initialization phase, once the initial layout is calculated eCITY sequentially goes through every other point-in-time to recalculate the city layout based on the appropriate insertion and/or removal of both classes and packages. Details of each city component, such as its location and modification status, are stored for each point-in-time in the form of key frames to allow for real-time interaction. Once all the artifacts have been loaded, key frame animation technique is used to interpolate each components color representation and the city suburbs grow and shrink to depict changes made.

This view is designed for the user to explore these changes interactively. As such, the city contains navigation support through the use of a mouse to pan and zoom to examine its suburbs more closely. Further, there are two different modes of tracking the software structures evolution. One of these modes requires the user to interactively invoke a slider and manually update the city to another point in time, while the other relies on a date from the Timeline View to animate the city sequentially through all the intermediate points-in-time.

These modes of interaction either update or animate the city through the use of colors that depict the modification status and the growing and shrinking of suburbs to represent the addition and/or removal of classes and packages (Fig.3b). The earlier mentioned traffic-light scheme is extended to include components that have not changed in between different time stamps; i.e. while red, yellow, and green colors are used for the removal, modification, and insertion of each component, we use grey to represent unchanged components.

However, in certain scenarios it is not sufficient for the analyst to sequence through the points-in-time to detect the structural changes. An example of this would be when the analyst would like to directly compare the architecture at two disjoint points in time. To address this, a comparison mode is being developed that would allow the user to compare the states of these dates side by side. Fig. 4 depicts a first attempt where the layout is recalculated using the from date and sequentially only inserts the new components till the to date is reached. Each component is split in half, where the left or top half depending on the whether the orientation is horizontal or vertical represents the state at the from date and the right or bottom half shows the aggregation of changes to the to date. If a component does not exist in any one of these time stamps, the color of the canvas is used to depict this case.



Fig. 4. Comparing a suburb on two different dates

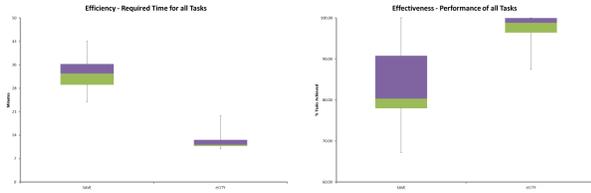


Fig. 5. Overall Efficiency and Effectiveness results

### III. EVALUATION RESULTS SUMMARY

The plug-in concept of the eCITY-Tool makes it possible to integrate it into a traditional software maintenance tool. As part of this work we have been working together with product line architects at the Fraunhofer IESE to enhance their SAVE (Software Architecture Visualization and Evaluation) tool that evaluates software architectures while they are constructed as well as after their construction [10]. In the original SAVE workflow, architects would apply a *Point-In-Time* filter to update their diagrams and manually track the number and type of changes made - a process that was deemed to be tedious and error prone. With eCITY, however, the architects were able to use views that provided them with an interactive overview of these structural changes.

One of the core functionalities of the SAVE tool is responsible for extracting the underlying architectural model from the software systems source code. This SAVE data model contains architectural data about a software system over a period of time and includes details such as when a component or a dependency was created, modified or removed. Using the model interface described in Section II-A, we created a simple graph representation that was utilized by the eCITY views to perform basic software architecture evolution tasks.

A controlled experiment was conducted with 38 participants to gauge the benefit of applying our approach instead of relying on manual tracking of the number and type of changes using the SAVE tool. We choose the Apache Tomcat system to analyze due to the fact that it is a real software system and due to its architectural models being available in SAVE. The underlying data model contained 16 fact extractions (points-in-time) of the Apache Tomcat system between the time period 14.05.2012 and 02.07.2012. The results of our experiment clearly indicate that an improved configuration of the visualization influences the efficiency and the effectiveness of typical architecture evolution tasks greatly. As Fig. 5 shows, we recorded a gain in *efficiency (required time)* by 170% and a gain in *effectiveness (accuracy)* by 15%. Further, our results also show a significant improvement in the usability of our approach as compared to the original configuration.

### IV. CONCLUSION

In this paper we presented the eCITY tool, a plug-in for software maintenance tools, which provides additional views to better comprehend the software structure and more specifically its evolution. Integrating this tool with a conventional

maintenance tool helps the users be more efficient and effective in performing basic software evolution tasks. Using the eCITY views, the stakeholders can interactively get a good overview of the structural changes made to their system over time and may directly compare two disjoint points in time. Our approach of interactively evolving a city layout to highlight structural changes over time was well received by Product Line Architects at the Fraunhofer IESE, who especially appreciated this adaptation of a city layout and found our implementation to be both natural and intuitive.

In the future, we intend to continue to work on eCITY and add more functionalities and features in order to encompass more evolution details and provide improvements. One of these enhancements we are currently working on addresses the most significant limitation of eCITY – that it is confined to the evolution of software structures. It is quite typical that not only the structure of a software system evolves over time but also the relationships and dependencies of its subcomponents. Other possible improvements include an embedded search engine to locate specific components, highlighting of suburbs undergoing change in a time interval using multiple fish-eye views, improvement of the Comparison mode, and the addition of other useful views. An example of such a view would be an overview of the City View that provides navigational context.

### REFERENCES

- [1] M. M. Lehman and L. A. Belady, Eds., *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.
- [2] M. Petre and E. Quincey, "A gentle overview of software visualization," *PPIG News Letter*, pp. 1–10, September 2006.
- [3] L. Voinea and A. Telea, "Multiscale and multivariate visualizations of software evolution," in *Proceedings of the 2006 ACM symposium on Software visualization*, ser. SoftVis '06. New York, NY, USA: ACM, 2006, pp. 115–124. [Online]. Available: <http://doi.acm.org/10.1145/1148493.1148510>
- [4] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th international symposium on Software visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 193–202. [Online]. Available: <http://doi.acm.org/10.1145/1879211.1879239>
- [5] M. D'Ambros and M. Lanza, "Reverse engineering with logical coupling," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–198. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1174510.1174729>
- [6] D. Holten and J. J. van Wijk, "Visual comparison of hierarchically organized data," *Computer Graphics Forum*, vol. 27, no. 3, pp. 759–766, 2008. [Online]. Available: <http://dblp.uni-trier.de/db/journals/cgf/cgf27.html#HoltenW08>
- [7] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*, ser. SoftVis '03. New York, NY, USA: ACM, 2003, pp. 77–ff. [Online]. Available: <http://doi.acm.org/10.1145/774833.774844>
- [8] G. Canfora, M. D. Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, no. 4, pp. 142–151, 2011.
- [9] B. Naveh, "JGraphT Website," 2003-2011, online; Accessed 04-June-2013. [Online]. Available: <http://jgraph.org>
- [10] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *Proceedings of the Conference on Software Maintenance and Reengineering*, ser. CSMR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 279–294. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1116163.1116412>

# ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications

Juan Castrejón<sup>\*†</sup>, Geneveva Vargas-Solar<sup>†‡</sup>, Christine Collet<sup>§</sup>, Rafael Lozano<sup>¶</sup>  
<sup>\*</sup>Université de Grenoble, <sup>†</sup>LIG-LAFMIA labs, <sup>‡</sup>Centre National de la Recherche Scientifique,  
<sup>§</sup>Grenoble Institute of Technology, Saint Martin d'Hères, France  
 {firstName.lastName}@imag.fr  
<sup>¶</sup>Instituto Tecnológico y de Estudios Superiores de Monterrey,  
 Campus Ciudad de México, México, México  
 ralozano@itesm.mx

**Abstract**—The use of scalable and heterogeneous data stores within a single system is gradually becoming a common practice in application development. Modern applications tend to rely on a polyglot approach to persistence, where traditional databases, non-relational data stores, and scalable systems associated to the emerging NewSQL movement, are used simultaneously. However, considering that a large number of these systems rely on schema-less data models, developers in charge of maintaining applications that depend on these data stores need to manually analyze the application source code, in order to discover their data schemas. To help overcome this situation, in this paper we demonstrate ExSchema, a tool for the automatic discovery of external data schemas, directly from the source code of polyglot persistence applications. ExSchema is available as an Eclipse plugin, and can be easily included within existing development environments. Our tool can also be integrated with Git repositories, in order to analyze the evolution of the schemas used by an application over a period of time. In particular, this paper presents a demonstration scenario where the schemas from a system that relies on multiple document, graph, relational and column-family data stores, are discovered from the application source code.

## I. INTRODUCTION

Reverse engineering of existing databases into functional and technical specifications is a key database management process, usually provided by database reverse engineering tools [1], [2]. Although the general processes and techniques to achieve this goal are still based on classic methodologies [2], the increasing adoption of scalable data stores, such as those associated to the NoSQL [3] and NewSQL<sup>1</sup> [4] movements, promote a revision of the implementation details of such tools. In particular, these scalable data stores usually provide *schema-less* data models, and implement low-level application programming interfaces (APIs) to manage and query their data [3], [4], [5]. Furthermore, these APIs are generally non-standard, and can also be incompatible across different data store implementations [6], [3].

Considering that schema-less data stores do not enforce any particular schema and generally lack a data dictionary from where to recover the structure of the entities that they manage [5], software developers need to analyze the application source code (including field names, types, etc.) of systems that rely on these data stores, in order to discover and maintain their

external data schemas [2]. Essentially, we consider that the data schemas are shifted from the database to the application source code.

Having the data schemas as part of the application code can lead to maintenance issues. For instance, developers have to manually analyze the full source code in order to effectively understand the data model used by these applications. This can be an error-prone activity, due to the combination of different programming styles, APIs and development environments.

To help overcome this situation, in this paper we demonstrate ExSchema<sup>2</sup>, a tool for the automatic discovery of external data schemas, directly from the source code of polyglot persistence applications. Our tool currently supports Java APIs for *relational*, *graph*, *document* and *column-family* data stores, and is available as an Eclipse<sup>3</sup> plugin for an easy integration with existing development environments. Additionally, ExSchema can be integrated with Git repositories, in order to analyze the evolution of the schemas used by an application over a period of time.

The remainder of this paper is organized as follows. Section II describes the motivation for our tool. Section III provides an overview of ExSchema. Section IV outlines a demonstration scenario. Section V describes implementation challenges. Section VI outlines current limitations. Section VII describes related work. Finally, conclusions and future challenges are discussed in section VIII.

## II. MOTIVATION

As part of the emerging *polyglot persistence* movement [5], the simultaneous use of multiple scalable SQL, NoSQL and NewSQL data stores is gradually becoming a common practice in modern application development [4], [5]. Nonetheless, the combination of these heterogeneous data stores, flexible schemas and non-standard APIs, represent an added complexity for application developers.

For instance, considering that the schemas used by these applications are spread across multiple data stores, each of which possibly relies on distinct data models (such as key-value, document, graph, etc.), developers must be familiar with

<sup>1</sup>A marketplace push to support scalable SQL databases [4]

<sup>2</sup><http://code.google.com/p/exschema/>

<sup>3</sup><http://www.eclipse.org/>

a high number of implementation details, in order to effectively work with and maintain their data entities.

Consider for example the way that a sample application that relies on Neo4j [5] (*graph* data store) defines nodes and relationships between them (Fig. 1). In this case, developers rely on Neo4j’s native API and use a combination of variable declarations and update methods to specify graph nodes and relationships. However, within the same application, the source code that interacts with other data stores differs. For example, Fig. 2 depicts the definition of relational and document entities, by using variable declarations and annotations of the Java Persistence API [7] (*relational* databases) and Spring Data MongoDB project [7] (*document* data store).

```
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Relationship;

class ActorImpl implements Actor {
    private static final String NAME_PROPERTY = "name";

    private final Node underlyingNode;

    public void setName( final String name ) {
        underlyingNode.setProperty( NAME_PROPERTY, name );
    }

    public Role createRole( final Actor actor, final Movie movie, final String roleName ) {
        Node actorNode = ((ActorImpl) actor).getUnderlyingNode();
        Node movieNode = ((MovieImpl) movie).getUnderlyingNode();
        Relationship rel = actorNode.createRelationshipTo( movieNode, RelTypes.ACTS_IN );
    }
}
```

Fig. 1. Variables/Methods - Graph data store

```
import javax.persistence.Entity;
import org.springframework.data.mongodb.crossstore.RelatedDocument;

@Entity
public class Customer {

    @Id private Long id;
    private String firstName;
    private String lastName;

    @RelatedDocument
    private SurveyInfo surveyInfo;
}
```

Fig. 2. Annotations/Variables - Relational and Document data stores

Future developers in charge of maintaining this sample application would require to be highly familiar with the low-level APIs of the data stores that they use, in order to retrieve the implicit data schemas of the application. The objective of our tool is to automate this source code analysis so that developers can easily have a high-level view of the data entities (external data schemas) that their application relies upon.

### III. OVERVIEW AND ARCHITECTURE

This section overviews the architecture and the schema discovery mechanisms provided by ExSchema. As shown in Fig. 3, our tool is composed of five main elements, including the representation of a meta-layer, and a set of source code

analyzers. These source code analyzers focus on the declaration of variables, the invocation of update methods, the use of standard data layer patterns (particularly Java Persistence API (JPA) repositories), and finally, the use of annotations in Java applications. Selection of these source code analyzers was based on the examination of the test applications that each of the supported data stores provides.

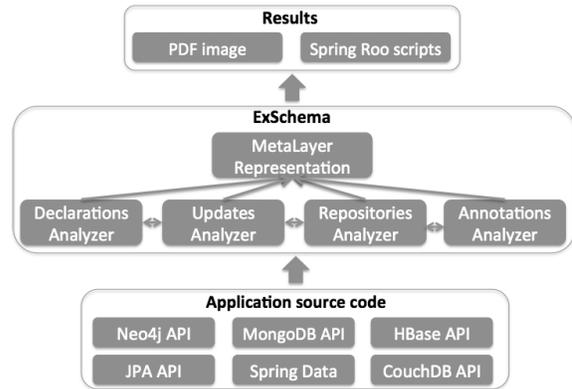


Fig. 3. ExSchema Architecture

As depicted in Fig. 3, our tool receives application source code as input, containing invocations to the APIs of one or more data stores. The ExSchema analyzers examine this application source code, and share their analysis results between each other. For example, in order to identify schema update methods, we first identify the declaration of variables. The schema fragments recovered by the code analyzers are grouped together, according to the data models supported by our tool, and by extending the translation mechanisms detailed in [6], with the identification of relationships between graph entities.

The discovered schemas are represented using a set of meta-layer constructs (Fig. 4), that will be explained next, and finally, this meta-layer representation is transformed into a PDF image and a set of Spring Roo scripts [8]. This means that if, for example, the application under analysis relies on graph and document data stores, our tool will generate two schemas, one for each data model. Both schemas will be depicted in a single PDF image, and two Spring Roo scripts will be generated, one for each schema.

We rely on Spring Roo because it allows developers to easily regenerate the Java source code that interacts with the entities contained in the discovered schemas. The code generated by Spring Roo relies on the Spring Data projects [7] to manage multiple data stores. Developers can take advantage of this feature to avoid using non-standard data stores APIs.

In order to reduce complexity and promote compatibility with other research tools, our meta-layer descriptions are based on the model proposed by Atzeni et al. in [6]. This meta-model relies on three main constructs (*Structs*, *Sets* and *Attributes*), and was originally designed to represent key-value, document and column-family models [6]. We extend this meta-model with an additional *Relationship* construct (Fig. 4), to effectively represent graph models.

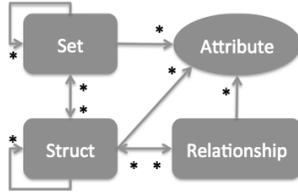


Fig. 4. ExSchema MetaLayer

ExSchema currently focuses on the analysis of Java source code and supports the following data stores APIs:

- *Graph data stores*: Neo4j [5], Spring Data Neo4j [7].
- *Document data stores*: MongoDB [5], Spring Data MongoDB [7], CouchDB4J<sup>4</sup>.
- *Column-Family data stores*: HBase [5], Spring Data Hadoop [7].
- *Relational data stores*: Spring Data JPA [7].

Finally, ExSchema supports the generation of Spring Roo scripts for *relational*, *document* and *graph* data stores.

#### IV. DEMONSTRATION SCENARIO

The demonstration will be based on the set of applications that we have used to test and validate ExSchema. This set of 16 applications is composed of the sample scenario described in [6], the test systems that each of the supported projects provide, examples from representative books, and finally, an industrial application designed to automatically validate digital tax receipts<sup>5</sup>. The open source code of these test applications is available in the project website<sup>6</sup>.

The first sample scenario is an extension of the application described by Atzeni et al. in [6], which is a simplified version of Twitter that relies on HBase, Neo4j, MongoDB and JPA for persistence management. We developed this application in four stages, each marked by a tag in our Git repository. In the first stage, we defined a document structure to store *Tweets*, using MongoDB's native Java API. In the second stage we added the notion of *Users*, using Neo4j's native Java API. In the third stage we defined a way to store *Users information*, by relying on JPA. Finally, on the fourth stage we refactored the *Tweets* structure to rely on HBase instead of MongoDB.

By using ExSchema, we will demonstrate how we can discover the data schemas that were available at each stage of the application. We can identify at which point new entities were added to the application, or, for example, in the case of the *Tweet* entity, how its structure changed over time. Moreover, we could also execute the Spring Roo scripts generated by our tool, so as to rely on the Spring Data projects instead of the native APIs that were used in the original development.

The following listing depicts one of the Spring Roo scripts that ExSchema will generate from the second version of the application. The script contains the definition of the *User* entity as a Neo4j node, along with its attributes and relationships.

Finally, Fig. 5 shows the schemas that our tool will recover from the last version of the application. The PDF file contains the representations of *Tweets*, *Users* and *Users information*.

```

project --topLevelPackage exschema.projectTwitter
graph setup --provider Neo4j
entity graph --class ~.domain.User
repository graph --interface repository.UserRepository
relationship entity graph --class ~.domain.Relation1
--type FOLLOWS --from ~.domain.User --to ~.domain.User
field string --fieldName userId --class ~.domain.User
field string --fieldName userName --class ~.domain.User
field string --fieldName password --class ~.domain.User
  
```

During the demonstration we will present similar applications that rely on additional APIs, including the Spring Data projects and CouchDB4J.

#### V. IMPLEMENTATION

The implementation of our tool focuses on two main challenges: (i) seamless integration with existing development environments; and, (ii) simple extension mechanism to add support to multiple data stores. In particular, ExSchema is currently implemented as an Eclipse plugin, to facilitate integration with existing Java development environments. The ExSchema plugin relies on the Eclipse Java Development Tools (JDT) to analyze source code, by using the Eclipse Abstract Syntax Tree (AST) to traverse the compilation units contained in Java projects.

The core AST analysis is provided by a generic infrastructure associated to the main types of code analyzers provided by our tool, that is, declarations, updates, repositories, and Java annotations. ExSchema code analyzers can then reuse this generic infrastructure to support specific APIs.

In order to analyze the evolution of the schemas used by an application, we provide an integration with the Git source code management system<sup>7</sup>. In this way, developers can analyze different versions of their application, represented by *tags* and *branches* in the Git repository, and then compare the results that ExSchema generates for each of them.

#### VI. LIMITATIONS

The current implementation of ExSchema supports a particular set of programming styles found in the sample applications provided by the supported data stores. In particular, code analyzers heavily rely on the project structure, *update* methods and the declaration of local variables. At the moment, we do not consider *queries* and *get* operations as part of our analysis.

Our tool provides limited support for associations between entities that belong to different data models, besides Neo4j's relationships and MongoDB cross-store support [7]. Finally, since the current implementation of ExSchema relies on the analysis of the Eclipse AST, the applications under analysis must not have compilation errors when accessed by Eclipse.

We validated ExSchema by relying on a set of open source systems, including the test projects provided by the APIs that we currently support. These test applications rely on a particular set of programming styles, in particular, persistence

<sup>4</sup><https://github.com/mbreese/couchdb4j/>

<sup>5</sup><http://www.indvalid.com/>

<sup>6</sup><https://code.google.com/p/exschema/#Tests>

<sup>7</sup><http://git-scm.com/>

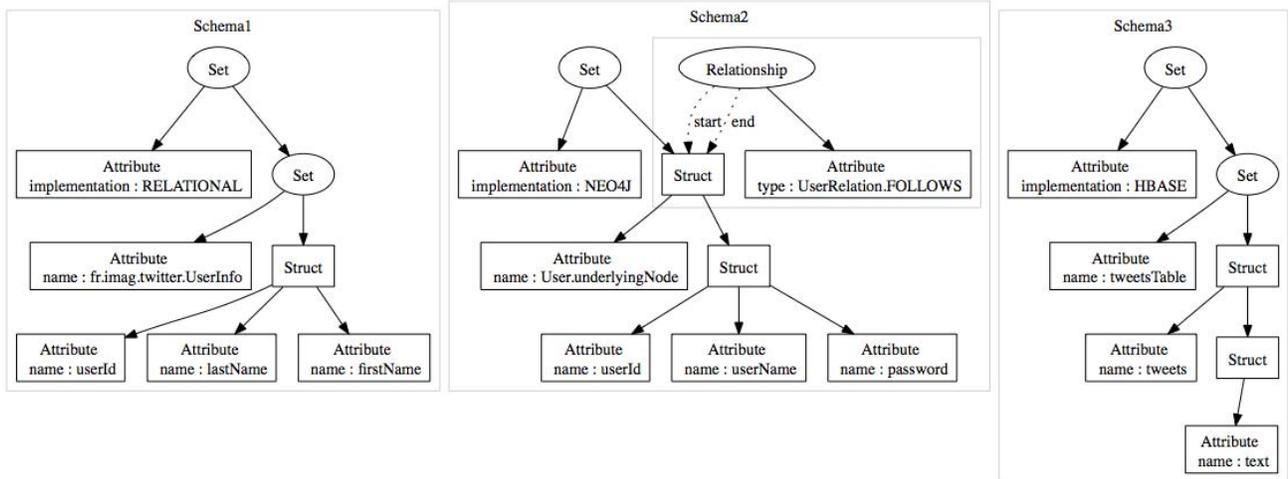


Fig. 5. Resulting PDF image

operations related to one data store occur in the context of only one class. As a consequence, our tool will not always be able to correctly discover the data schemas of real-world applications that rely on a wider set of programming styles.

## VII. RELATED WORK

Database reverse engineering (DBRE) is a well studied subject, associated to proven methodologies and techniques [2]. Traditionally, much of the research conducted in this field has focused on the relational model. However, progressive adoption of alternative data models also motivates an update of the tools implementing DBRE techniques.

For example, in [9], Cabot et al. describe an approach to recover schemas from object-relational databases. The main objective of this approach is to recover conceptual schemas, represented as UML diagrams, based on the analysis of the data dictionary of database management systems (DBMS). In comparison, our tool relies on source code analysis from applications that use scalable data stores. This is due to the fact that modern DBMS tend to rely on schema-less models, and may not always provide reliable data dictionaries [5].

A rule-based transformation language (Schemol) intended to extract models from web 2.0 databases is proposed by Díaz et al. in [10]. The model analysis is performed directly on the database content and schemas, assuming a relational data model, instead of relying on the source code of the applications that use these web 2.0 databases (such as wikis, blogs and social networks). The resulting models are expressed by using the Eclipse Modeling Framework<sup>8</sup>, instead of a custom modeling notation.

The schemas discovered by our tool are represented using an extension of the meta-model constructs presented by Atzeni et al. in [6]. This work proposes a common programming interface to multiple NoSQL systems, based on a meta-modeling approach. However, instead of hiding the specific details of

the data stores APIs used by an application, we encourage an heterogeneous API environment, and then automate the recovery of the schemas, using a common notation.

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper we demonstrated ExSchema, a tool that enables the automatic discovery of schemas from polyglot persistence applications. The discovery mechanism is based on source code analysis techniques, particularly on API usage and on the analysis of standard data layer patterns. We also described a demonstration scenario, where the schemas from an application that relies on different data stores are discovered directly from its source code.

For our future work, we would like to support additional data stores and improve the discovery techniques implemented by our tool.

## REFERENCES

- [1] J. Hainaut, "Legacy and future of data reverse engineering," in *WCRE*, p. 4, IEEE, 2009.
- [2] J. Hainaut, C. Tonneau, M. Joris, and M. Chandonel, "Transformation-based Database Reverse Engineering," in *ER*, vol. 823 of *Lecture Notes in Computer Science*, pp. 364–375, Springer, 1994.
- [3] R. Cattell, "Scalable SQL and NoSQL Data Stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2010.
- [4] J. Corbett, J. Dean, M. Epstein, A. Fikes, et al., "Spanner: Google's globally-distributed database," in *OSDI*, USENIX Association, 2012.
- [5] M. Fowler and P. Sadalage, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.
- [6] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to non-relational database systems: the SOS platform," in *CAiSE*, vol. 7328 of *Lecture Notes in Computer Science*, pp. 160–174, Springer, 2012.
- [7] M. Pollack, O. Gierke, T. Risberg, J. Brisbin, and M. Hunger, *Spring Data - Modern Data Access for Enterprise Java*. O'Reilly Media, 2012.
- [8] J. Long and S. Mayzak, *Getting Started with Roo - Rapid Application Development for Java and Spring*. O'Reilly Media, 2011.
- [9] J. Cabot, C. Gómez, E. Planas, and M. E. Rodríguez, "Reverse engineering of object-relational database schemas," in *JISBD*, pp. 241–252, 2008.
- [10] O. Díaz, G. Puente, J. L. Cánovas Izquierdo, and J. García Molina, "Harvesting models from web 2.0 databases," *Software & Systems Modeling*, vol. 12, pp. 15–34, 2013.

<sup>8</sup><http://www.eclipse.org/modeling/emf/>

# A Visualization Tool for Reverse-engineering of Complex Component Applications

Lukas Holy, Jaroslav Snajberk, Premek Brada and Kamil Jezek  
 Department of Computer Science and Engineering, Faculty of Applied Sciences,  
 University of West Bohemia, Pilsen, Czech Republic  
 {lholy, snajberk, brada, kjezek}@kiv.zcu.cz

**Abstract**—Nowadays, component applications can contain thousands of components whose structure is difficult to understand. As a solution, we proposed a visualization technique that removes large part of connections from component binding diagrams. This technique uses a separated components area to display components with a big amount of connections detached from the main diagram. In this area, component interfaces are shown clustered instead of showing them all. Benefit of this approach is improvement of application understanding by reducing the diagram’s visual clutter during its reverse engineering. In this work, we present implementation of the technique in a form of a tool, called CoCA-Ex. CoCA-Ex is a publicly accessible web application and a reverse-engineering solution for various component systems. The tool is built on modern technologies such as HTML5 and JavaScript and has Java EE server backend.

**Keywords**—software visualization; component; visual clutter;

## I. INTRODUCTION

Software applications become more and more complex and their structure is difficult to handle. Although there are lots of tools helping with application reverse engineering, the tools are still limited in helping human understanding of the structure.

Software components are one of the ways to handle this complexity as they encapsulate functionality to (ideally black-box) components. However, applications can easily consist of hundreds or thousands of the components. It is therefore difficult to explore the structure of component bindings and create a mental model of the whole system.

One of the ways to get an insight into a component application structure can be UML component diagrams. However, when the diagrams become large, there are many problems with exploring it. One of them is a contradictory need of providing enough details and showing the overall structure at the same time. Another question is how to reduce visual clutter caused by the large number of elements and connections between them. This visual clutter makes the tracing of dependencies difficult and hinders orientation in the diagram. In this tool, we present implementation helps to reduce visual clutter in UML component diagrams and help user to form clusters of related components.

### A. Structure of the Paper

In the following section, a problem of visual clutter is defined first. After that, a related work is described in Section III. Then, in Sections IV and V a technique called SeCo (Separated Components) and its implementation called CoCA-Ex (Complex Component Applications Explorer) are

presented. This technique helps to reduce the visual clutter in large graphs. After that, we briefly discuss extra-functional properties of CoCA-Ex in Section V-A.

## II. PROBLEM DEFINITION

Developers often face multiple challenges in the reverse engineering of large applications and their visualization. While visualizing diagrams of such applications, there are usually problems like difficult orientation, limited amount of elements fitting to the screen when their details are shown, insufficient details when showing overview, or the visual clutter [1].

This paper focuses on the CoCA-Ex tool which copes with the problem of highly connected components and provides component clusters visualization.

The problem of highly connected components contributes to the visual clutter of the displayed area. Very often, only a small amount of components is connected to a large number of other components. The result of visualizing such a component is illustrated in Figure 1, where part of an Eclipse structure is shown in the Plugin Dependency Visualization tool<sup>1</sup>. Such components are often, among developers, informally called “God Objects”. It is difficult to trace the connections in the surrounding area of these objects.

A component cluster visualization is a process to form and work with visual component groups, which usually represent one feature or a logical unit of the system. When clusters are not used, the screen space, which is one of the essential resources in the visualization, is rapidly exhausted

## III. RELATED WORK

Visual clutter can be reduced by many techniques. The whole taxonomy of these techniques has been described by Ellis and Dix in [2]. A short description of those techniques related to our work is provided.

The clutter caused by the lines is often reduced by edge bundling [3]. Although this approach reduces the clutter, it makes it difficult to trace the dependencies between connected nodes leading through the edge bundles.

The visual clutter can be also lowered by using node clustering, where one cluster usually represents multiple nodes. Thus the number of nodes in the whole diagram is lowered, though the connections among components are usually still

<sup>1</sup><http://www.eclipse.org/pde/incubator/dependency-visualization/>



Fig. 1. Visual clutter caused by highly connected component

present. Clusters can either be marked manually, in an automated way or by a combination of those approaches.

Another influencing factor is the chosen layout algorithm, which can ease orientation.

Existing tools related to this domain were described in our previous paper [4]. One of the most related tool is SoftVision. It is a software visualization framework described in [5] which is able to interactively explore relations between data structures. It is a desktop application which offers a generic interface for describing a particular parser.

In the following section, our visual clutter reduction approach is described.

#### IV. THE SEPARATED COMPONENTS AREA TECHNIQUE

A technique we have proposed reduces the visual clutter by removing components with a large number of connections from the main diagram into a so called *separated components area* (abbreviated as SeCo) placed on the border of a window (the right sidebar in Figure 2). When a user moves components from the main diagram to this area, the lines between these components and remaining components are elided. This essentially marks a component as a “familiar one”. Once component removed, the user may continue getting familiar with the rest of the system.

SeCo consists of a list of items. Each item consists of clustered interfaces (indicated with the mark (T) in Figure 2), components (U) and one corresponding symbol (S). Interfaces are clustered into two sets (T): all provided interfaces (displayed as “lollipops”) and all required interfaces (displayed as “sockets”). Numbers inside the clustered interfaces denote the number of elements clustered in the respective symbol.

The purpose of symbols is to create a clear and easily recognizable key which uniquely identifies items within SeCo. These symbols can be used in the diagram area to represent connections between a given component and the corresponding item placed in SeCo. They are shown as small rectangles neighboring to the displayed components (K) and containing the symbol, which corresponds to the connected item (S). These symbols have graphics, which may humans easily remember, to help build a mental model.

Often, a particular functionality of the system is implemented by several components. Such components are good candidates to be place to SeCo (M), when their functionality is used by a large number of other components. Then, graph’s visual clutter is usually considerably decreased.

One of the cases may be a component implementing a logger. Such a component is probably used by most of

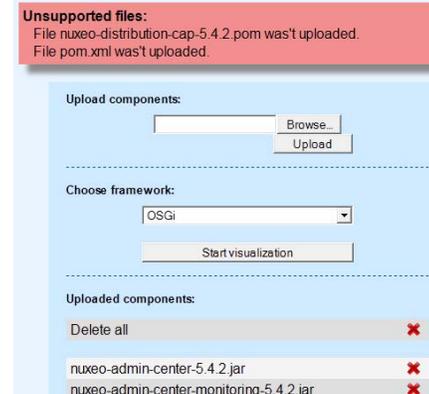


Fig. 3. Complex Component Application Explorer tool demonstration upload screen

components in the system and its displacement reduces the graph complexity.

We assume both automatic and manual component selection to be used. In the automatic case, all components with the number of connections overcoming a certain threshold are displaced. In the manual use, a user drags-and-drops the components from the main graph to the SeCo.

#### V. THE SEPARATED COMPONENTS AREA TECHNIQUE’S IMPLEMENTATION

SeCo technique implementation is called CoCA-Ex. CoCA-Ex works on the ComAV platform (Component Application Visualizer) [6], so it benefits from ComAV’s reverse engineering and management features. ComAV can automatically reverse engineer the whole component-based applications from one of the supported component models. Further component models can be easily added using an extension mechanism offered by RCP (Rich Client Platform). ComAV is also able to add other visualization styles. It means that CoCA-Ex is only one way of how structures of applications can be visualized on ComAV platform.

In CoCA-Ex, a user starts the visualization process by picking desired components from the local machine and uploading them to the server, as shown in Figure 3. Unsupported file types are automatically ignored, as indicated by a red sign. The ComAV platform creates the model of the application and the CoCA-Ex shows the application diagram in the web page. The demonstration of the CoCA-Ex’s GUI is shown in Figure 2. It demonstrates the features of the tool on a reverse engineered component diagram of a OpenWMS application, a modern warehouse management systems which is based on components.

Figure 4 shows a data flow between ComAV and CoCA-Ex and CoCA-Ex’s client and server parts. While CoCA-Ex server emits raw component data to ComAV and receives reverse-engineered models, the CoCA-Ex client provides model’s visualization and user interaction.

CoCA-Ex uses servlets from the enterprise Java technology as the back-end technology. Servlets are used mainly because of the Java implementation of the ComAV tool. HTML5,

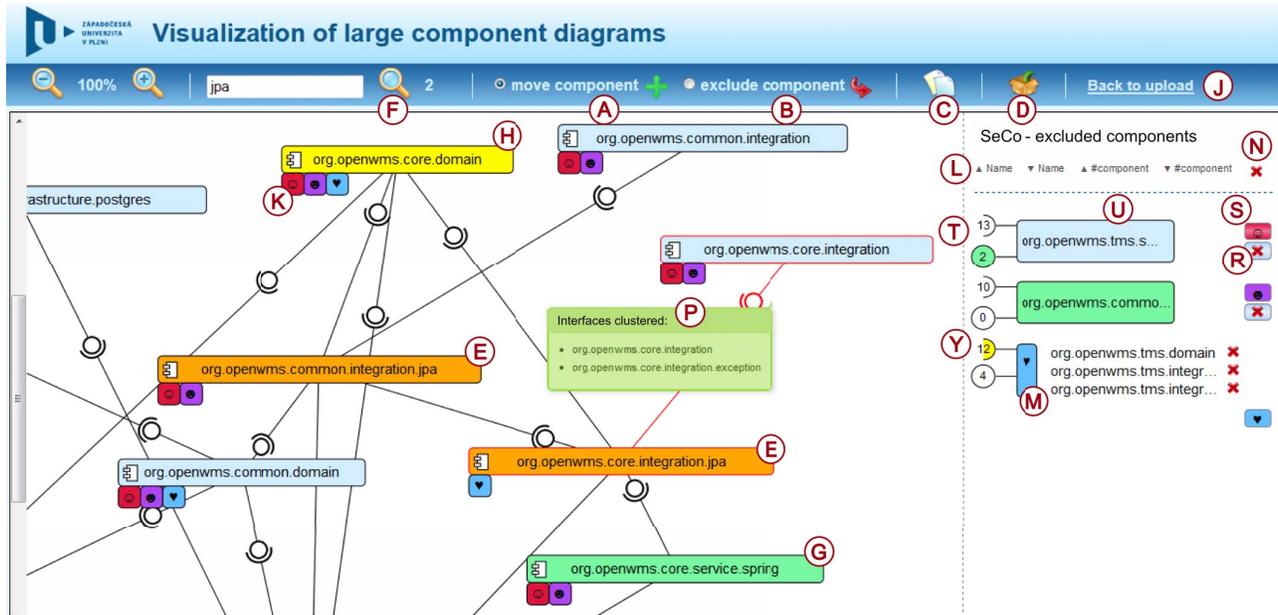


Fig. 2. Complex Component Application Explorer tool demonstration

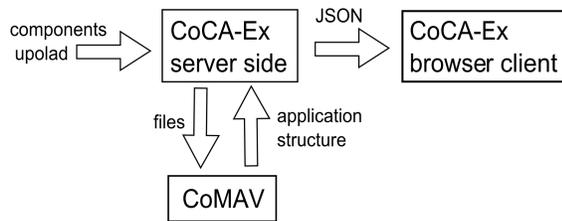


Fig. 4. CoCA-Ex architecture

JavaScript, jQuery framework and CSS3 were used for the front-end. Canvas and SVG elements from HTML5 are used to represent the nodes of the diagram. Although the HTML5 technology is still not fully supported by all main browsers, its current state is sufficient for CoCA-Ex purposes. Also desired features such as SVG support or Canvas are likely to be stable in the near future. We chose these technologies because they are well-known and widely supported, which will ease further improvements. We also tested several graphical frameworks, but we concluded that their customization would be difficult because most of our present and future features require custom implementation. That is the reason why we implemented the GUI from scratch.

The tool provides standard features such as panning and zooming. There are two modes of manipulating the components with appropriate icons in the toolbar. First mode is for moving components (A) where the user can manually adjust the layout of the diagram. Second mode (B) serves for removing components from the diagram area to the SeCo area simply by clicking on the desired components, which should be removed. Last two icons in the toolbar serve for the automatic removal of a configured amount of components from the diagram to the SeCo area. The tool is currently configured to remove 15% of most connected components, but

more sophisticated algorithms are subject of a future work. The icon (C) is used for removing these components and adding them to the SeCo area as individual items. The next icon (D) creates one group for all of them.

The tool also offers to change the set of components the user works with. For this purpose the link "Back to upload" (J) is in the toolbar. It is also possible to remove all components from SeCo area back to the diagram area by using the red cross icon (N). There is also a possibility to sort the components shown in SeCo (L). Currently, there is a possibility to sort components by their names or their amount in a group. Both mentioned sorting possibilities can be ascending or descending.

CoCA-Ex offers a fulltext search by components' names. In Figure 2, one can see the search result for a string "jpa". Two components in the diagram contain this word as indicated by the number two (F). Components matching the search are highlighted by an orange color (E).

If one clicks to the required interfaces of a component in SeCo, these interfaces and connected components become highlighted by a yellow color. This feature is demonstrated on a dependency between an *org.openwms.core.domain* component (H) and required interfaces (Y) of a blue group with the heart symbol (M). For the provided interfaces, a green color is used for highlighting. It can be seen between provided interfaces of the first component in SeCo (T) and *org.openwms.core.service.spring* component (G) in the diagram area.

For several components from the SeCo area (those with symbols' background highlighted by different colors (S)) there are delegates shown in the diagram area, e.g., (K). One can also see a group consisting of three components (M). For inspecting interfaces, the tool offers highlighting of a connection by a red color and showing the interfaces involved in the connection (P), as shown in the green tooltip. Each individual component

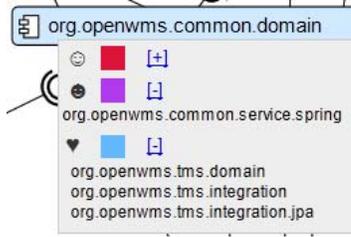


Fig. 5. Adding components from diagram to SeCo groups

shown in SeCo has its own button (R) to remove it back to its original position in the diagram area.

There is also possibility to add desired components from diagram area to existing items in SeCo by a context menu, as shown in Figure 5. The context menu will appear after a right mouse click on any components in the diagram area. It contains symbols and colors of all items included in SeCo. There is also possibility to explore each of these items in the context menu by using the plus symbol. After such expanding, all components included in the corresponding SeCo area item are listed.

#### A. Extra-functional properties of CoCA-Ex

CoCA-Ex is able to work with thousands of elements on a usual desktop computer without uncomfortable delays. The longest delay in the reverse-engineering process is the initial upload and the diagram creation. It is caused by the fact that applications can have hundreds of megabytes and thus their uploading and processing can take some time, depending on the network connection speed and computer performance. For instance, diagram containing 914 components, which have in total 113 MB, took to show 10 seconds<sup>2</sup>. Another CoCA-EX property is its compatibility with browsers. Since the tool is written in new technologies, which are not fully supported in all browsers, the application currently works well in Firefox 21. We are working on compatibility with Chrome.

#### B. Deployment and Availability

The CoCA-Ex application is available online<sup>3</sup> for testing. Please note that due to new technologies, only Firefox 21 is officially supported<sup>4</sup>. For the demonstration purposes, we provide a corpus of test data files<sup>5</sup> to be uploaded to the CoCA-Ex server. The corpus needs to be extracted and components (.jar files) uploaded. The tool can be also downloaded as a Java EE WAR file<sup>6</sup> and deployed to a Java EE server (e.g. Apache Tomcat<sup>7</sup>). In a case of own installation, it is necessary to configure the storageLocation attribute for uploaded files in a configuration file<sup>8</sup>. All necessary information and screencasts are described on the information webpage<sup>9</sup>.

<sup>2</sup>System configuration: Intel(R) Core(TM) i5-2300 @ 2.80 Ghz, Windows 7 64-bits, Firefox 17.0

<sup>3</sup><http://relisa.kiv.zcu.cz:8083/cocaex/>

<sup>4</sup>[http://portableapps.com/apps/internet/firefox\\_portable/localization](http://portableapps.com/apps/internet/firefox_portable/localization)

<sup>5</sup>[http://home.zcu.cz/~lholy/visualization/demo\\_data.zip](http://home.zcu.cz/~lholy/visualization/demo_data.zip)

<sup>6</sup><http://home.zcu.cz/~lholy/visualization/cocaex.war>

<sup>7</sup><http://home.zcu.cz/~lholy/visualization/apache-tomcat-7.0.27.exe>

<sup>8</sup>[cocaex.war/WEB-INF/configuration.properties](http://home.zcu.cz/~lholy/visualization/cocaex.war/WEB-INF/configuration.properties)

<sup>9</sup><http://www.kiv.zcu.cz/research/groups/dss/projects/cocaex>

## VI. CONCLUSION AND FUTURE WORK

In this paper, the CoCA-Ex tool for maintenance of component applications was described. The tool helps to reduce the amount of lines in UML component diagrams of large applications by removing the selected components from the main diagram area. The core concept of the tool is a Separated Components area where the selected components are detached. Symbolic delegates replace lines from the original diagram. In a typical scenario, developers remove highly connected components from the main diagram to get the insight of important parts faster.

This tool is useful in the reverse engineering process when the user is interactively getting familiar with a complex component system. It helps with creating the mental model of the application by easing the process of clusters creation. While the reverse-engineering process is delegated to the ComAV platform, the tool presented here concerns on a user interaction when maintaining a complex component applications and getting familiar with its structure.

Preliminary evaluation shows that the presented ideas are helpful in a large graph visualization, where one suffers from visual clutter caused by the large number of connection lines. In one experiment<sup>10</sup>, only 7 out of 202 components have been removed from the diagram area leading to 65% connection lines reduction. It showed up that by using the proposed technique, a significant visual clutter reduction may be achieved.

We are currently implementing a layout algorithms integration and we will continue work on clustering techniques. We have chosen force-directed algorithms to be used in the application after a short survey. These features should let the user get the insight faster. We also plan to evaluate above mentioned ideas by a case or user study.

## ACKNOWLEDGMENT

The work was supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems. Authors would like to thank Jindra Pavlikova, Daniel Bures and Stepan Kubasek for the work on the implementation.

## REFERENCES

- [1] R. Rosenholtz, Y. Li, and L. Nakano, "Measuring visual clutter," *Journal of Vision*, vol. 7, no. 2, August 2007.
- [2] G. Ellis and A. Dix, "A taxonomy of clutter reduction for information visualisation," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 6, pp. 1216–1223, nov.-dec. 2007.
- [3] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, Sep. 2006.
- [4] L. Holy, J. Snajberk, and P. Brada, "Evaluating component architecture visualization tools - criteria and case study," 2012.
- [5] A. Telea and L. Voinea, "A framework for interactive visualization of component-based software," in *Proceedings of the 30th EUROMICRO Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 567–574. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1018420.1019732>
- [6] J. Snajberk, L. Holy, and P. Brada, "Comav - a component application visualisation tool," in *Proceedings of International Conference on Information Visualization Theory and Applications*. SciTePress, 2012.

<sup>10</sup>reverse engineering of Nuxeo application

# Interactive Exploration of Collaborative Software-Development Data

Eleni Stroulia, Isaac Matichuk and Fabio Rocha  
 Department of Computing Science  
 University of Alberta, Edmonton, Canada  
 Email: {stroulia, isaacm, fabiorocha}@ualberta.ca

Ken Bauer  
 Department of Computing Science  
 Tecnológico de Monterrey, Zapopan, Jalisco, México  
 Email: kenbauer@itesm.mx

**Abstract**—Modern collaborative software-development tools generate a rich data record, over the lifecycle of the project, which can be analyzed to provide team members and managers with insights into the performance and contributions of individual members and the overall team dynamic. This data can be analyzed from different perspectives, sliced and diced across different dimensions, and visualized in different ways. Frequently the most useful analysis depends on the actual data, which makes the design of single authoritative visualization a challenge. In this paper we describe an analysis and visualization tool that supports the flexible run-time mapping of such a data record to a number of alternative visualizations. We have used our framework to analyze and gain an understanding of how individuals work within their teams and how teams differ in their work on these term projects.

URL (screencast and system): <http://hypatia.cs.ualberta.ca/collab/Visualizations/Dashboard/main/splash/splash.html>

## I. THE PROBLEM CONTEXT

Collaborative software-engineering projects rely on the use of tools for the coordination of the work among the team members and the management of their work products. Systems such as CVS, Subversion and GitHub enable team members to work in an asynchronous yet coordinated manner, to share their work products, and even to backtrack over erroneous commits to their codebase. Issue tracking systems enable developers to inform each other about their activities, and to prioritize and track progress on pending milestones and outstanding bugs. Mailing lists and wikis offer more flexible channels through which to communicate about schedules, pending work and documentation, and to broadcast information to all team members. Instant messengers and IRC chat channels provide a real-time communication when developers are working synchronously but not in the same physical location.

The artifacts produced by these tools provide a rich history of the project as well as clues to the communication and development patterns of the individual developers and the team as whole. Yet understanding this collection of data is not straightforward and can be time consuming. A number of tools have been developed to analyze and visualize this data, yet more work is necessary especially when one considers the question of “understanding individual contribution and team dynamics”. This question is interesting for all software projects but especially relevant for teams of novice developers with immature work practices and varied development knowledge and skills, as is the case with academic project-based courses.

Motivated by the project-course scenario, we have developed a visual tool, the “*Collaboratorium Dashboard*”, to help instructors navigate through, and interact with, their student teams’ data through the web browser.

Aware of a number of special-purpose source-code repository visualizations, and having developed some ourselves, in this project we set out to define a general framework, for flexibly creating visualizations of any dataset. The goal of our tool is to provide a framework to easily map the data collected in modern collaborative-software development tools into configurable and interactive visualizations, through which to enable the user to explore and interact with the data. The tool handles most kinds of data collected through collaborative software-development tools (relying on a general conceptual model to which this data can be mapped) and offers a number of interactive visualizations through which to explore the development, management and communication activities of each individual developer and the team as a whole.

The rest of this paper is organized as follows. Section 2 describes the software architecture of our tool and reviews its features. Section 3 describes the different visualizations provided by our tool, through a typical usage scenario. Section 4 places this work in the context of previous related research. Section 5 draws some conclusions and outlines some avenues for future work.

## II. SOFTWARE ARCHITECTURE AND FEATURES

The *Collaboratorium Dashboard* relies on a “products vs. process vs. people” [1] conceptual model of collaborative software development. Simply put, the tool is designed to explore the information about (a) the *people* involved in the software project, (b) the *products* they develop in order to complete the project, (c) the *process* they follow to share tasks and manage their activities, and the relations among these three major concepts. This conceptual model, similar to PCANS [2], covers the major types of information captured by most modern collaborative software-development tools. More specifically, under the term “people” we understand the team members involved in software project; “products” include all software assets, including source code, make files, images and HTML documents, and, generally, all the artifacts required for the software system under development to work; finally, under the term “process” we consider all information relevant

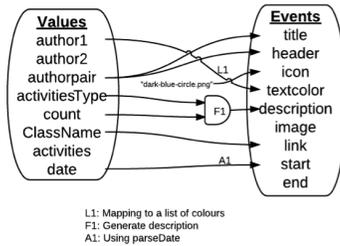
to how the team members communicate with each other and coordinate their activities, including issues managed through an issue-tracking system, implicit communications through contributions to the same artifacts and explicit communications through channels such as email, IRC and wikis. The generality of this conceptual model enables the integration of *Collaboratorium Dashboard* with any collaborative-development environment; to date, we have integrated it with IBM Jazz and the WikiDev toolkit [3] that includes an integration of SVN, Bugzilla, email and wikis.

```

{ from:values,
  to:events,
  iterate:[
    { from:authorpair, to:title },
    { from:authorpair, to:header },
    { value:dark-blue-circle.png, to:icon },
    { from:activitiesType, to:description },
    { from:ClassName, to:link },
    { from:date, to:start }
  ] }

```

(a) A strategy for data mapping



(b) A diagrammatic explanation of the strategy above

Fig. 1: Constructing a strategy

The *Collaboratorium Dashboard* software architecture involves three layers. The lower layer involves the various data sources, exposing their data (typically through REST APIs) into the *Collaboratorium Dashboard* data store. Essentially, in order to use the *Collaboratorium Dashboard* with a chosen collaborative software-development tool, one has to develop data-extraction adapters to import the data from the chosen tool to the *Collaboratorium Dashboard* repository. Once the data has been fed into the *Collaboratorium Dashboard*, the *data-mapping* layer enables a number of “strategies” for mapping a collection of JSON objects, produced by REST APIs implemented by the data store, into a different set of JSON objects understandable by the *visualizations* included in the top-layer dashboard. The various data-mapping strategies enable multiple alternative mappings of the same data to different visualizations, which tend to bring to the forefront different relations and trends. All the implemented visualizations support a number of functionalities, including data filtering and commenting on specific data points. A number of visualizations can be combined into a tabbed “dashboard”, where each tab can be laid out to include the desired number of rows and columns to accommodate the chosen visualizations. In the *Collaboratorium Dashboard* version demon-

strated in <http://hypatia.cs.ualberta.ca/collab/Visualizations/Dashboard/main/splash/splash.html> we have chosen a particular dashboard structure, based on our experimentation with a number of layouts; however, the data-mapping strategies can flexibly define any desired composition of dashboard tabs. In the rest of this section, we detail the various components of the *Collaboratorium Dashboard* tool.

### A. The Dashboard

The dashboard configuration is defined (in JSON) in terms of its *tabs* and its *preloadMap*. Each tab is given a name and the set of visualizations it includes. Each visualization is defined in by its title, a description of its purpose, and the strategy that defines how the data should be mapped to the visualization elements. The *preloadMap* object defines lists of objects of interest in the system, so that any auto-configured settings are consistent across sessions. In particular, we use this mechanism to establish a consistent mapping of objects to colours. These colours remain consistent across all visualizations and all tabs.

### B. Strategies

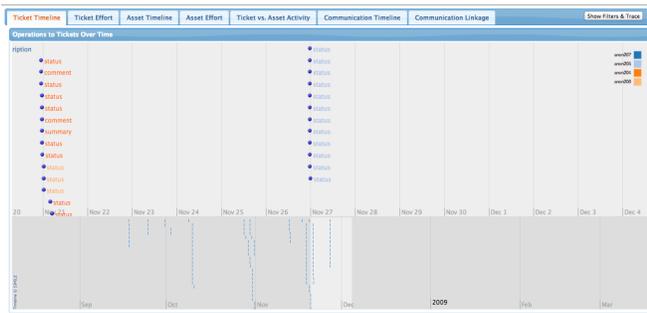
The actual mapping of the “data” of a given visualization to its “container” is defined by a *strategy* in the main configuration. A “strategy” is an object that defines how the data exposed by the data-store should be mapped to the data expected by a particular visualization. A *strategy* can be conceptualized using a diagram similar to Figure 1(b), which visually depicts the strategy described in the JSON object of Figure 1(a).

The strategy uses a simple form to describe how the JSON object exposed by the data store must be transformed to be consumed by the destination visualization. The first part instructs that a “list of values” should be mapped to a new “list of events”. The “iterate” fragment instructs how to transform every single element in values list; namely, as depicted with the arrows in Figure 1(b).

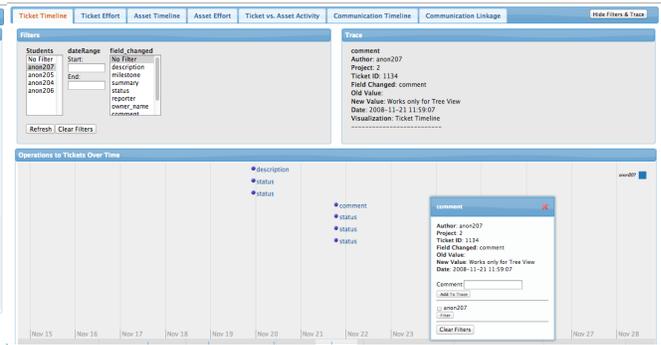
### C. Visualizations

Currently, the *Collaboratorium Dashboard* supports five different types of interactive visualizations: (a) a timeline; (b) a histogram; (c) a partition diagram; (d) a chord diagram; and (e) a force-directed graph. The implementations of these visualizations share two common features, in order to support consistency of use. As shown in Figure 2(b), they all support filtering of the displayed data through the top left window, and commenting on particular data points; all comments are appended in the trace window, shown in the top right window. The filter and trace windows can be shrunk or expanded.

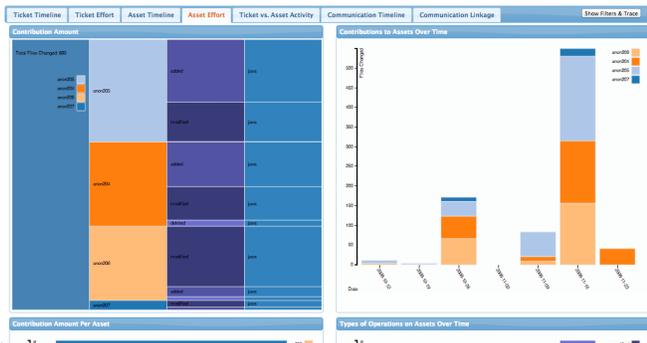
Each visualization can be used to display different datasets; for example, the dashboard in Figure 2(f) includes two histograms populated with the team’s activities on tickets and assets correspondingly. This is supported by the definition of each visualization as an object with two parameters: a *container* and its *data*. The “container” defines a “div” element on the HTML page to display the visualization; the “data” parameter is a json object containing the data to be visualized.



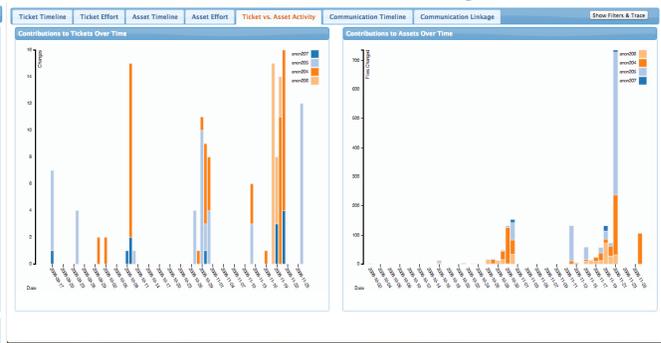
(a) team 2: few tickets, lack of coordination



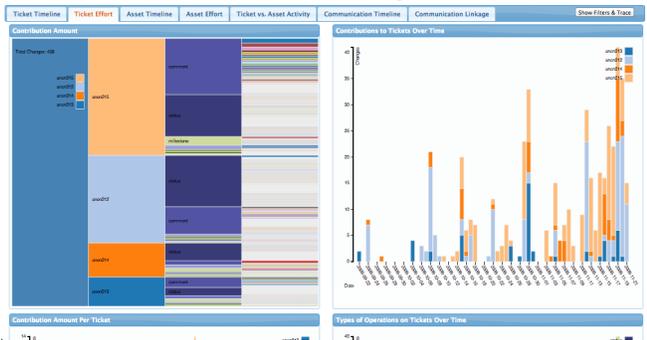
(b) team 2: almost no ticket management



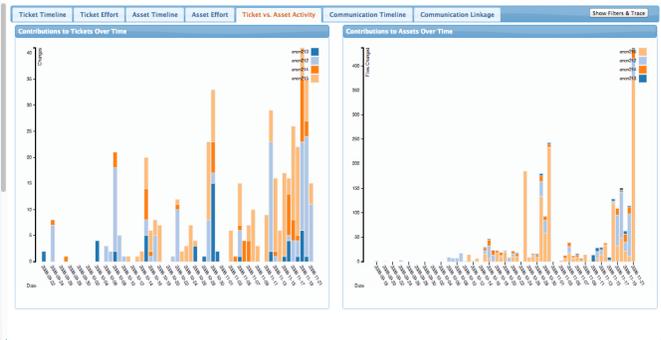
(c) team 2: a non-contributing team member



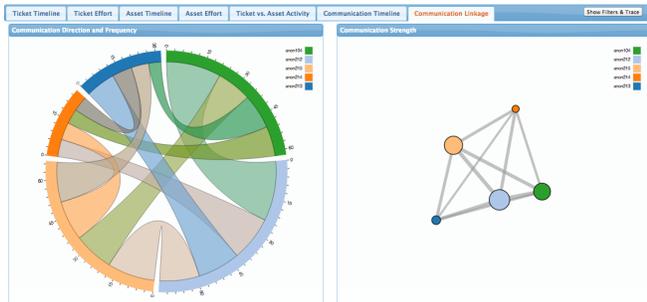
(d) team 2: lack of contribution



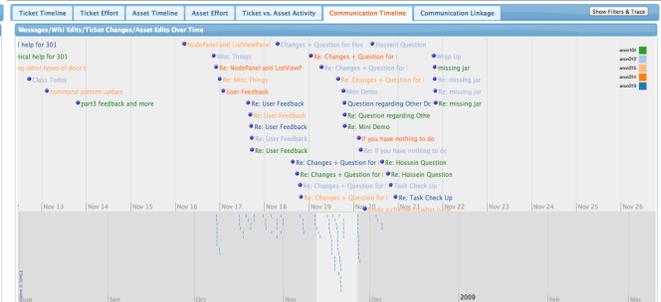
(e) team 4: substantial effort spent on ticket management



(f) team 4: consistent work products



(g) team 4: complete communication linkages



(h) team 4: project-long, continuous communication

Fig. 2: Collaboratorium Dashboard visualizations for two teams

### III. TYPICAL USAGE SCENARIO

Figure 2 contains a series of screenshots, which illustrate a typical scenario of an instructor (or a manager of multiple teams) using *Collaboratorium Dashboard* to reflect on the performance of the teams and their developers. Examining the process aspect of *team 2* in Figure 2(a) and (b), one sees that they did not manage their tasks through the ticket system; they established relatively few tickets and they made few, and infrequent, operations on them. Filtering by *anon207* in particular, one can see that he only made 12 operations on tickets over 6 days (throughout the two-month project lifecycle). The relatively low contribution of *anon207* is further established through 2(c), where one can see that he made only 37 edits to the team's source-code files (out of a total of 1729) on two weeks only. Based on these three visualizations, one can establish that this team exhibits a rather unbalanced work distribution and irregular productivity, which is frequently a cause for discontent and stress among the team members.

*Team 4* exhibits much healthier behavioral patterns. Figure 2(f) shows all team members are involved in task management (as evidenced by their contributions to ticket management over time) and in actual code development (as evidenced by their consistent contribution to products over time). Still *anon213* seems to contribute to a lesser degree, especially as far as development is concerned. Based on Figure 2(e), one can see that the team members of *team 4* were all involved in task management; it is also interesting to note that this team has started planning their tasks before they started developing. Finally, examining *team 4* from the communication perspective, in Figure 2(g) one can see that all team members appear to communicate (through email) equally with all other team members, but *anon213* and *anon214* appear to have sent fewer emails than the other three members. Finally, in Figure 2(h) one can observe that the team followed a regular and consistent pattern of communication across the project lifecycle.

### IV. RELATED TOOLS

Today, several collaborative software-development tools offer dashboards. Atlassian JIRA and GitHub, both provide tools for visualizing reports of the data they collect. However, these visualizations are tool specific and focus on the progress that the project is making and do not examine the role of individual team members, or the evolution of the individual tickets and assets. Our own team has developed two different visualization tools, based on the WikiDev and WikiDev 2.0 tools [3] that adopted a wiki as the central platform for integrating information about the various artifacts of interest (e.g., source code in SVN and bug entries in Bugzilla). These visualizations were fixed, offering a pre-defined set of mappings between data and views. Tesseract [4] is similar in that it performs analysis on multiple datasets but does not have the same focus on patterns of developer behaviour.

More recently, the design of flexible dashboards for monitoring collaborative software development has been advocated by Treude and Storey [5] who, however, focus primarily on supporting the awareness for an individual developer of

the rest of his/her team activities, whereas our objective to comparatively review the contributions of all developers in a team.

### V. IMPLEMENTATION STATUS AND TECHNOLOGIES

Our tool is built using JavaScript and execution is fully client side. As mentioned in the previous section we are pre-processing data into a JSON formatted database file but this could easily be extended to communicate with a web service for the data of which we have done similar work in the past with WikiDev2.0. The dashboard interface relies on the Dashboard plugin, and the code for parsing the data and the visualizations are using data-driven documents. The timeline visualization is built on top of the SIMILE widget and the popup windows are implemented with the clueTip library.

### VI. EXPERIENCE, CONCLUSIONS AND FUTURE WORK

The *Collaboratorium Dashboard* tool was developed in the context of the MSc thesis of one of the co-authors [6], in the context of which it was integrated with the IBM Jazz collaborative system. Through an empirical ethnographic study with a single software team, we observed that the intuitions communicated by the tool match well the team members own perceptions of their work and progress. This experiment validated the tool's usefulness. The usage scenarios reported in this paper are with a set of 15 WikiDev repositories, thus validating the tool's flexibility. We are now working on extracting GitHub data in the *Collaboratorium Dashboard* data store. Finally, we are working on providing views across multiple projects, similar in scope to what the single project/team exploration is providing.

### ACKNOWLEDGMENTS

This work was funded by NSERC, AITF (Alberta Innovates Technology Futures) and IBM.

### REFERENCES

- [1] E. Stroulia, F. Rocha, and N. Tsantalis, "A framework for collaborative software development analytics," in *Proceedings of the Workshop on "The Future of Collaborative Software Development"*, ser. CSCW '12. New York, NY, USA: ACM, 2012.
- [2] D. Krackhardt and K. M. Carley, "A pcans model of structure in organizations," *Proceedings of the 1998 International Symposium on Command and Control Research and Technology*, pp. 113–119, June 1998.
- [3] M. Fokaefs, D. Serrano, B. Tansey, and E. Stroulia, "2d and 3d visualizations in wikidev2.0," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–5. [Online]. Available: <http://dx.doi.org/10.1109/ICSM.2010.5609696>
- [4] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–33. [Online]. Available: <http://dx.doi.org/10.1109/ICSE.2009.5070505>
- [5] C. Treude and M.-A. Storey, "Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 365–374. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806854>
- [6] F. R. D. Pinho, "Analyzing individual contribution and collaboration in student software teams," Master's thesis, University of Alberta, Edmonton, AB, January 2013.

## SourceMiner Evolution: A Tool for Supporting Feature Evolution Comprehension

Renato L. Novais<sup>1</sup>, Camila Nunes<sup>3</sup>, Alessandro Garcia<sup>3</sup>, Manoel Mendonça<sup>2</sup><sup>1</sup>Information Technology Department, Federal Institute of Bahia, Campus Santo Amaro, Bahia, Brazil<sup>2</sup>Fraunhofer Project Center for Software and Systems Engineering @ UFBA, Brazil<sup>3</sup>Opus Research Group, Software Engineering Lab, Informatics Department - PUC-Rio, Rio de Janeiro, Brazil  
{renatoln, mgmendonca}@dcc.ufba.br, {cnunes, afgarcia}@inf.puc-rio.br

**Abstract**— Program comprehension is an essential activity to perform software maintenance and evolution. Comprehensibility often encompasses the analysis of individual logical units, called features, which are often scattered through many program modules. Understanding how the feature code is implemented along the software evolution history is essential, for instance, to perform refactoring activities. However, existing tools do not provide means to comprehend the feature code evolution. To overcome this shortcoming, this paper presents a tool called SourceMiner Evolution (SME) that provides multiple interactive and coordinated views to comprehend feature code evolution. SME implements a feature-sensitive comparison of multiple program versions. Our usability assessment with experienced developers indicated that SME allows them to efficiently perform recurring comprehension tasks on evolving feature code. The developers' performance was influenced by the combination of visual SME mechanisms, such as colors, tooltips and menu-popup interactions over the features' code elements.

**Keywords**— program comprehension; feature evolution, software visualization.

## I. INTRODUCTION

Program comprehension is an essential activity in software maintenance and evolution. The comprehensibility is even more important in large software systems. To improve comprehensibility, programs are often represented through logical units, so-called features. A feature can be defined as a prominent or distinctive user-visible aspect, quality or characteristic of a software system [1]. It is particularly challenging in evolving software systems to comprehend features as well as perform refactoring activities when they are scattered and tangled across many program modules [2]. For this reason, the feature evolution comprehension often requires: (i) to understand how the feature code (e.g., methods) is implemented across multiple software versions [3]; (ii) refactoring or modify the code realizing specific features [4]; and (iii) the completely reengineering of software features [3].

Several tools have been proposed to support feature comprehension, such as ConcernMapper [4] and CIDE [5]. Most of these tools only analyze the feature realization in a single program version. Although there are also visualization tools that provide developers with the feature history analysis, such as Feature Survival Charts [6], ConcernLines [7], and Feature Views [3], they still have several limitations. These tools do not provide developers with comprehensive information about the feature evolution realization. Additionally, they do not provide means to visualize

differential comparisons between software versions that implement certain features.

This paper presents a software visualization tool called SourceMiner Evolution (SME), which supports feature evolution comprehension in evolving software systems through graphical views. It is implemented as an Eclipse plug-in. This tool helps developers: (i) to comprehend features' code across the software evolution history when using multiple interactive and coordinated views; (ii) to do a feature-sensitive comparison of multiple program versions; and (iii) to perform maintenance tasks, such as refactoring or modification of existing features' code. Our assessment revealed that SME allows developers to effectively perform comprehension tasks on evolving feature code.

## II. SOURCEMINER EVOLUTION

The SME tool supports feature evolution comprehension through interactive visualizations. It allows users to select multiple program versions in order to analyze properties of the feature code evolution. SME uses feature mappings to portray the code elements that realize a feature. The feature mappings contain the explicit identification of code elements realizing each feature [4]. They are generated through a set of mapping expansion heuristics for each program version. SME processes the source code of the software versions, and uses colors to highlight both code elements that realize features and the feature code changes when comparing two software versions.

## A. SME Design

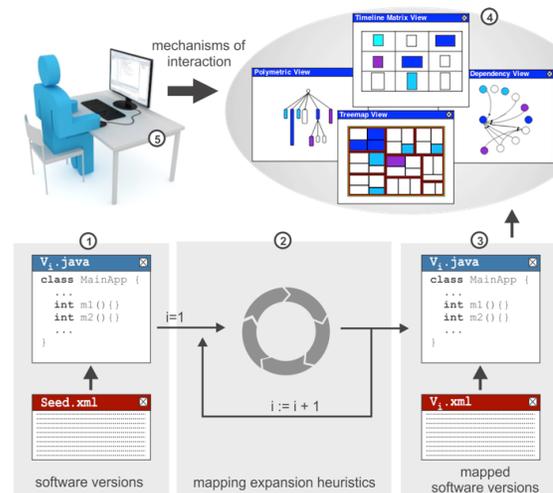


Figure 1. SME design.

Figure 1 depicts the SME design and how its main elements are related to each other. *SME inputs* consist of two different sources of information: the software versions to be analyzed and an XML file, so-called mapping *seed* (box 1 in Figure 1). The seed contains an initial set of code elements that are known to realize the selected features. More details are presented in Section II.E. The inputs are processed by a set of *mapping expansion heuristics* (Section II.E). These heuristics generate feature mappings for each software version (box 2 in Figure 1). By means of *multiple and coordinated views*, the mappings are displayed from four different perspectives. These views are used to represent the code elements realizing the features contained in mappings (box 4 in Figure 1). SME provides a set of interaction mechanisms that allows the users to interact with the tool, filtering, zooming, and getting on demand details about the analyzed features (see Section II.C).

### B. Views

SME provides an integrated, coordinated and interactive multiple view environment. It uses four views to address the feature evolution comprehension. Each view provides means for analysing the feature evolution under a different perspective: structure, inheritance and dependency. The first view addresses the structural perspective using a Treemap, which is a hierarchical 2D visualization that maps a tree-structure into a set of nested rectangles [8]. This view reveals how the feature code is organized into packages, classes and methods. The second view addresses the inheritance perspective of feature code evolution. It uses a Polymetric view [9] to show which classes extend others or implement certain interfaces. The third view addresses the dependency perspective. It uses interactive directed graphs (IDG) to describe coupling between software’s modules; i.e., features’ elements that depend on each other. The fourth view, called Timeline Matrix [15], is an on-demand view of the three previous ones. It shows the whole evolution of selected modules in detail. The visual elements of the views (rectangles) represent source code elements. SME decorates the visual elements realizing each feature or its evolution.

### C. Interaction Mechanisms

SME provides a set of mechanisms that enable the user to interact with the views while analyzing large software applications. SME provides filters, zooms, tooltips, and coupled navigation between views. The user can use filters to search for specific code elements realizing a feature or even select entities according to its properties. These filters are very useful in locating or displaying elements of one or more features that cannot be easily drawn together in the same screen shot. Graphical and conceptual zooms can be used to observe a feature element in detail. The views of SME are integrated so that the user can navigate among them. For instance, if the user identifies a class or interface of interest in Polymetric or IDG views, he can “*ctrl right click*” on the visual attribute that represents the feature element and ask to open it in the Treemap view or Timeline Matrix view. The views are also coordinated so that a change in some filter updates all the views at the same time. Some mechanisms of interactions work

through menu popups. The user can use “*ctrl right click*” over the selected visual element to trigger specific SME functionalities.

### D. Feature Evolution Visualization

SME allows users to select any color and associate it with a given feature. It then paints the code elements that realize a feature with the selected color. SME also uses colors to represent the changes of features’ code elements. The user can use a range bar slider to select any two versions of the software and observe their feature evolution. The views portray the elements of the most recent selected version and compare it against the other one. The code elements realizing the features are painted according to their evolution. As an example, consider two versions  $i$  and  $j$ . The elements of version  $j$  are drawn in the views, and three colors are used to portray their differences when comparing with the version  $i$ . It paints in light blue the code elements that realize features in version  $i$ , but not in  $j$ . It paints in dark blue the code elements that realize features in version  $j$ , but not in  $i$ . Finally, it paints in purple the code elements that have been removed from one feature in version  $i$  and added to another one in version  $j$ , or vice-versa. For this reason, we named these transferred elements. Figure 1(box 4) shows the views and colors supported by SME.

### E. Mapping Heuristics

SME relies on mappings, which contain the explicit identification of all code elements realizing each feature [4]. The mappings are XML files that contain the name of the features and their respective code elements. SME requires feature mappings to visualize the features’ code elements. It requires a feature mapping for each version of the evolving software system. However, manual feature mapping is time-consuming and costly to produce with high accuracy. For this reason, SME uses an external suite of heuristics to automatically generate the mappings for all the software system versions from a seed (Figure 1) [10]. This process, named feature mapping expansion is detailed in [10][11].

TABLE I. DESCRIPTION OF THE MAPPING EXPANSION HEURISTICS

Mapping Heuristic	Description
DFF - Detecting Omitted Feature Partitions	It detects methods comprising multi-partition features that were not mapped.
DCC - Detecting Code Clone Mismatches	It detects occurrences of code clones have not been mapped to a particular feature.
DIS - Detecting Interfaces and Super-Classes	It detects cases of super-classes and interfaces, which were either not mapped or those incorrectly mapped.
DCF - Detecting Communicative Features	It detects code elements incorrectly mapped to a feature that have a large interaction with other features’ elements.
DOA - Detecting Omitted Attributes	It detects occurrences of omitted attributes that have not been mapped.

Feature mapping expansion involves the automatic identification of feature elements in the code departing from an initial (seed) mapping. Even with the high accuracy of the expansion heuristics (80% to 100% [10][11]), they may produce a few imperfections in mappings. However, we observed in our studies that these imperfections do not impact

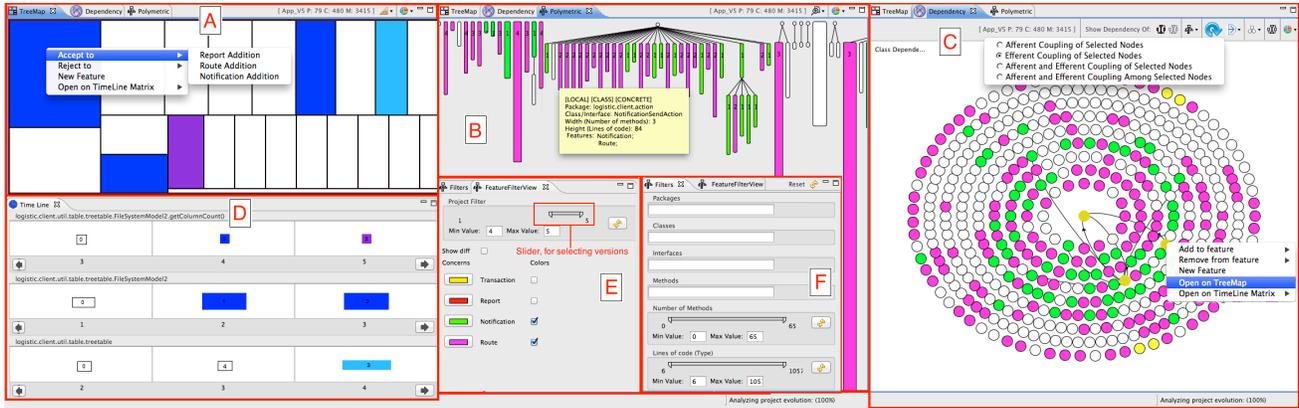


Figure 2. SME Views and Filters.

users’ performance when performing a wide range of feature evolution tasks [12]. However, SME allows developers to manually confirm or reject the code elements mapped to a given feature using interaction mechanisms over the views (Section II.C). Due to space limitations, we present the mapping expansion heuristics very briefly in Table I. For instance, given a feature, the heuristic DFP analyzes the previously mapped methods by comparing them with the non-mapped ones in terms of interaction similarity [13]. Interaction similarity is defined in terms of method interactions regarding its callees and callers in similar contexts [13]. Existing visualization tools do not address the idea proposed by DFP with respect to identify similar methods by considering the feature evolution and software history analysis.

#### F. Examples

This section describes some examples that highlight the main SME functionalities. Figure 2 shows the main SME views and filters. It illustrates the Treemap, Polymetric, Dependency and Timeline Matrix views (A, B, C, D in Figure 2), and filters (E and F in Figure 2), respectively. Views A, B and C portray the comparison between the versions 4 and 5 of the evolving logistic software for the oil industry. View D portrays the whole evolution of selected modules.

**Feature Evolution Comprehension.** The Treemap view (Figure 2.A) is illustrating through the colors dark blue, light blue and purple the feature code evolution when comparing the versions 4 and 5 of the analyzed system. The versions 4 and 5 are selected using the slider, which is highlighted in Figure 2.E. The dark blue color represents the code elements added to a feature; i.e., code elements are now realizing a feature. The light blue color represents the code elements are not realizing a feature anymore in version 5. The purple color represents the code elements removed from a feature in version 4 and added to another feature in version 5. The user is able to interact with the Treemap view and observe which code elements are realizing a given feature. According to the Figure 2, the menu popup indicates that the selected code element realizes the features *Report*, *Notification*, *Route*. (Section II.E).

**Interaction Mechanisms.** In the Polymetric view (Figure 2.B), we highlight the use of tooltip. When the user passes the mouse over a graphical element, SME shows its details. In this case, the tooltip illustrates that the selected code element is the Local Concrete Class named *NotificationSendAction*,

from *logistic.client.action* package, with three methods, 84 lines of code, and realizing the features *Notification* and *Route* (Section III). The dependency view (Figure 2.C) represents the coupling among the features’ code elements. The dependency represented in the view can be filtered, showing, in the example, only efferent (fan out) coupling [14] of selected nodes. In this view, we highlight the “*Open on Treemap*” option since the user identified an element of interest and requested to open this element in detail on Treemap view.

**SME Filters.** SME provides filters at the feature and system levels. Figure 2.E shows the FeatureFilterView filter. The user can select any two versions to be compared, request the diff between two versions, and assign colors to the features in this view. As it can be observed, the light green color represents the feature *Notification*, and the pink color represents the feature *Route*. Checkboxes are used to activate the visualization of the features’ code elements. When a code element realizes more than one feature, the displayed color will be the last color selected. Finally, Figure 2.F shows structural and lexical filters. The user can, for instance, search for specific classes or methods realizing features based on their names or size. These filters are useful in software visualization as they interactively reduce the amount of information being analysed.

### III. USER EXPERIENCE

We evaluated the tool by comparing it against the ConcernMapper tool (CM) with respect to feature evolution comprehension tasks [12]. The tools were empirically compared with regard to task execution time and correctness. Information about the participants involved in our evaluation, tasks, procedures and results are presented throughout this section. The heuristics’ inaccuracies did not impact on this evaluation since both tools used the same mapping files generated by the expansion heuristics.

**Participants.** The study involved 20 participants with industrial experience. These participants were professionals with similar level of experience in software evolution and maintenance.

**Tasks and Procedures.** The participants had to perform five comprehension tasks over five versions of a software system for the logistics of the oil industry [12]. None of the participants were familiar with this software system. This logistic software has been evolved since 2006, has complex

modules and an average size of 120 KLOC. Here we exemplify two of those tasks. In the first task, the developers should identify, when comparing the versions 1 and 2, which code elements in the `ValidationFrame` class are realizing the feature *Notification*. The feature *Notification* defines a system notification to users (e.g., email). In the second task, developers should locate, when comparing the versions 2 and 3, which code elements in the `Route` class are not realizing anymore the feature *Route*. The feature *Route* represents a route of products between two points in the logistics context.

**Results.** The SME outperformed CM with respect to task correctness and average task time [12]. Considering the time to execution, SME had a lower mean considering all the tasks (8.95%). Considering the correctness, the results showed that SME entailed a significantly higher correctness when comparing it to CM. The SME mean was 26.94% higher than CM. The quantitative and qualitative analysis enabled us to list SME strengths and weaknesses as well as new directions for our feature evolution visualization research. SME's integration and coordination allow users to navigate easily among views. The participants claimed the importance of different mechanisms for double check their answers when executing a feature comprehension tasks. These mechanisms are mainly useful when developers are not familiar with analyzed software system. Researchers have observed that visualization tools are useful to show all modules of each software version [6][16]. However, these tools might be empowered by other views that track the evolution of features' code elements [15]. Based on our initial analysis, we intend to observe how challenging is to comprehend the evolution of feature code smells using SME.

#### IV. RELATED WORK

Many researchers have proposed visualization strategies to deal with feature comprehension tasks. For instance, Wnuk et al. [6] developed a technique called Feature Survival Charts for visualizing the scoping changes. This visualization helps the user on the decision process of including or excluding features on the next release. Fischer and Gall [16] presented a feature evolution visualization that relies on logical coupling information introduced by changes required to fix a reported problem. This approach uncovers hidden dependencies between features using a graph-based visual form. Treude and Storey [7] use a timeline-oriented view of features over time. This tool uses a regular display to explore the characteristics of software components based on features. Greevy et al. [3] propose execution traces to map feature to source code. They plot the software evolution to reveal how and which features were evolved. Unfortunately, these tools do not represent which elements have been added or changed to realize the feature implementation throughout the software system evolution. On top of that, our work differs from others because we use four interactive, coordinated and integrated views that present the feature code evolution from different perspectives.

#### V. CONCLUSIONS AND FUTURE WORK

This paper presented SME, which is a tool that provides an integrated, coordinated and interactive multiple views for

feature code comprehension in evolving software systems. SME provides a set of mechanisms to allow developers to interact with the views, such as filters, zooms, and coupled navigation between views. Additionally, SME allows developers to assign colors to features of interest and analyze the evolution of the features' code. Our preliminary usability assessment of the SME tool indicated that developers could easily comprehend the feature evolution through the mechanisms provided. For instance, the use of colors, filters and interactive mechanisms enable developers to understand the code elements that realize a given feature across multiple versions. As future work, we intend to run controlled experiments to evaluate SME in other contexts, such as refactoring of feature envies. The SME tool and its demonstration video are available in <http://wiki.dcc.ufba.br/SoftVis/SMEFeature>.

#### REFERENCES

- [1] K. Kang, et al., "Feature-oriented domain analysis (foda) feasibility study," Tech. Report, Carnegie-M. U., SE Inst, 1990.
- [2] E. Figueiredo et al. "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability," Proc. of ICSE, pp. 261-270, 2008.
- [3] O. Greevy, et al., "Analyzing software evolution through feature views," JSME vol. 18, pp. 425-456, 2006.
- [4] M. Robillard and F. Weigand-Warr, "Concernmapper: simple view-based separation of scattered concerns," in Proc. of the OOPSLA workshop on Eclipse Tech., ACM, pp. 65-69, 2005.
- [5] J. Feigenspan, et al., "Using background colors to support program comprehension in software product lines," in Proc. of the EASE, IEEE, pp. 66-75, 2011.
- [6] K. Wnuk, et al., "What happened to our features? visualization and understanding of scope change dynamics in a large-scale industrial setting," in Proc. of the RE, IEEE, pp.89-98, 2009.
- [7] C. Treude and M-A. Storey, "ConcernLines: A timeline view of co-occurring concerns," in Proc. of the ICSE, pp. 575-578, 2009.
- [8] B. Johnson and B. Shneiderman, "Tree-Maps: a space-filling approach to the visualization of hierarchical information structures," in Proc. of the VIS, IEEE, pp. 284-291, 1991.
- [9] M. Lanza and S. Ducasse. "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Trans. Softw. Eng.* 29, 9, pp. 782-795, September 2003.
- [10] C. Nunes, "On the proactive identification of mistakes on concern mapping tasks," in Proc. of the AOSD, First Place at ACM Competition, NY, USA 85-86, 2011.
- [11] C. Nunes, et al. "Heuristic expansion of feature mappings in evolving program families," SP&E, 2013 (to appear).
- [12] R. Novais, et al., "On the proactive and interactive visualization for feature evolution comprehension: an industrial investigation," Proc. of the ICSE - SEIP. pp. 1044-1053. 2012.
- [13] T. Nguyen, et al., "Aspect recommendation for evolving software," in Proc. of the ICSE, NY, USA, 361-370, 2011.
- [14] L. Briand, et al., "Using coupling measurement for impact analysis in object-oriented systems," in Proc. of the ICSM, pp. 475-482, 1999.
- [15] R. Novais et al., "TimeLine Matrix: an on demand view for software evolution analysis," in Proc. SV Workshop, 2012.
- [16] M. Fischer and H. Gall, "Visualizing feature evolution of large-scale software based on problem and modification report data," in JSME. pp. 385-403, 2004.

# CONQUER: A Tool for NL-based Query Refinement & Contextualizing Code Search Results

Manuel Roldan-Vega, Greg Mallet, Emily Hill, Jerry Alan Fails  
 Department of Computer Science  
 Montclair State University  
 Montclair, NJ, USA  
 {roldanvegam1, malletg1, hillem, failsj}@mail.montclair.edu

**Abstract**—Identifying relevant code to perform maintenance or reuse tasks is becoming increasingly difficult. Software systems continue to grow and evolve, and developers often find themselves searching within thousands to even millions of lines of code to identify code relevant to a particular maintenance task. Automated solutions are vital to help developers become more efficient at locating code to be modified when performing maintenance tasks. In order to address this need and help developers reduce the time spent finding and searching for relevant code, we have built an Eclipse-plugin, CONQUER, that helps developers identify relevant results by providing critical insight and context of how query words are used in the code. CONQUER leverages advanced natural language (NL) information in the source code to group, sort and display the results in a meaningful way. In addition, CONQUER analyzes the frequency and co-occurrence of words in the method result set to provide alternative phrases that can help further refine the query. This rich contextual hierarchy helps the developer quickly determine if the query is correct and hone in on relevant results. The NL-based organization of results reduces the number of relevance judgments the developers need to make, and thus can reduce the overall time for a maintenance task.

**Index Terms**—feature location, source code search, software maintenance

## I. INTRODUCTION

When performing software maintenance or reuse tasks, developers must first identify the relevant code fragments to be modified or reused. In fact, locating the code to be modified is often more time consuming than the maintenance task itself [1]. As software systems continually grow in size and complexity, it becomes more difficult for developers to identify code relevant to a particular task within millions of lines of code.

Traditional search tools that use natural language (NL) queries display search results as a simple list [2], [3]. These tools typically do not offer feedback, context, or insight for the developers to understand the results and determine if they are relevant to their query, nor whether the query was successful or not. The developer cannot easily determine if the query is accurate, overly general (and needs to be more specific), or if the query is completely inaccurate. In addition, few of these search tools provide alternative words or phrases that can help the developer refine or reformulate the query.

To satisfy this need, we developed CONQUER, an Eclipse-plugin that provides NL context to the developers so that

they can identify relevant code or reformulate queries more efficiently and effectively. CONQUER builds on previous research on natural language phrasal representations [4] and source code context [5] and enhances this prior work by analyzing additional contextual information, grouping the results by actions and themes, and suggesting alternative query words. These elements are all integrated into the plugin and made available to the developer for query reformulation and refinement.

## II. BACKGROUND AND RELATED WORK

In order to provide developers with useful context on how the query words are used in the source code, it is necessary to identify linguistic factors such as the action and theme of a method signature. The action represents the verb or main activity of the method, whereas the theme is the direct object of that action. The SWUM (Software Word Usage Mode) [4] for Java allows us to capture these linguistic relationships in code. For example, given the signature `addToList(Item)`, SWUM would identify the action as “add” and the theme as “item”. In the cases where there are multiple actions or themes, SWUM also analyzes the head distance to determine the most relevant action or theme. This linguistic information has been used in prior work to improve source code search [4], but has never before been used to group and arrange the query results.

Although the search mechanism is the same as proposed in prior work [4]—which integrates location, semantic role, head distance, and usage information to calculate the relevance score—this query mechanism and results view were not integrated into the Eclipse interface and thus were not usable by the average developer. The current approach not only integrates the query mechanism into the Eclipse search dialogue box and displays the results in the view pane; it also provides additional contextual feedback for determining whether the results meet the search needs of the developers. CONQUER goes beyond the simple hierarchy of phrases used in prior work [5] by organizing the search results by actions and themes. In addition, CONQUER automatically recommends alternative query words to help refine overly general or inaccurate queries.

CONQUER differs from previous search tools [2], [3], [6] in that it not only displays the relevant results in a list, but allows the developer to quickly skim a list of actions and

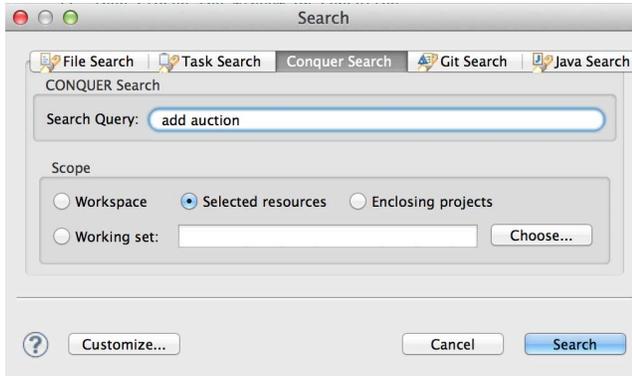


Figure 1. CONQUER is an Eclipse plug-in that adds a Conquer Search tab to the general Eclipse Search dialog box.

themes in the query. In the future, the results could also be presented in the context of a call graph [7], [8], [9]. This work could be enhanced in future by including semantically related words [10], [11].

### III. TOOL OVERVIEW

The driving insight for our tool is that the same query might be used to search for multiple information needs. Our challenge is in designing a results view that meets the needs of developers searching with queries that are ideal, overly general, partially correct, or completely incorrect. Thus, we designed the CONQUER results view to address 3 key challenges:

- 1) Quickly determine if results are relevant (i.e., did the query work?).
- 2) Find alternative query words to refine overly general queries or substitute words in a partially correct query.
- 3) Find relevant results quickly.

To meet these challenges, we implemented CONQUER as an Eclipse-plugin that allows developers to search any Java software project. The search is integrated into Eclipse’s general search dialog box with a CONQUER tab (see Figure 1). After the developer enters a natural language query (i.e., a series of keywords), the plug-in initiates the search mechanism [4].

The CONQUER tool gathers the results obtained from the search and groups them by three categories: action, theme or relevance score. These results are displayed in the view pane in three individual sections in a tree structure based on the action, theme or relevance score. The method signature of each result is displayed as an Eclipse Java element, allowing the developer to click on it to directly navigate to the source code for that method (see Figure 4). While CONQUER can be used to search for methods or fields, for this first phase we have limited the scope to present only methods, leaving fields to future work.

#### A. Prevalence of Query Words

The organization of the CONQUER results view provides the developer with quick feedback on the relative frequency of

query words in the result set (see Figure 2). This information allows the developer to quickly determine how well the results match their information need (as expressed by the query), and judge which words should be removed from inaccurate queries.

For example, consider the query “mute music” searching for the “mute” feature in music management software, where the concept of ‘mute’ is more important to the feature than ‘music’ (show in Figure 2). If ‘music’ is much more prevalent in the source code than ‘mute’, it will fill the search results with irrelevant entries, causing more work for the developer. The frequencies next to each query word approximate each word’s power to discriminate between relevant and irrelevant results. For instance, if ‘music’ occurs 100 times in the result set and ‘mute’ just 4 times, the developer will know to revise the query to focus on ‘mute’, since ‘music’ is likely to occur with irrelevant results. This will also help the developer focus on the concept of ‘mute’ in the other parts of the results view.

#### B. Suggested Alternative Query Words

Between the action and theme labels and the tree of action/theme results, we display alternative query words to help reformulate poor queries. As shown in Figure 3, these words can help the developer refine or reformulate the query in cases when the original query is overly general or inaccurate.

For example, consider searching for the shuffling global playlist feature in a music player. A developer might first enter the query “shuffle global playlist”, where the ideal query for this particular search task is “shuffle queue”. CONQUER will calculate the frequencies of the other action and theme words that co-occur with shuffle and global, and suggest the most highly co-occurring words as alternative query words. As shown in Figure 3, ‘queue’ would be suggested as an alternative word, helping the developer refine the query to “shuffle global queue”, which is much closer to the ideal query.

#### C. Results Grouped by Action or Theme

When a query is not ideal, there still may be relevant results buried in the bottom of the result list with a lower score. We introduce two hierarchies of results, organized by action and theme, to allow more opportunities for a variety of search results to be displayed in the initial result view and facilitate hierarchical navigation of the search results. These two hierarchies are designed to allow the developer to locate relevant results quickly, even for less than ideal queries.

The action section, depicted in Figure 2, displays the search results in a tree structure grouped by the action. All the results that have the same action are grouped into one category. For example, methods `saveFile(File)` and `saveText()` would belong to the same action category, “save”. Each category has subcategories grouped by theme. In this example, “file” and “text” would both be subcategories. CONQUER calculates the maximum theme score for each subcategory and sorts the categories based on it. CONQUER initially displays a list of all the actions (categories), which can be expanded to see the different themes (subcategories) that appear with

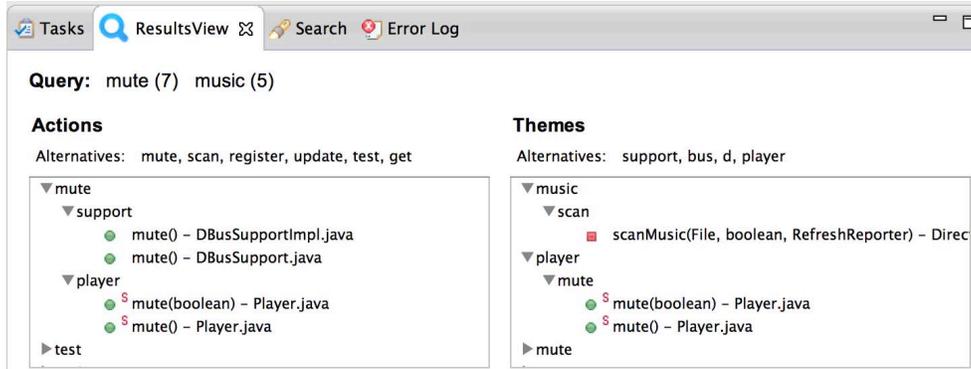


Figure 2. Results grouped by action (left) and grouped by theme (right), displayed in a tree structure.

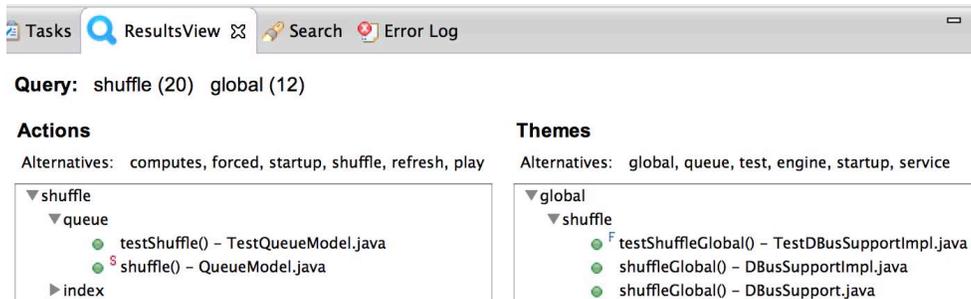


Figure 3. Alternative query words are shown above the action and theme sections.

that specific action. The developer has the ability to expand each subcategory further and see the method signature of each result, as displayed in Figure 2.

Similar to the action groups, the theme view displays the search results, but the categories are formed by the themes and the subcategories by the actions (see Figure 2). In this view, `saveFile(File)` and `openFile(File)` would belong to the same category, “file”. Under the “file” category CONQUER displays subcategories “save” and “open”. CONQUER calculates the maximum action score for each of the themes and sorts the theme categories based on this score.

#### D. Results Grouped by Relevance Score

Unlike the tree views of the action and theme section, the bottom of the result view simply displays the results in a list sorted by overall score [4], shown in Figure 4. This section is useful when the query is on target and the relevant results are highly ranked.

The result phrase list view is closer to a more traditional search results view, simply listing the methods and files where matches occur. However, we go beyond a simple list view by providing a short natural language phrase that describes each method, to make the results easier to be quickly skimmed without requiring the cognitive overhead of mentally parsing the syntax of method signatures and file names. This facilitates rapidly scrolling through the results, to see what types of words and phrases describe the methods appearing in the result

set. In the event of an ideal query, this section may contain the relevant results at the top of the list.

## IV. INITIAL USER FEEDBACK

We asked 13 Java developers with industry experience to compare CONQUER (CONQUER) with Eclipse’s built-in File Search (ECLIPSE). Many of the users appreciated the alternative suggested query words as well as the action and theme tree view by noting “the summary trees help faster navigation” and “Query recommendations are *very* helpful and important when searching unfamiliar code.” However, not every user found all aspects of the interface to be intuitive, and some had trouble understanding the action vs. theme view. Because ECLIPSE is familiar, many participants found it easy to use and uncomplicated. Some especially liked the straightforward results view: “I liked seeing the list of class names and the line numbers and the context of where my keywords were found.” However, other participants found the ECLIPSE queries difficult to use: “Touchy results. Make one wrong move and your search results get it.”

In analyzing the results, we observed that there are certain situations that lend themselves to each search mechanism. When developers have an idea of the appropriate identifier names to search for, they want to perform strict matching of identifiers or keywords with an ECLIPSE-like search. In contrast, when developers are not familiar with a codebase or its naming conventions, and have no insight into what

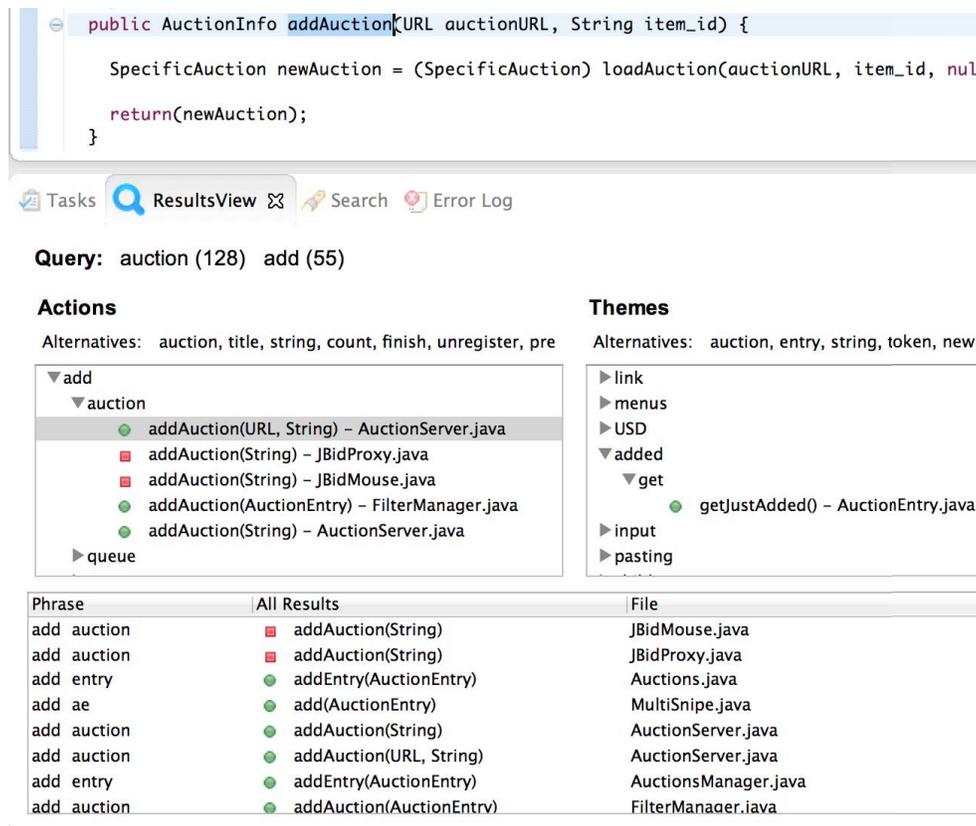


Figure 4. The CONQUER results view displays 3 sections: grouped by action, grouped by theme, and all results sorted by relevance score.

identifiers would be relevant, they need the support of a natural language search provided by CONQUER. Providing a flexible interface for either scenario will further enable the developer to use a single search interface for all their search needs. Future work will investigate how to integrate that customizability in an intuitive way, without using too much screen real estate.

In general, participants appreciated the alternative query words suggested. In fact, some participants requested suggesting synonyms as well co-occurring words. In future, we would like to explore using synonyms [10], [11] as well as co-occurring words to enhance the suggested alternative query words.

## V. CONCLUSION

In this paper, we present CONQUER, a tool for NL-based query refinement and contextualizing source code search results. It allows developers to quickly understand and determine if the query they use to search the source code returns relevant results, and if not, help identify related words to reformulate the query. These insights into the search results can reduce the overall time developers spend in maintenance tasks by helping them locate relevant code more quickly. In future work, we anticipate integrating these techniques into a Visual Studio search framework such as Sando [6].

CONQUER can be downloaded from <http://www.cs.montclair.edu/~hillem/CONQUER/>.

## REFERENCES

- [1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE TSE*, vol. 32, no. 12, pp. 971–987, 2006.
- [2] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, "Source code exploration with Google," in *ICSM*, 2006, pp. 334–338.
- [3] T. Savage, M. Reville, and D. Poshyvanyk, "Flat3: feature location and textual tracing tool," in *ICSE*, 2010, pp. 255–258.
- [4] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *ASE*, 2011, pp. 524–527.
- [5] —, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *ICSE*, 2009, pp. 232–242.
- [6] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *FSE*, 2012, pp. 15:1–15:2.
- [7] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: a search engine for finding functions and their usages," in *ICSE*, 2011, pp. 1043–1045.
- [8] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *ICPC*, 2011, pp. 1–10.
- [9] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD*, 2007, pp. 212–224.
- [10] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *MSR*, 2013, pp. 377–386.
- [11] J. Yang and L. Tan, "Inferring semantically related words from software context," in *MSR*, 2012, pp. 161–170.

# srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code

## A Tool Demonstration

Michael L. Collard<sup>1</sup>, Michael John Decker<sup>2</sup>, Jonathan I. Maletic<sup>2</sup>

<sup>1</sup>Department of Computer Science  
The University of Akron  
Akron, Ohio  
collard@uakron.edu

<sup>2</sup>Department of Computer Science  
Kent State University  
Kent, Ohio 44242  
mdecker6@kent.edu, jmaletic@kent.edu

**Abstract**—srcML is an XML representation for C/C++/Java source code that forms a platform for the efficient exploration, analysis, and manipulation of large software projects. The lightweight format allows for round-trip transformation from source to srcML and back to source with no loss of information or formatting. The srcML toolkit consists of the src2srcml tool for robust translation to the srcML format and the srcml2src tool for querying via XPath, and transformation via XSLT. In this demonstration a guide of these features is provided along with the use of XPath for constructing source-code queries and XSLT for conducting simple transformations.

**Keywords**—srcML; static code analysis; source transformation

### I. INTRODUCTION

The research and practice of software maintenance and evolution almost always, in some manner, requires the exploration, analysis, or manipulation of source code. Here we demonstrate the features of a powerful infrastructure, namely srcML, which supports these tasks via an underlying format, parser, and tool set.

While srcML [3, 4] has been publicly available to the research community since the mid 2000's it has recently gained more traction in both industry and the research community. Much of this is due to a number of new usability features recently added, expanded language support, scalability, and robustness of the platform.

In short, srcML is an XML format for source code. Specifically, the parsing technology supports C/C++ and Java. However, we will soon be releasing support for C#. The XML markup identifies elements of the abstract syntax for the language. This allows us to leverage XML tools to support various tasks of exploration, analysis, and manipulation.

A number of underlying features make srcML particularly useful for evolution and maintenance. The main philosophy is to take a programmer-centric view of the code rather than a compiler-centric one. First, the conversion from source code to srcML is lossless. That is, no formatting, comments, or actual code is lost. There is a round-trip equivalency from source code to srcML and back to the original source code. Additionally, macros, templates, and preprocessor statements are marked up. That is, the preprocessor is not run (or need not be run) prior to conversion to srcML. This also implies that code with missing includes, libraries, or code fragments can be converted to well-formed srcML. Lastly, the conversion to

srcML is extremely efficient, running faster than a compiler (over 25KLOC/sec).

srcML has been used for a variety of maintenance problems. This includes, but not limited to, such things as the analysis of large systems to automatically reverse engineer class and method stereotypes [9], supporting syntactic differencing [12], and applying transformations to support API and compiler migration [5].

In the demonstration we will provide an introduction of how to use srcML. It is a command-line tool with a number of options for converting one file, multiple files, or complete source archives to srcML. Then, a number of basic features to explore and analyze source code using the infrastructure and XPath are presented. Means to develop specialized programs using such things as XML utility libraries and Python programs are also demonstrated. Lastly, simple ways to manipulate the source code to produce complex transformations are given.

The srcML infrastructure, including the format, parsing technology, and a select set of tools is currently open source and licensed under GPL. It is available for download at:

<http://srcml.org>

### II. THE SRCML PLATFORM

The srcML platform is based on the srcML format, which is an XML format where the syntactic aspects of the source code are marked with srcML elements. An example of srcML is given in Figure 1. The main element is *unit* which contains some optional attributes about the source code, including filename, directory, and version. The required attribute *language* includes the programming language, which must be specified when the srcML is created. The *unit* element contains the srcML form of the source code. This includes all of the original source code for that translation unit, and the srcML markup elements. srcML includes elements for all syntactic statements, e.g., *if*, *while*, *for*, *expr\_stmt*, *decl\_stmt*, and program structural elements, e.g., *function*, *class*, *namespace*, all in the namespace <http://www.sdml.info/srcML/src>. C-preprocessor statements are also marked, e.g., *cpp:include*, in their own namespace <http://www.sdml.info/srcML/cpp>. Special elements include escape for invalid XML characters, e.g., formfeeds. Optional markup is provided for literals, operators, and type modifiers. The complete list of srcML elements are on the website.

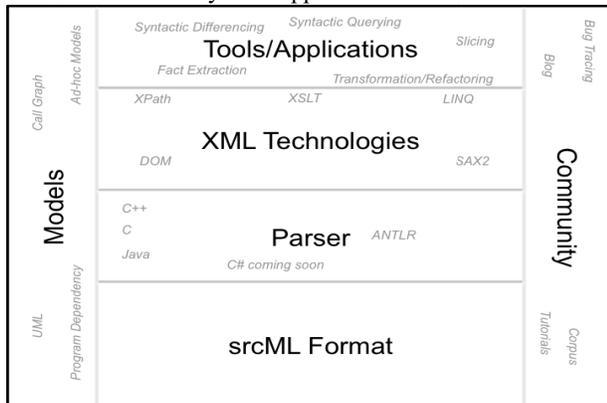
```

<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++"
filename="ex.cpp">
<comment type="line">// copy the input to the output</comment>
<while>while <condition>(<expr><name><name>std</name>::<name>cin</name></name> &gt;&gt;
<name>n</name></expr></condition>
  <expr_stmt><expr><name><name>std</name>::<name>cout</name></name> &lt;&lt; <name>n</name> &lt;&lt;
  '\n'</expr></expr_stmt></while>
</unit>

```

**Figure 1. A code fragment in the srcML format. Note that all original text is preserved, including white space and comments. The XML markup is placed to indicate syntactic context. The srcML format can also represent complete source code files and even complete projects in a srcML archive.**

The main architectural elements of the srcML infrastructure are presented in Figure 2. Underlying everything is the srcML format. Directly on top of that is the parsing technology. Then XML technologies can be used to build various tools and applications. On the left are the abstract models we use as software engineers to conduct maintenance and evolution. These models cross the various layers. On the right we have the srcML community and support.



**Figure 2. Architecture of the srcML infrastructure.**

### A. srcML Archive

For simple code fragments and individual files, the basic srcML format works well. But, as is done for source-code archives such as the tar format for systems with large numbers of files, it is much more convenient to combine these separate files into one large srcML file, which is a srcML archive. It basically consists of separate *unit* elements for each source code file, wrapped in an outer *unit* element. For multiple file input, the srcML archive is the default. The srcML toolkit automatically adjusts to basic and archive srcML files.

### B. Implementation

The srcML parser takes full advantage of the srcML format. The parser is custom built using a modified version of the LL(k) ANTLR 2.7.7 parsing toolkit. A separate parsing stream is used for white space and comments, allowing them to be completely preserved. Another separate parsing stream is used for preprocessor statements. Both of these separate streams allow for the primary parser to work with the syntax of the language. Before output, the tokens from the separate parsing streams are merged back into the primary parser stream. Preprocessor statements can affect the primary parsing, so information from the preprocessor stream is used with a set of heuristics to deal with these cases.

In addition to ANTLR, many of the features are provided by the libraries *libxml2* and *libarchive*. The primary use of *libxml2* is for generating XML when creating srcML, and for conversion of srcML back to source code. *Libxml2* is also used for accepting http URLs, executing XPath queries, and in encoding issues (using the *libconv* library). The *libxml2* associated libraries *libxslt* and *libexslt* are used for performing XSLT transformations.

The other library used extensively is *libarchive*, which supports encoded and archive file types, e.g., gz, tar, cpio, etc. *libarchive* is combined with *libxml2* to provide for remote access of these file types, e.g., an http tar.gz file.

## III. EXPLORATION & ANALYSIS

The first step in using the srcML toolkit is to translate the original source code into the srcML format. This can be the entire system, a few select files, or a code fragment. The system may be a single language or multi-language. The following command can be used to convert all the source code in *KOffice* into srcML directly from the url of the *KOffice* archive:

```

src2srcml --register-ext h=C++
http://download.kde.org/stable/koffice-2.3.3/koffice-
2.3.3.tar.bz2 -o koffice.xml

```

In this example, files with the extension '.h' are treated as C++. At this point a srcML version of the entire *KOffice* source code project is generated and put into the file *koffice.xml*. The remote access and conversion of the source code archive (tar.bz2) is handled automatically. The resulting srcML file is about 164 MB, while the original source code (as text) takes about 48 MB, producing a reasonable 3.42 times increase in size.

In addition to an input source from a URL, *src2srcml* can be given an individual file, a list of files, or a directory. In either case, the language of an input source is determined automatically from the filename's extension. *src2srcml* also has multiple options for specifying input source-code encoding (e.g., Latin1) and optional markup for literals, type modifiers, and operators.

Once in the srcML format, the code can be explored using XPath. To find the number of a particular syntactic item, we can directly count them. For example, to find the total number of *for-statements* in *KOffice*:

```

srcml2src --xpath "count(//src:for)" koffice.xml

```

In this case, the result is 6,813. Elements in the XPath are by default prefixed with *src* for the syntactic elements and *cpp* for c-preprocessor elements. To find a list of particular items, we can also use XPath. The result is a srcML archive with

each query result in a separate unit element. The following will find these names, and then count the number of them using the `--longinfo` option:

```
srcml2src --xpath "//src:class/src:name" koffice.xml |
srcml2src --longinfo
```

This query took about 6 seconds on a MacBook Pro 2.66 GHz

```

1 reader =
libxml2.newTextReaderFilename(sys.argv[1])
2 while reader.Read():
3     if reader.NodeType() == 1 and
        reader.Name() == 'function':
4         # expand the subtree
5         node = reader.Expand()
6         # setup for XPath evaluation
7         ctxt = node.doc.xpathNewContext()
8         ctxt.setContextNode(node)
9         ctxt.xpathRegisterNs("src",

"http://www.sdml.info/srcML/src")
10        ctxt.xpathRegisterNs("cpp",

"http://www.sdml.info/srcML/cpp")
11        # output the function name
12        print
13        ctxt.xpathEval("src:name")[0].getContent() + ', ',
14        # extract the call names
15        calls =
16        ctxt.xpathEval("src:block//src:call/src:name")
17        calllist = [call.getContent() for call in
18        calls]
19        # output the list of calls
20        print ', '.join(set(calllist))
21        # finish up
22        ctxt.xpathFreeContext()
23        # delete this subtree
24        reader.Next()

```

Intel Core i7 with 8GB RAM.

**Figure 3. Main loop from a Python program using the `libxml2 xmlTextReader` API to generate data for a call graph. The program reads nodes (Line 2) until a function element is found (Line 3). The tree for that function is expanded (Line 5), and XPath extracts the name of the function (Line 12) and the list of calls (Line 14). This function subtree is then deleted (Line 21).**

When it is necessary to further analyze the system, XPath querying can still be used, either entirely alone, or as a first step of further analysis. This further analysis can be performed by any XML tool or API. As an example, the Python program in Figure 3 takes each function in a system and generates call information. The output is each function name followed by a list of calls made by that function.

The Python program (Figure 3) uses the `libxml2 TextReader` interface. The main loop (Line 1) reads each node in the srcML file. When a function element is detected (Line 2), the entire XML subtree for that function is created by the call to the method `Expand()` (Line 5). Since we now have the complete tree for this function, we use XPath to extract the necessary information. First, the XPath evaluation is setup (Lines 7 – 10). Then the function name is extracted using the XPath `src:name` (Line 12). Note that the XPath expressions are written from the context of this `src:function` element. Lastly for this function, all of the calls are extracted using the XPath `src:block//src:call/src:name` (Line 14). It is relatively straightforward to extend this example to the extraction of

other parts of methods/functions. Additional cases for other elements are easy to add.

This approach is highly scalable, as it takes about a minute to run this on a srcML archive of the entire linux kernel. General evaluation of XPath requires a complete XML tree in memory (similar to DOM). However, in this case, we only create a tree for the current function, and then remove it when finished. This general approach can be used with other programming languages and other XML APIs, especially with pull parsers. For instance, a similar approach is built into the srcML toolkit for XPath and XSLT evaluation.

#### IV. MANIPULATION

We now show a simple XSLT transformation that uses srcML to instrument source code to gather and report basic profiling information. Figure 4 shows parts of the XSLT program. The full program is available at the demo website. The source code is transformed in three ways: 1) access to a global profiling object is added to each source file, 2) functions are instrumented to update the global profile object, and 3) the main function is modified to report the profiling information. A sample program is provided on the accompanying website to illustrate the program. The instrumentation of the sample program can be performed using the following command:

```
src2srcml sort.cpp sort_lib.cpp | srcml2src --xslt profile.xsl
| srcml2src --to-dir profiled
```

First, the source files are converted to the srcML format. Then the XSLT program `profile.xsl` is applied to all of the files in the srcML archive. After the profile code is inserted, the transformed source code files are extracted.

The `profile.xsl` transformation program consists of separate templates to insert the required code. Note that any inserted source can be put as plain text, i.e., it is not necessary to insert srcML markup. It also contains an identity copy XSLT template that assures that any unmodified code is copied over.

For efficiency, the XSLT program is applied to the srcML of each individual source code file one at a time. The result of applying the XSLT to each individual unit is then merged into the output srcML archive. This provides scalability of the transformation to very large systems.

Although our example was an XSLT program (using the builtin `srcml2src` feature), transformations can be written using any XML tool or API, e.g., LINQ, SAX2, DOM, etc. For example, the approach used in the Python program with `TextReader` (as shown previously) can be extended into a transformation. A potential complexity is an identity copy of all of the unchanged parts of the source code being transformed. In XSLT, the identity template handles this.

#### V. RELATED WORK

It has been observed that automated source code transformations intended to be handed back to a developer must preserve the programmer's view of the document, i.e., preserve white space, comments, and the expressions of literals, and failure to do so may mean the rejection of the result [6, 14] and tool. It has been observed that these compiler-centric approaches are often not a good match to the problems that they are trying to solve [10, 14]. There are exceptions to this problem with compiler-centric approaches, with one example being the DMS system by Baxter [2].

```

<!-- Insert call to global profile object to record call of this function -->
<xsl:template match="src:function/src:block">
<xsl:copy-of select="node()[1]"/>
<xsl:text>functions.count(_LINE_, "<xsl:text><xsl:value-of select="."/src:name"/><xsl:text>");
</xsl:text><xsl:apply-templates select="node()[position()>1]"/></xsl:template>

<!-- identity copy -->
<xsl:template match="@*|node()"><xsl:copy><xsl:apply-templates
select="@*|node()"/></xsl:copy></xsl:template>

```

**Figure 4. Templates from an XSLT program to insert profiling information into source code. Other templates insert the proper includes, and output at the return of the main() function. The identity copy template makes sure that all other source code is preserved.**

Baxter has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph.

One approach is to move down to the level of lexical analysis and provide for the transformation at that level, as in [8]. This allows for the preservation of all of the text, but at a cost of complex regular expressions. Another approach that preserves the programmer's view is to move the transformation to the level of the grammar as in TXL [7]. Using this approach, the transformations are written as part of the grammar for parsing the language. The approach shares many of the advantages of our approach: preservation of programmer's view, scalability, robustness, etc. The difference is in the format of the transformation. Instead of grammar rules, our approach treats the text of the source code as data in XML, and the transformations are XML transformations. ASF+SDF and Rascal [11] use a similar approach. Stratego [15] also support various means to apply transformation rules.

The Proteus system [16] addresses similar problems of performing transformations on large C++ systems while preserving the layout and handling code before preprocessing. Other approaches include JavaML [1] and others using an intermediate language to describe the source, as in the case of the C Intermediate Language (CIL) [13].

## VI. CONCLUSIONS AND FUTURE WORK

The srcML platform has shown itself to be highly useful for the exploration, analysis, and transformation of source code. The principle of the format and tools to preserve all elements of the original source code text allows for any information present in the code to be extracted, whether lexical, documentary, or syntactic. Although all of this information may not be needed for a particular task, using XPath, the srcML toolkit, or a particular XML tool, unneeded information is easy to avoid. The reasonable increase in file sizes of srcML over that of the original source code lessens the cost of this potentially unneeded information.

For the future, plans are to increase language support for Java and the new features of C++11. At the request, primarily from industry, the plan is to include language support for C#. As the number of supported languages increases, the use with mixed-language systems becomes more important, and the plan is to include full support for this. The current internal architecture of the system makes it difficult for anybody (except the coauthors) to fix bugs or support new languages and language features. The plan is to change to a plug-in

parser architecture. We envision that srcML parsers for new languages would be specified using a ANTLR-type declaration. Our goal is to first implement this for DSLs, and then eventually move currently supported programming languages (e.g., C++) to this format. This would

greatly increase the ability of other developers to support new language features and versions.

One of the main roadblocks to usage of any system is the support in the form of examples, tutorials, documentation, etc. The plan is to increase and organize these materials at our new website (srcml.org). We especially want to support more integration into other APIs and tools. Finally, for srcML to grow in both capability and usage, a broader community needs to be formed.

## REFERENCES

- [1] Badros, G. J., "JavaML: a markup language for Java source code", *Computer Networks*, vol. 33, no. 1-6, 2000, pp. 159-177.
- [2] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in *ICSE*, May 23-28 2004, pp. 625-634.
- [3] Collard, M. L., Decker, M., and Maletic, J. I., "Lightweight Transformation and Fact Extraction with the srcML Toolkit", in *SCAM*, Sept 25-26 2011, pp. 10 pages.
- [4] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in *IWPC*, May 10-11 2003, pp. 134-143.
- [5] Collard, M. L., Maletic, J. I., and Robinson, B. P., "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in *ICSM*, Sept 12-18 2010, pp. 10 pages.
- [6] Cordy, J. R., "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation", in *IWPC*, May 10-11 2003, pp. 196-206.
- [7] Cordy, J. R., Dean, T. R., Malton, A. J., and Schneider, K. A., "Source transformation in software engineering using the TXL transformation system", *Info and Software Technology*, vol. 44, no. 13, 2002, pp. 827-837.
- [8] Cox, A. and Clarke, C., "Relocating XML Elements from Preprocessed to Unprocessed Code", in *IWPC*, June 2002, pp. 229-238.
- [9] Dragan, N., Collard, M. L., and Maletic, J. I., "Automatic Identification of Class Stereotypes", in *ICSM*, 2010, pp. 10 pages.
- [10] Klint, P., "How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective", in *IWPC*, May 10-11 2003, pp. 2-12.
- [11] Klint, P., Storm, T. v. d., and Vinju, J., "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation", in *SCAM*, 2009, pp. 168-177.
- [12] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in *ICSM*, September 11-17 2004, pp. 210-219.
- [13] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W., "CIL: Intermediate language and tools for analysis and transformation of C programs", *Lecture Notes in Computer Science* 2002, pp. 213-228.
- [14] Van De Vanter, M. L., "The Documentary Structure of Source Code", *Info and Software Technology*, vol. 44, no. 13, October 1 2002, pp. 767-782.
- [15] Visser, E., "Stratego: A Language for Program Transformation Based on Rewriting Strategies", in *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, 2001, pp. 357-362.
- [16] Waddington, D. and Yao, B., "High-fidelity C/C++ code transformation", *Science of Computer Programming*, vol. 68, no. 2, 2007, pp. 64-78.

# TRINITY: An IDE for The Matrix

Jeroen van den Bos  
Netherlands Forensic Institute (NFI)  
The Hague, The Netherlands  
Email: jeroen@infuse.org

Tijs van der Storm  
Centrum Wiskunde & Informatica (CWI)  
Amsterdam, The Netherlands  
Email: storm@cw.nl

**Abstract**—Digital forensics software often has to be changed to cope with new variants and versions of file formats. Developers reverse engineer the actual files, and then change the source code of the analysis tools. This process is error-prone and time consuming because the relation between the newly encountered data and how the source code must be changed is implicit. TRINITY is an integrated debugging environment which makes this relation explicit using the DERRIC DSL for describing file formats. TRINITY consists of three simultaneous views: 1) the runtime state of an analysis, 2) a hexview of the actual data, and 3) the file format description. Cross-view traceability links allow developers to better understand how the file format description should be modified. TRINITY aims to make the process of adapting digital forensics software more effective and efficient.

## I. BACKGROUND

### A. Maintenance Challenges in Digital Forensics

The storage capacity of digital devices continues to grow. Forensic software is currently required to analyze data in the terabyte range in very short time frames. This requires perfective maintenance to optimize and tune analysis tools. At the same time, corrective maintenance has to be performed when new variants and versions of file formats are encountered. Most of these variants are non-standard, so standards documents cannot be consulted for the required changes. Moreover, the data is often created by proprietary firmware (e.g., of digital cameras) or other types of closed-source applications (e.g., word processors, photo-editing software). As a result, the source code is generally unavailable for inspection.

Corrective maintenance then boils down to reverse engineering the file format variant based on the binary data itself. This process is quite cumbersome, since the structure of the data is not a first class citizen in general purpose programming languages. In hand-coded file processing software, the layout of a binary file format like PNG [1], for instance, is encoded in complex control-flow and (interdependent) data structures. This means that debugging requires ad hoc decoding of values, inspection of input data to check dependencies between values and manually tracking structural layout and ordering.

Besides time consuming, these steps also tend to be error-prone. For example, an off-by-one error in an offset calculation causes a wrong value to be used, but also shifts interpretation of all consecutive values and their dependencies. Such small errors are hard to catch since there are no explicit links between the input data and how the code interprets it.

When adapting existing implementations of file processing software, interactive debuggers can be used, but they are agnos-

tic to the domain-specific aspects of file formats. Furthermore, each file format may have its own conventions such as whether length fields include or exclude marker values, and whether indices are 0- or 1-based. As a result, reverse engineers have to mentally translate the information that is presented to them.

TRINITY is an IDE for reverse engineering binary data which automates a significant portion of this translation. By maintaining semantic links between data, runtime state and code, it becomes possible to *debug the data*, instead of just the code. The key enabler for this is representing file format structure at a higher level of abstraction. DERRIC is a domain-specific language (DSL) that precisely does that [2].

### B. Declarative File Format Descriptions

DERRIC is a domain-specific language to declaratively describe binary file formats. It allows the definition of the components of a file format (called “structures”), their sequential arrangement, and the possible dependencies between elements. For instance, a file format description may contain structure definitions for headers, footers and data blocks. These structures are arranged sequentially according to a (regular) grammar, capturing the layout of a file format. An example of a dependency is when the length of a certain sequence of bytes is constrained by the value of certain bytes elsewhere in the file. DERRIC provides a configurable language for expressing these and other aspects of file formats.

A DERRIC description is divided in two main sections. The first part of a DERRIC description is the sequence section, which consists of a regular expression capturing the sequential layout of a file format. For instance, the following example presents an abridged version of the layout of PNG (where ellipses indicate omitted details):

```
sequence
Signature IHDR
  (...)* PLTE? (...)* IDAT IDAT* (...)*
  bBpN? IEND?
```

The regular operators `*` and `?` have the usual meaning of repetition and optionality. The identifiers (e.g., `Signature`, `IHDR`, etc.) refer to specific components of PNG. These structures are described in the second part of a DERRIC description. As an example, the following snippet describes the `IEND` structure:

```
IEND {
  length: 0 size 4;
  chunktype: "IEND";
  crc: 0xAE, 0x42, 0x60, 0x82
}
```

---

TRINITY is available at <http://github.com/jvdb/trinity>.

This declaration states that the IEND structure consists of a length field of 4 bytes (containing zeros), followed by the (ASCII encoded) string "IEND", and terminated by a CRC code consisting of 4 constant values. To factor out common fields in structure definitions, DERRIC allows structures to inherit from other structures. For instance, in PNG, most structures inherit from an abstract Chunk structure which declares common fields for length, type, data and CRC check; such fields can be overridden if needed.

A DERRIC description is input to the DERRIC compiler which generates executable *validators*. A validator tries to match binary input streams against the file format definition captured in DERRIC. One application of these validators is *file carving*: the process of recovering possibly damaged or fragmented files from storage devices [3], [4]. Previous research has shown that the generated validators perform well, both in terms of recovered files and runtime speed [2], and that DERRIC descriptions can be automatically transformed to improve runtime performance [5].

The benefits of DERRIC are only fully realized, however, if the file format description can be considered correct. If files are encountered that are not recognized, there are two possibilities:

- The binary data is not an instance of the file format we are looking for, or the data is corrupted. In other words, the data is at fault.
- The file format description is incorrect and has to be changed to cope with this specific variation of the file format.

Note that these situations may overlap. In fact, it is quite common to relax a file format description to trade some precision for a higher recall. Nevertheless, in both cases the question remains: how to find out if a description should be adapted to the new situation? And if so, how should the description be changed? TRINITY helps to answer such questions by providing debugger functionality at the level of DERRIC itself. This way, both the data and the runtime state of an analysis can be interpreted in terms of the sequential layout and the structures and fields of the file format.

## II. TRINITY

### A. Integrated Data Debugging

TRINITY is an IDE which aims to leverage the domain-specific information contained in DERRIC descriptions to bring integrated data debugging support to the process of reverse engineering binary file formats. A screen shot of TRINITY is shown in Figure 1. The IDE consists of three synchronized views:

- **Data:** A hexview showing the input data (top right).
- **State:** An outline view of the runtime state, with root nodes for structures and child nodes for fields (left column).
- **Code:** A syntax-highlighting editor for showing a DERRIC description (bottom right).

The user can navigate between views using hyper links which connect all three views. For instance, after selecting the

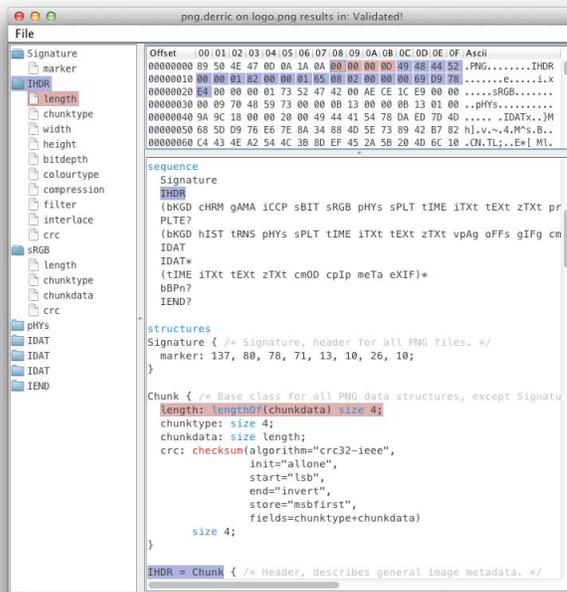


Fig. 1: Screenshot of TRINITY used on a PNG example file.

byte at offset 8 in the Data view at the top right, the contextual structure and field of this byte are highlighted. Similarly, the IHDR structure and its length field are highlighted in the State view on the left, which provides the dynamic execution context to this byte. In the Code view at bottom right, the IHDR structure is highlighted in both the **sequence** and **structures** sections. Finally, the length field is highlighted in the Code view as well, where it is defined not directly in the IHDR structure, but in the Chunk structure it inherits from.

It is also possible to go the other way. For instance, clicking on a field in the code view will highlight all the bytes in the input stream that have been successfully matched using that very field. Similarly, clicking on an element in the sequence section highlights all bytes in the input stream captured by that syntactic element. Because syntactic elements in the sequence may occur multiple times (through the use of the regular operator \*), clicking on a source element may highlight multiple parts of the input data.

Figure 2 illustrates the relationships between the three views in more detail. On the left (Data) is a hexview of the input data (between offsets 16 (0x0010) and 48 (0x002C + 4). In the center (State) the trace of interpreting the input data (showing matches for structures named Header, Config and Data, of which only Config is expanded and showing its fields). On the right (Code) the text editor view of the DERRIC description (showing the definition of the Config structure). In all three views, the dotted line marks the Config structure and the dashed line its storetype field.

By making the links between data, runtime state and code explicit, TRINITY simplifies the reverse engineering and maintenance tasks in dealing with binary file formats. The developer can interactively explore the original file format

Data	State	Code
0x0010 02 FF FF 7F	Header (offset=0x000E, size=14)	structures
0x0014 22 C4 00 FF	Config (offset=0x001C, size=8)	Block { marker:0xFF, !0; }
0x0018 A0 AF 15 BE	marker =0xFF,0x07	Config = Block {
0x001C FF 07 0F BB	storetype =0x0FBB	storetype:0,0x1F00 size 2;
0x0020 02 04 AA 7B	packtype =0x02	packtype:1 2 4;
0x0024 FF 10 00 FF	tabletype =0x04	tabletype:!0;
0x0028 54 FE 3E 23	reference =0xAA7B	reference:size 2;
0x002C BB 32 3F 1B	Data (offset=0x0024, size=259)	}

Fig. 2: The relationship between Data view (left, hexview), State view (center, outline) and Code view (right, text editor).

description in DERRIC directly in the context of the actual bytes in the input data. Below we describe how TRINITY can be used in digital forensics practice.

### B. A File Format Reverse Engineering Scenario

The design of TRINITY is informed by more than a decade of experience in reverse engineering file formats. Additionally, in previous research we have performed an experiment which studied corrective maintenance of DERRIC descriptions [6]<sup>1</sup> by executing evolution scenarios. These scenarios for “fixing” the descriptions all represent typical cases where TRINITY could be used. In fact, the research of [6] would have been much less time consuming if TRINITY had been available at the time, as most of the effort consisted of relating error locations in binary data to source locations in DERRIC.

The use of TRINITY starts when a file is encountered that is expected to validate, but fails to do so. The following steps describe the expected work flow using TRINITY:

1) *Initial Run*: The file and the DERRIC description of its expected file format are loaded into TRINITY and the interpreter halts at the first byte where validation fails. The file’s contents is shown in the Data view, the DERRIC description in the Code view and the generated trace after an initial run in the State view.

2) *Locate Area of Interest*: The user clicks on the last data structure listed in the trace, automatically showing the relevant child nodes. The Data view is automatically scrolled to the corresponding bytes. The cursor in the Code view is positioned on the structure where validation failed.

3) *Inspect Structure*: The user clicks the last field below the structure in the trace. This keeps the existing highlighting but adds additional ones of the fields’ bytes in the Data view and its description in the Code view.

4) *Make Corrections*: Based on whether that field is the source of the validation error, the user will either make a modification or move up to the previous field, backtracking until a field or structure is encountered which accounts for the failure. Finally, the validation is rerun, and the process repeats if there are (new) failures.

## III. IMPLEMENTATION

DERRIC is implemented as an external DSL in the metaprogramming language Rascal [7]. Rascal provides built-in gram-

<sup>1</sup>The changes can be reviewed online at <http://github.com/jvdb/derric-eval/>.

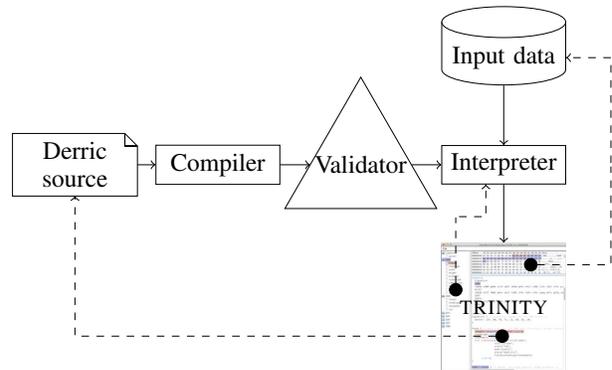


Fig. 3: The TRINITY architecture. The dashed arrows indicate the information sources of the three views in the IDE.

mars for describing syntax, primitives for analyzing and transforming source code, and provides hooks into the Eclipse IDE to obtain editor services (e.g., syntax coloring, outlining, hyperlinking etc.).

The DERRIC compiler operates in three steps. First the DERRIC description is desugared (e.g., flattening inheritance, constant propagation). Second, a DERRIC description is transformed to an intermediate representation called Validator, which is an imperative but platform-independent model of the final validator. Finally, the Validator model resulting from the previous step is transformed to Java source code.

An overview of the architecture of TRINITY is shown in Figure 3. TRINITY reuses the front-end part of the DERRIC compiler, up to and including the transformation to the Validator model. Instead of generating Java code however, the Java foreign-function interface of Rascal is used to build an in-memory model in Java of the Validator. Following the Interpreter design pattern, the classes representing the model contain evaluation methods to execute the validator. This interpreter is then hooked up to the TRINITY IDE.

To realize the fine-grained cross-linking of views in TRINITY, origin tracking is used [8]. This means that the original source locations of syntactic elements in a DERRIC description are maintained throughout all phases of the compiler and interpreter. The DERRIC parser generated by Rascal initially annotates the parse tree with such origins. During desugaring and the transformation to the Validator model, the origins are propagated. Finally, the in-memory model in Java is decorated

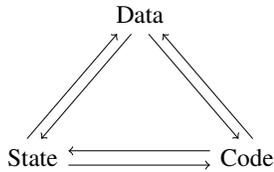


Fig. 4: The trinity of debugging in TRINITY

so that, when the interpreter is stopped, the TRINITY runtime environment knows where in the DERRIC description execution is taking place. The same technique is used to maintain a mapping from the runtime state (i.e. the values of the matched structures and fields), to the source code, and from the source code to the data.

#### IV. RELATED WORK

The key idea of TRINITY is to integrate the input data into the activity of debugging and to provide bidirectional cross-links among code, state and data. Moreover, the integration is domain-specific: DERRIC descriptions capture file formats at a level that can be understood by forensic investigators. In TRINITY this understandability extends to the data and the runtime state of the validator. As a result, TRINITY provides debugging for reverse engineering file formats at a higher level of abstraction.

Using TRINITY the user can navigate from the source code to the data and vice versa, but also from the runtime state to the data and vice versa, and finally, it is possible to go from the data to the runtime state and the source code. We have depicted these 6 types of cross links in Figure 4. Traditional debuggers, on the other hand, provide only two of such links: 1) from the runtime state (e.g., stack trace) to the source code, and 2) from the data to the code (e.g., from a variables view to declaration sites). Although specialized visualizations for general purpose debuggers are quite common (e.g., [9], [10]), these do not provide the same level of integration as TRINITY.

TRINITY is most related to domain-specific debuggers in other domains. For instance, ANTLRWorks [11] is an IDE which provides support for debugging ANTLR grammars. The generated parsers communicate with the IDE during their execution, allowing the user to replay its actions and inspect the input data, grammar and parse tree afterwards. Similar tools exist for debugging regular expressions. A recent example is Debuggex [12], which features coloring of the (matched) input data, and visualization of the finite-state automaton.

#### V. CONCLUSION AND FUTURE WORK

Reverse engineering binary file formats is a time-consuming and error-prone activity. One of the reasons is that the relation between the structure of the data and how software processes that data is obscured by low-level implementation details and has to be mentally reconstructed. In this paper we have presented TRINITY, an IDE that brings integrated data debugging support to the DERRIC IDE for file format description. It consists of three views, which display the input data, the runtime state of a file format validator and the DERRIC source code respectively. Each view is related to the other. Clicking

in any of the views highlights corresponding elements in the others. If a file fails to validate, the three integrated views allow the developer to assess the situation: why does validation fail? What changes are needed to the file format description? TRINITY aims to reduce the effort of performing corrective maintenance of digital forensics software.

There are ample opportunities for further improving TRINITY. For instance, the way elements are highlighted in the different views is mostly syntactic. One extension would be to add more semantics to the visualization. For instance, clicking a field that has a dependency on another field in its length or content specification, could also highlight the bytes that were captured by those dependency fields. Conversely, clicking on a byte in the data view could also trigger highlighting of all expressions affected by it. Such data flow visualization could further increase understanding of what happens at runtime and help diagnosing failures.

Another direction for further work is increasing the “liveness” of TRINITY [13]. Currently, TRINITY allows the dynamic inspection of state and data. However, changes to the DERRIC description still requires a full rerun of the validator. The potential benefits presented by TRINITY could be increased further by instantly reflecting a change to the format description in the other views. One way to approach this is to incrementally update the runtime state of the interpreter based on the changes to the code (see, e.g., [14]).

Finally, we plan to perform a user study to evaluate to what extent TRINITY helps to improve the maintenance of DERRIC descriptions. The evolution scenarios obtained in [6] can provide a starting point for the maintenance tasks to set up this experiment.

#### REFERENCES

- [1] W3C, “PNG Specification,” 2003, <http://www.w3.org/TR/PNG/>.
- [2] J. van den Bos and T. van der Storm, “Bringing Domain-Specific Languages to Digital Forensics,” in *ICSE’11*. ACM, 2011, pp. 671–680.
- [3] M. I. Cohen, “Advanced Carving Techniques,” *Digital Investigation*, vol. 4, no. 3-4, pp. 119–128, 2007.
- [4] A. Pal and N. Memon, “The Evolution of File Carving,” *Signal Processing Magazine*, vol. 26, no. 2, pp. 59–71, 2009.
- [5] J. van den Bos and T. van der Storm, “Domain-Specific Optimization in Digital Forensics,” in *ICMT’12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 121–136.
- [6] —, “A Case Study in Evidence-Based DSL Evolution,” in *ECMFA’13*, ser. LNCS, vol. 7949. Springer, 2013, pp. 207–219.
- [7] P. Klint, T. van der Storm, and J. Vinju, “Rascal: A Domain Specific Language for Source Code Analysis and Manipulation,” in *SCAM’09*. IEEE, 2009, pp. 168–177.
- [8] A. v. Deursen, P. Klint, and F. Tip, “Origin tracking,” *Journal of Symbolic Computation*, vol. 15, pp. 523–545, 1993.
- [9] A. Zeller and D. Lütkehaus, “DDD - A Free Graphical Front-End for UNIX Debuggers,” *SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, 1996.
- [10] Hex Rays, “IDA,” <https://www.hex-rays.com/products/ida/index.shtml>.
- [11] J. Bovet and T. Parr, “ANTLRWorks: an ANTLR grammar development environment,” *Softw., Pract. Exper.*, vol. 38, no. 12, pp. 1305–1332, 2008.
- [12] S. Toarca, “Debuggex,” <http://www.debuggex.com/>.
- [13] H. Lieberman and C. Fry, “Bridging the Gulf between Code and Behavior in Programming,” in *CHI’95*. ACM, 1995, pp. 480–486.
- [14] T. van der Storm, “Semantic deltas for live DSL environments,” in *LIVE’13*. IEEE, 2013, pp. 35–38.

# E-Xplore: Enterprise API Explorer

Allahbaksh M. Asadullah, Basavaraju M., Nikita Jain

Infosys Labs, Infosys Ltd.

Bangalore, India

Email: {allahbaksh\_asadullah, mbraju, nikita\_jain02}@infosys.com

**Abstract**—Plenty of open source libraries and frameworks are available for developers these days for reuse in their projects. However the difficulty in finding and reusing the correct API among the hundreds of available APIs far outweighs the advantage of saving time. The problem is acute in enterprise codebase. Online forums like StackOverflow are no help for enterprise source code as they are closed in nature. We have developed a tool called E-Xplore that addresses this issue by letting the programmers search in large source codebase and browse them effectively. The tool also provides related artifacts in the form of result clustering. We evaluated E-Xplore with other tools via user study with developers working on an enterprise banking system with more than 10 million lines of code. A set of common tasks was given to the developers with and without the tool. We observed that the tool offered appreciable time and effort benefits in large scale software system development and maintenance. In this paper we describe the tool and its features which help develop and maintain source code effectively.

**Index Terms**—E-Xplore, API, search, source code, semantic search, API Exploration

## I. INTRODUCTION

Developers spend much of their time exploring the APIs. Over time, there has been tremendous increase in the number of open source projects. In this modern computing era and rapid application development environment, the number of APIs and frameworks is increasing every day. Exploring these new APIs or using them efficiently in projects is difficult and time consuming. Many systems have sub-systems and developers focus on exploring only the sub-system they work with. They face problems in finding and understanding APIs in other sub-systems having different modules. Using web search engines, code search engine, StackOverflow, and blogs to understand the code better works well with the open source projects or popular commercial libraries since those have already been explored by others. But it's not much help in case of enterprise software, for example a banking system.

Generally developers try to find the way a particular function or a class has been defined and used, the relationship among the classes (is-a, has-a, etc.), the source of a particular method etc. The need can be to locate a re-usable code, to find a possibility of common bug in all the locations, for modifying the code, or even to locate an example of usage of a particular API.

Developers refer to different documents to understand the code and the systems. They talk to other developers distributed across geographical locations. Segal [1] observed that professional end users don't voluntarily produce documentation. It is difficult to comprehend the code with limited or no

documentation available. Developers also refer to JavaDoc and the sample code written by other developers. Previous studies [2], [3] have suggested that these JavaDocs are not enough to learn about the API quickly. For example if there is a Factory Method to instantiate an object, it would be difficult to search through the JavaDoc and find out how to use it.

This implies that empowering developers with proper tools that help them to easily search and explore the API is important in enterprise environment. These tools will not be used much for the open source libraries because many code search engines are already present, and support exists in the form of StackOverflow and similar forums. Our tool E-Xplore is primarily designed for use in enterprise to help maintain internal products and customer legacy projects. It has features that help developers locate the API and find their examples. It also helps developers find out related APIs with the help of semantic search.

The philosophy behind E-Xplore is as follows. The UI design and the functionality of E-Xplore is heavily inspired by the Eclipse IDE, one of the most widely used open source IDEs for developing Java projects. Developers do not take much time to get familiar with the tool as its look and feel is very close to that of Eclipse.

The rest of the paper is organized as follows: Section II gives the related work. In section III, we describe the features of the tool. Section IV briefly explains the architecture. Section V presents the study. We conclude the paper in section VI with future work.

## II. RELATED WORK

Roehm et al. [4] found that developers believe in source code rather than documentation because often, documentation is inaccurate, or even incomplete. As a result, developers use JavaDoc sparingly. Tools like Sourcerer [5] and Prospector [6], which focused on locating examples of API in different contexts, fail when developers do not have sufficient knowledge to construct a context for generating examples. Codelets [7] is a plugin to the IDE that helps to program effectively by giving sample examples for API, but it focuses more on developing, rather than analyzing or exploring the API. Also, it is difficult to develop and maintain such a collection of examples in enterprise. Other tools like CodeWeb [8] and SpotWeb [9], focusing on usage patterns of the given API library in open source frameworks, are difficult to use in enterprise source code.

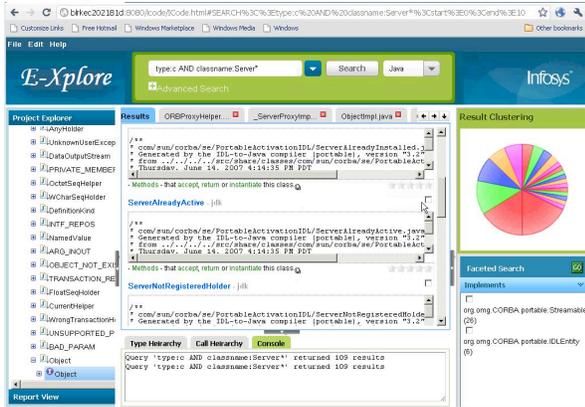


Fig. 1. E-Xplore Web UI

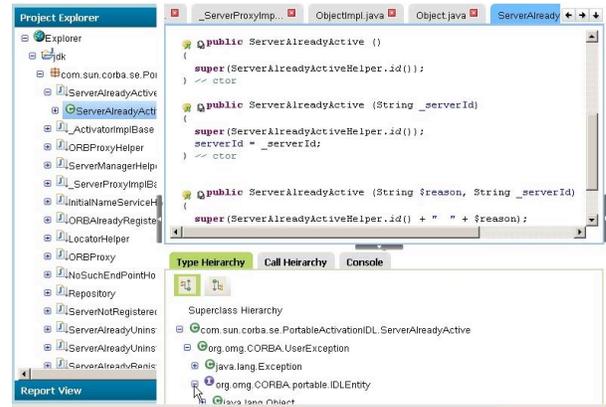


Fig. 2. Type hierarchy, call hierarchy and package explorer

Jadeite [10] uses information on API usage to make its documentation easier to navigate. Additionally, it allows users of the API to add method templates that were not part of the original API. Some development recommendation systems have applied machine-learning techniques to help programmers automatically find example code relevant to their projects [11]. David et al. [12] presented a tool, Apatite, for visualizing the API better through common associations between the APIs. Instead of conventional top-down hierarchical display of packages, classes, and methods, Apatite lets the users search across any level of API hierarchy and also presents the most popular items related to the selected item. Seyed Mehdi Nasehi et al. [13] supplement the standard API documentation by extracting examples from unit test cases of the API. This approach relies on having well-written test suite for all the API libraries, which may not be always the case. API-Explorer [14] leverages the structural relationships between API elements to recommend API methods or types which are not directly accessible from a given API type. All these tools address specific issues but they lack a complete integrated solution which helps explore the API along with source code in a developer-friendly way. In the present work, we built a tool based on the way professional developers comprehend the source code [4].

### III. FEATURES

E-Xplore has three types of searches to explore the APIs: simple boolean query search, faceted search, and semantic search. Simple boolean query search is performed for standard Java class exploration like what all functions are returning an object of particular class or interface type. For example, if the developer is looking for all the methods which are returning an object of type `ILoan`, then the query would be *type:method AND returns:ILoan*.

Faceted Search is an accordion feature visible only after simple query. For example, if user searches for *type:class AND classname:IAccount\** it returns all the classes starting with `IAccount` along with extra information to localize the results by using predefined facets like *implements*, *ex-*

*tends*, *calls*. The accordion is populated with the *implements com.infy.finacle.account.IAccount*. The third and important type of search is Semantic search. Here, we apply LDA to extract business concept from source code. More details about the approach are given by Rama et al. [15]. This helps in finding the related classes. For example, a search for *Loan* would give classes like *Interest*.

Tobias Roehm et al. [4] conducted studies on how developers comprehend source code. In their studies, they found that *developers trust source code more than the documentation*. This is the precise reason why we provided a source code browse feature in E-Xplore. We have also provided dependency and structural information through Hierarchy Explorer and Call Explorer. This is a key feature which helps developers comprehend the source code as per hypothesis 22 of [4]. E-Xplore also addresses the hypothesis 10 of [4] which says that developers use notes as comprehension support. In E-Xplore, on each method and class, the developer finds a note icon which they can use to annotate the method / class. These notes can be searched later by using a keyword search.

The whole intent of our work was to create a single tool with integrated features which addresses the developers' complaint of loose integration of the tools which hinder comprehension [16], [17].

### IV. ARCHITECTURE

E-Xplore is independent of Eclipse IDE and is platform-independent. The client requires an HTML5 compatible browser to use it. Figure 3 shows the architecture diagram of E-Xplore. Below is the description of its major components.

**Source Code Indexer:** The source code indexer parses the source code using Eclipse JDT parser to extract the code elements, their attributes, and their dependencies. It stores the extracted elements in the index. It also converts the source code file into an equivalent HTML which has anchor tags for cross navigation between the sources. These files are stored in the Document Server.

**Search Server:** The search server stores the index and returns the search result based on the query. This has a clus-

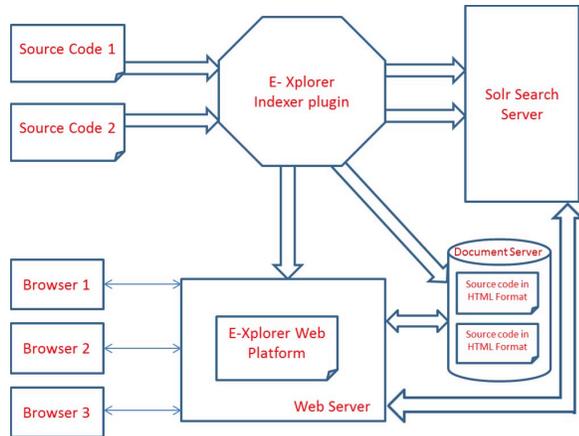


Fig. 3. Architecture diagram

tering component which clusters the search results. Clustering is based on the work of Rama et al. [15]. We use Apache Solr as the search server. This server provides RESTful interface.

**Document Server:** The document server stores the HTML files generated by the Source Code Indexer. Meta data files are stored for each package, class, and project. These are used to populate this package explorer.

**Web Interface:** Figure 1 shows the web interface of E-Xplore which mimics Eclipse UI. It has a package explorer, Type and Call Hierarchy and an area where source code is shown. The web interface posts the query to server, which in turn fetches results from search server. The search results are displayed in the center of the interface. The results are formatted and provided with links at the bottom of each result to sub query. When a user clicks the search result it shows the resulting file served by the Document Server.

## V. STUDY

### A. Method

We ran a comparative study to evaluate E-Xplore. We gave a set of 5 tasks to the developers. There were four groups, two with E-Xplore and the other two with standard tools like Eclipse Search, `grep`<sup>1</sup> and find in file. Participants used Eclipse 3.8 on Windows 7. We used Camtasia<sup>2</sup> to record the event. This study was carried out for 5 days and the duration of the tasks was 1 hour or 2 hours. We gave the tasks from different modules of the software system every day.

### B. Participants

The study was carried out on 32 developers, for 5 consecutive days and the task required them to understand and make changes in the source code. Participants had 0 to 8 years of industry experience. We selected the participants based on the questionnaire prepared to judge their proficiency in basic Java programming and the application. They were required

to score a minimum of 7/10 to participate in the study. They were equally divided into four groups. Two groups consisted of expert programmers (3 to 8 years of experience) and the other two consisted of novices (less than 3 years of experience). One group of novice programmers and one group of experts were given E-Xplore and the remaining two groups were given standard tools.

### C. Task Design

The tasks were designed to analyze how effectively the developers used E-Xplore and other standard tools. We describe the tasks here.

**Task 1:** The first task was meant to be as simple as possible so that the developers could get acquainted with the tools. Their goal was to finish the task as soon as possible. We gave the following simple tasks to perform in 1 hour. a) Find all the classes which implement particular interface (say `ICustomer`), b) Find methods which call a particular method (say `createCustomer`), c) Find a class which extends a class (say `SavingAccount`) and has a method (say `getAccountBalance`), d) Find all classes which extend `CustomerDAO` and implement `Serializable`, e) Find all methods whose return type is `AccountVO` and resides in class which starts with `Account`.

**Task 2:** The second task had a higher level of difficulty. The task involved searching through the source code and finding the depth of inheritance of some classes, usage of some methods, and related classes in an hour.

**Task 3:** This task had a similar level of difficulty and time duration as task 2. The participants were asked to remove a single method interface say `IFoo1`. This interface was extending another interface say `IFoo`. Many classes were implementing either `IFoo` or `IFoo1`. We asked the developers to give their suggestions on how they would change the source code without impacting the functionality. Note that we did not consider unit test case changes as part of this task.

**Task 4:** This task involved knowledge of Abstract Factory pattern and was carried out for 2 hours. We asked the developers to use several classes to create a Report. The task essentially involved less than 20 lines of code, but it required them to understand the API of the present system well. Some of the classes involved were `DocumentFactory`, `PDFDocumentFactory`, `HTMLDocumentFactory`, `IAccountStatement` etc. This task was expected to be tough for the developers as it involved knowledge of different classes and interfaces.

**Task 5:** This task required the developers to have prior knowledge of AbstractFactory and Observer Pattern. The task was something like “Write a class which contains a logic to send an email, when the account balance of a customer falls less than 10,000.” According to hypothesis 4 of [4], the developers usually run the system to acquire knowledge related to it. As this information is difficult to acquire in large distributed software system by running it, we provided them with related information on a piece of paper. The classes, viz. `Message`, `ICustomerNotification`,

<sup>1</sup><http://www.wingrep.com/>

<sup>2</sup><http://www.techsmith.com/camtasia-features.html>

TABLE I  
OBSERVATIONS

Task No	Max Time Given (Hrs.)	Average Time Taken (Minutes)			
		Novice		Expert	
		E-Xplore	Standard Tools	E-Xplore	Standard Tools
1	1	50	59	35	40
2	1	52	60	40	47
3	1	60	X	45	55
4	2	110	120	75	95
5	2	120	X	90	110

AccountNotificationManager, EmailFactory along with two lines of explanation for each were provided to the developers. This task had the highest level of difficulty in the study and the duration was two hours.

#### D. Results

Table I shows the average time (in minutes) taken by all the groups for each task. The generalized observation is that the group that worked with E-Xplore performed better than the one which worked with other tools in case of both novices and experts. The participants who did not use E-Xplore found it difficult to complete the tasks. Completion of task means that participants finish the task correctly within the given period of time. For example, novices with standard tools could not complete two tasks (represented by X in Table I).

Participants used various features of E-Xplore to perform better at tasks. For task 2, for example, 7 out of 16 participants of E-Xplore used the faceted search feature to find the depth of inheritance. The other participants spent time using hierarchy explorer to find the inheritance depth. For Task 3, 10 out of 16 developers used the faceted search, the rest used query *type:c AND implements:IFool*. This task finds its applicability in the refactoring for Java 8<sup>3</sup>.

Task 4 and 5 involved deeper knowledge of the API. It required developers to understand the current system and reuse the classes and methods. This task was difficult for novices because of the lack of design pattern knowledge.

## VI. CONCLUSION AND FUTURE WORK

We believe E-Xplore can help explore and analyze large projects in an enterprise. It will reduce the development time and effort. Overall, it will also increase reuse and reduce cloning of the code.

Although we initially targeted the E-Xplore for API exploration, we believe it can be used by the architects to study the source code and also draw semantic information about it. Many participants with more than 6 years of experience showed interest in E-Xplore.

Our work is still in progress and we would like to improve the semantic search and clustering of the results. We are exploring how to better the search and user interface through user studies. We plan to integrate the E-Xplore search and clustering feature with Eclipse search.

<sup>3</sup><http://jcp.org/en/jsr/detail?id=335>

## ACKNOWLEDGMENT

We would like to thank Dr. Srinivas Padmanabhuni for his guidance at different stages. We extend our gratitude to Mrs. Roli Saraswat for her review comments. We also extend our thanks to various business units within Infosys who provided support for experiments.

## REFERENCES

- [1] J. Segal, "Some problems of professional end user developers," in *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*. IEEE, 2007, pp. 111–118.
- [2] J. Beaton, S. Y. Jeong, Y. Xie, J. Stylos, and B. A. Myers, "Usability challenges for enterprise service-oriented architecture APIs," in *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 193–196. [Online]. Available: <http://dx.doi.org/10.1109/VLHCC.2008.4639084>
- [3] S. Y. Jeong, Y. Xie, J. Beaton, B. A. Myers, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse, "Improving documentation for esoa APIs through user studies," in *End-User Development*. Springer, 2009, pp. 86–105.
- [4] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.
- [5] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 681–682.
- [6] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 48–61.
- [7] S. Oney and J. Brandt, "Codelets: linking interactive documentation and example code in the editor," in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*. ACM, 2012, pp. 2697–2706.
- [8] A. Michail, "Code web: data mining library reuse patterns," in *23rd international Conference on Software Engineering, Toronto*. IEEE, 2001.
- [9] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 327–336.
- [10] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*. IEEE, 2009, pp. 119–126.
- [11] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *Software, IEEE*, vol. 27, no. 4, pp. 80–86, 2010.
- [12] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite: A new interface for exploring APIs," in *Proceedings of the 28th international conference on Human factors in computing systems*. ACM, 2010, pp. 1331–1334.
- [13] S. M. Nasehi and F. Maurer, "Unit tests as API usage examples," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.
- [14] E. Duala-Ekoko and M. P. Robillard, "Using structure-based recommendations to facilitate discoverability in APIs," in *ECOOP 2011—Object-Oriented Programming*. Springer, 2011, pp. 79–104.
- [15] G. M. Rama, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *ISEC, 2008*, pp. 113–120.
- [16] W. Maalej, "Task-first or context-first? tool integration revisited," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 344–355.
- [17] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 174–188.

# Browserbite: Accurate Cross-Browser Testing via Machine Learning Over Image Features

Nataliia Semenenko, Marlon Dumas

Institute of Computer Science  
University of Tartu, Estonia  
{nataliia, marlon.dumas}@ut.ee

Tõnis Saar

Browserbite and Software Technology and Applications  
Competence Center, Estonia  
tonis.saar@stacc.ee

**Abstract**— Cross-browser compatibility testing is a time consuming and monotonous task. In its most manual form, Web testers open Web pages one-by-one on multiple browser-platform combinations and visually compare the resulting page renderings. Automated cross-browser testing tools speed up this process by extracting screenshots and applying image processing techniques so as to highlight potential incompatibilities. However, these systems suffer from insufficient accuracy, primarily due to a large percentage of false positives. Improving accuracy in this context is challenging as the criteria for classifying a difference as an incompatibility are to some extent subjective. We present our experience building a cross-browser testing tool (Browserbite) based on image segmentation and differencing in conjunction with machine learning. An experimental evaluation involving a dataset of 140 pages, each rendered in 14 browser-system combinations, shows that the use of machine learning in this context leads to significant accuracy improvement, allowing us to attain an F-score of over 90%.

**Keywords**—cross-browser testing; machine learning; image processing

## I. INTRODUCTION

A long-standing issue in Web application development is that a given Web page may be rendered differently in different browsers or systems depending on the rendering engine, screen resolution, font, etc. Some of these differences are minor or even imperceptible, but others constitute layout, formatting or functional incompatibilities. Manual detection of these incompatibilities by means of visual inspection is labor-intensive and error-prone, as fatigue may cause testers to miss incompatibilities. Depending on a Web site's popularity, the number of browser-system combinations that need to be tested in order to cover 90-95% of users can go up to 20-30 [1]. This is a major hindrance for Web application maintenance.

A number of automated cross-browser testing prototypes, such as CrossCheck [2] and WebDiff [3] as well as commercial tools such as Mogotest<sup>1</sup> and Browsera<sup>2</sup> significantly reduce the required amount of manual effort by automating the screenshot capture and comparison steps. However, these systems suffer from over-sensitivity and produce an excessive amount of false positives. For instance,

an evaluation of CrossCheck showed 64% of false positives, while for WebDiff this number reached 79% [2]. Fundamentally, this is due to the fact that what constitutes an incompatibility, as opposed to a simple difference, is to some extent subjective [2]. Thus, setting specific thresholds to classify a difference as an incompatibility is far from trivial.

This paper reports our experience building an industrial-strength cross-browser compatibility testing tool, namely Browserbite<sup>3</sup>, by combining image processing techniques with machine learning. One of the novelties of Browserbite is that rather than attempting to set thresholds for classifying image discrepancies as incompatibilities during image comparison, the task of classifying differences as incompatibilities is almost entirely pushed to the machine learning phase. We evaluate the accuracy of neural networks and classification trees for this task, based on a dataset of 140 popular Web pages rendered across 14 configurations.

The rest of the paper is structured as follows. Section 2 gives an overview of the Browserbite system. Section 3 introduces the machine learning approach to post-process the output of Browserbite's image comparison module. Next, Section 4 discusses the evaluation results, while Section 5 concludes and discusses directions for future work.

## II. BROWSERBITE

The aim of Browserbite is to detect potentially incompatible renderings of a given Web document (identified by its URL) across different browsers and operating systems (OS). At present, Browserbite supports 14 browser-OS combinations (called *configurations*), covering major versions of popular browsers (Chrome, Firefox, IE, Safari) running on Windows XP, Windows 7 and Mac OS.

Browserbite's architecture comprises three main modules: screenshot capture, screenshot comparison and classification. The screenshot capture module consists of a scheduler that controls a number of workers. The workers are instances of different types of Virtual Machines (VMs). Each worker is capable of taking full-page screenshots of Web pages in a given configuration.

<sup>1</sup> <http://mogotest.com>

<sup>2</sup> <http://browsera.com>

<sup>3</sup> <http://www.browserbite.com>

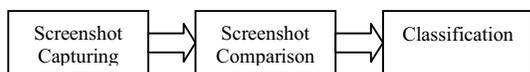


Fig. 1 Browserbite main modules

Web pages are opened and rendered using Selenium.<sup>4</sup> Full-page capture is achieved via image stitching or window resizing depending on the configuration. In the first case, the Web page is automatically scrolled and a screenshot is taken after each scrolling operation. The resulting partial-page screenshots are stitched together into a full Web page image. In the second case the browser window is resized to match the size of the web page document, and a single full Web page screenshot is taken. The latter method has been found to work with IE, while the former method is used for other browsers.

The screenshot comparison module relies on image segmentation and comparison techniques. Its input is a collection of screenshots (images). One of these images is designated by the user as the *baseline image*, while other images are called *Images Under Test (IUTs)*. The baseline image is meant to correspond to a correct rendering of the Web page. IUTs are compared against the baseline image.

Each image is first segmented into smaller rectangular regions called Regions of Interest (ROI), based on borders and color changes in the image. The set of ROIs extracted from the baseline image is matched with the set of ROIs extracted from each of the IUTs. An ROI in the baseline image (called an ROIB) is mapped to at most one ROI in the IUT (an ROIT). The matching of an ROIB to an ROIT is based on a correlation-based image comparison technique. The ROIB and ROIT are correlated and a *correlation index* is extracted. This index captures the similarity between the pair of ROIs. An ROIB and ROIT are declared compatible if their similarity is above a certain threshold. Otherwise, Browserbite reports the pair (ROIB, ROIT) as a *potential incompatibility*. Every ROIB not matched to an ROIT (*missing ROI*) is reported as an incompatibility of the form (ROIB,  $\perp$ ), where  $\perp$  is the null value. Conversely, an ROIT not matched to any ROIB (*additional ROI*) is reported as a pair ( $\perp$ , ROIT). When reporting a potential incompatibility, Browserbite superposes the ROIB to the ROIT using 50% transparency, enabling users to check the extent of the incompatibility. For instance, Fig. 2 shows a true positive, while Fig. 3 shows a false positive.



Fig. 2. Difference corresponding to an actual incompatibility (www.sourceforge.net). The text element presented in the baseline is absent in the IUT; instead IUT contains another text.



Fig. 3. Minor difference that users did not classify as an incompatibility (www.rik.ee)

An empirical evaluation of an early pre-commercial version of Browserbite by Tamm [4] showed a rate of false positives of around 10% and a rate of false negatives of 44%. Attempts to manually fine-tune the correlation index threshold and other parameters used during image comparison turned out to be unproductive, as decreases in false negatives led to considerable increases in false positives. Given the strong impact that false negatives can have in commercial usage, we decided to increase the sensitivity of Browserbite in the first commercial release of the tool, in such a way that false negatives rate is close to zero, at the expense a high rate of false positives.

Feedback collected during commercial usage confirmed that Browserbite is oversensitive and practically does not miss any incompatibility. To validate this observation, we conducted another experiment in which the first author of the paper manually compared pairs of Web pages rendered on different configurations. A total of 140 Web pages (dataset described below) were analyzed and each was rendered in four browsers (Chrome 22.0, IE8 and Firefox 3.6 and 16.0.1 on Windows 7). The resulting screenshot pairs were manually compared by the first author. We found that 98% of incompatibilities detected manually were reported by Browserbite in the form of ROI pairs (i.e. 98% recall), but the precision on the other hand was in the order of 66%.

Accordingly, we decided to supplement the screenshot comparison module with a classification module in which machine learning is used to reduce the false positive rate. The next section discusses the classification module.

### III. CLASSIFICATION MODULE

The aim of the classification module is to classify potential incompatibilities reported by Browserbite's screenshot comparison module into two categories: true positives (the potential incompatibility is perceived as such by a user) and false positives (the potential incompatibility is not perceived as such by a user). Below we present the datasets used for training/testing classification models, the employed features and machine learning techniques.

#### A. Dataset and Golden Standard

We collected a dataset consisting of home pages of the top 140 Websites of Estonia according to Alexa<sup>5</sup>. Each Web page was given as input to Browserbite, which generated around 20000 potential incompatibilities (ROI pairs). The first author trimmed down this set to 1200 potential incompatibilities by manually identifying 600 pairs that were likely to be true incompatibilities and 600 pairs that were likely to be false positives. This classification by the first author was only used to extract a balanced subset of samples. The judgments made by the first author were discarded in the subsequent evaluation – only the set of 1200 ROI pairs was kept.

We recruited 40 subjects through social media and asked them to classify pairs of ROIs into the two classes: “no difference or insignificant difference” and “major difference”. Subjects were asked to put a pair in the first class if either they

<sup>4</sup> <http://seleniumhq.org>

<sup>5</sup> <http://alexa.com>

noticed no difference at all, or they noticed a minor layout difference, which in their opinion would not affect their perception of a Web page containing that segment. Otherwise they were instructed to classify the pair in the second category.

Respondents were University students in the range of 20-25 years from 6 countries (Estonia, Russia, Ukraine, Germany, Italy and Hungary). The subjects came from different specialties: 60% with the IT background, 20% with economics and business background, 10% with philological background, and 10% others. Each respondent classified between 200 and 400 (ROIB, ROIT) pairs randomly sampled from the dataset with replacement. On the end, we obtained at least 8 classifications for each pair (up to 15 in some cases). For uniformity, we randomly trimmed the dataset so that each (ROIB, ROIT) had exactly 8 judgments (i.e. 8 subjects per pair). The final dataset contained 50.4% of true positives and 40.6% of false positives.<sup>6</sup> The inter-rater reliability of the resulting dataset is 0.94<sup>7</sup>, indicating little disagreement between judges. We aggregated the judgments by marking a potential incompatibility as a true incompatibility if at least 5 subjects rated it as a “major difference”.

### B. Feature Set

We recall that an incompatibility reported by Browserbite consists of a pair (ROIB, ROIT) where ROIB is an ROI in the baseline image and ROIT is a corresponding ROI in the IUT. In case of a missing or additional ROI, ROIT and ROIB can take null values. Given a pair (ROIB, ROIT), we extract, the following 17 features to build a sample for constructing classification models:

- 10 *histogram bins* (h0, h1, ... h9). These 10 integers encode the image histogram of the ROIB. 10 discrete bins represent pixel intensity distribution across the entire ROI image;
- Correlation between the ROI in the baseline image and ROI in the IUT. This is a number between zero and one. It is close to zero in case of very low correlation between ROIB and ROIT. It is zero in case of a missing or additional image.
- Horizontal and vertical position of the ROIB (X and Y coordinates) on the baseline image;
- Horizontal and vertical size of ROIB (width and height) of the baseline image;
- Configuration index – a numerical identifier of the browser-platform combination of the IUT. Browserbite supports 14 browser-platforms combinations, thus this is an integer between 1 and 14;
- Mismatch Density  $MD = E / T$ , where E is the number of ROIs in the IUT that are not matched 100% to an ROI in the baseline image, and T is the total number of

<sup>6</sup> False negatives were identified by the first author separately.

External subjects were used to classify true and false positives.

<sup>7</sup> Calculated using the Inter-Rater Reliability Calculator at <http://www.med-ed-online.org/rating/reliability.html> which implements the measure in [5]

ROIs in the IUT. This is a feature of the IUT itself rather than of an ROI inside the IUT. However, for the sake of convenience when constructing the machine learning models, we make the MD a feature of each ROI. All ROIs extracted from the same IUT will have the same MD (the MD of their enclosing IUT).

### C. Machine Learning Techniques

We explored two popular machine learning techniques for classification: classification trees (i.e. decision tree) [6] and artificial neural networks [7]. Specifically, we used the implementations of these techniques provided in Matlab.

The use of classification trees is motivated by the fact that they provide a convenient way to interpret the model. By analyzing the classification tree, we can obtain insights into the thresholds that determine whether a potential incompatibility is an actual incompatibility or not.

Neural networks imitate the brain's ability to sort out patterns and learn from trials and errors, discerning and extracting the relationships that underlie the data with which it is presented. Studies have shown that neural networks are a promising alternative to standard classification methods [7]. In this respect, a key advantage of neural networks is their ability to adjust themselves to the data without any explicit specification of functional or distributional form.

We selected the 3-layered feed-forward neural network. The first layer (input layer) consists of 17 neurons corresponding to the number of features. The output layer has 2 neurons (binary classification). As the dataset is not linearly separable one or more additional “hidden” layers are needed. In practice, very few problems that cannot be solved with a single hidden layer can be solved by adding another hidden layer [8]. Accordingly, we chose one hidden layer.<sup>8</sup>

The number of neurons in the neural network is another important parameter, as too few hidden neurons can cause underfitting so that the neural network cannot learn the details. Conversely, a too large number of hidden neurons can cause overfitting, as the neural network starts to learn insignificant details. Accordingly, the number of hidden neurons was determined experimentally. In order to determine the appropriate number of hidden neurons we applied empirically derived rules-of-thumb. One of the most common is that the number of hidden neurons should be around the mid-point between the size of the input and size of the output layers [9]. As the number of input neurons equals 17 (the number of features) and the number of output neurons equals 2 and 4 for binary and quaternary classification respectively, we tried to find an optimal number of hidden neurons between 8 and 13. To this end, we trained the neural network with different number of neurons and calculated the F-score for each trained model, using a set of 200 (ROIB, ROIT) samples not used in the subsequent evaluation. We experimentally found that the peak in F-score is reached for a number of hidden neurons of 11. This number was used in the evaluation reported below.

<sup>8</sup> Additional experiments conducted after the evaluation reported here confirmed that adding a hidden layer does not improve F-score.

Using the golden standard described above, we compared the classification accuracy of Browserbite without machine learning post-processing, with that of Browserbite post-processed with a classification tree and Browserbite post-processed with a neural network. Classification accuracy is measured in terms of precision, recall and F-score with their standard definitions [10]. The machine learning models were trained and evaluated using a five-fold cross-validation method. In other words, the dataset was partitioned into five equal parts, four parts were used to train a model and the remaining one was used to test the model. This process was repeated 5 times with each part playing the testing role once. The results from each fold were averaged to produce a single measurement of precision, recall and F-score for each method (classification tree and neural network).

Additionally using the same dataset, we also evaluated Mogotest – a commercial tool for cross-browser compatibility testing based on analysis of Document Object Models (DOM).

#### IV. EVALUATION RESULTS

The evaluation results are summarized in Table 1. It can be seen from the tables that neural networks outperform by far classification trees. The neural network achieves a very high precision at the expense of some degradation in recall. The improvement in precision provided by classification trees is less significant, and comes at the expense of a drop in recall.

TABLE 1. ACCURACY FOR BROWSERBITE W/OUT CLASSIFICATION, MOGOTEST, BROWSERBITE + CLASSIFICATION TREE AND BROWSERBITE+ NEURAL NETWORK

Measure	Plain Browserbite	Mogotest	Classification tree	Neural network
Precision	0.66	0.75	0.844	0.964
Recall	0.98	0.82	0.792	0.886
F-score	0.79	0.78	0.81	0.923

These results are a significant improvement with respect to state of the art techniques such as CrossCheck [2], which achieves a precision of 36% (64% of false positives) and its predecessor WebDiff, with a precision of 21% according to an evaluation reported in [2]. This latter evaluation of CrossCheck and WebDiff is focused on false positives (false negatives are not reported). Assuming 100% recall, these results imply an F-score of 52% and 35% respectively. Given the differences in experimental setups and evaluation goal, no conclusive comparative statements can be drawn, but the results suggest that Browserbite enhanced with a neural network-based classification module achieves high accuracy relative to state-of-the-art techniques.

#### V. CONCLUSION

This paper presented and evaluated the Browserbite cross-browser testing tool with an emphasis on its classification module. The results show that neural networks for incompatibility classification provide a high level of accuracy in this context (precision of 96% with recall of 89%) outperforming classification trees. Given the improvements

achieved, the neural network technique has been productized and included in Browserbite's private beta version (to be released in the public version later in 2013).

While the approach has been framed in the context of Browserbite, the underlying principles may be applied to enhance other image-based cross-browser testing techniques such as CrossCheck. Validating the technique in other settings is a direction for future work. A related direction is to evaluate the proposed technique with different types of stakeholders involved in Web application development (e.g. Web designers versus testers versus developers). In this respect, one can hypothesize that classification models for designers would be different than those for developers, for example.

Browserbite is representative of tools for single-page cross-browser compatibility testing. Prototypes and techniques such as Crosscheck [2], Webmate [11], [12] or the technique reported in [13], address the complementary problem of behavioral testing, meaning that they detect incompatibilities that arise when navigating from a given page. The integration of the techniques explored in this paper with behavioral testing techniques is another avenue for future work.

#### REFERENCES

- [1] StatCounter Gglobal Stats.. [Online]. <http://gs.statcounter.com>
- [2] S. R. Choudhary, M. R. Prasad, A. Orso, "CROSSCHECK: Combining Crawling and Differencing To Better Detect Cross-browser Incompatibilities in Web Applications," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, Montreal, Canada, 2012, pp. 171–180.
- [3] S. R. Choudhary, H. Versee, A. Orso, "WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, Timisoara, Romania, 2010, pp. 1–10.
- [4] A.-L. Tamm, "Visual testing in different testing approaches", University of Tartu, Bachelor thesis 2012.
- [5] R. L. Ebel, "Estimation of the reliability of ratings," *Psychometrica*, vol. 16, no. 4, pp. 407–424, 1951.
- [6] L. Breiman, J. Friedman, C. J. Stone, R. A. Olshen, *Classification and Regression Trees*, 1st ed.: Chapman and Hall/CRC, 1984.
- [7] G. P. Zhang, "Neural Networks for Classification: A Survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 30, no. 4, pp. 451–462, 2000.
- [8] J. Heaton, *Introduction to Neural Networks for Java*, 2nd ed.: Heaton Research, 2005.
- [9] A. Blum, *Neural Networks in C++: An Object Oriented Framework for Building Connections*, NY: John Wiley & Sons, 1992.
- [10] D. M. W. Powers, "Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness, and Correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.
- [11] V. Dallmeier, M. Burger, T. Orth, A. Zeller, "WebMate: A Tool for Testing Web 2.0 Applications," in *Proceedings of the Workshop on JavaScript Tools*, Beijing, China, 2012, pp. 11–15.
- [12] V. Dallmeier, M. Burger, T. Orth, A. Zeller, "WebMate: Generating Test Cases for Web 2.0," in *Software Quality. Increasing Value in Software and Systems Development*, S. Biffl D. Winkler, Ed.: Springer, 2013, pp. 55–69.
- [13] A. Mesbah, M. R. Prasad, "Automated Cross-Browser Compatibility Testing," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE'11*, Honolulu, HI, USA, 2011, pp. 561–570.

# Automated Classification of Static Code Analysis Alerts: A Case Study

Ulaş Yüksel\*<sup>†</sup>, Hasan Sözer<sup>†</sup>

\*Vestel Electronics, Manisa, Turkey

<sup>†</sup>Özyeğin University, İstanbul, Turkey

{ulas.yuksel, hasan.sozer}@ozu.edu.tr

**Abstract**—Static code analysis tools automatically generate alerts for potential software faults that can lead to failures. However, developers are usually exposed to a large number of alerts. Moreover, some of these alerts are subject to false positives and there is a lack of resources to inspect all the alerts manually. To address this problem, numerous approaches have been proposed for automatically ranking or classifying the alerts based on their likelihood of reporting a critical fault. One of the promising approaches is the application of machine learning techniques to classify alerts based on a set of artifact characteristics. In this work, we evaluate this approach in the context of an industrial case study to classify the alerts generated for a digital TV software. First, we created a benchmark based on this code base by manually analyzing thousands of alerts. Then, we evaluated 34 machine learning algorithms using 10 different artifact characteristics and identified characteristics that have a significant impact. We obtained promising results with respect to the precision of classification.

**Keywords**—alert classification, industrial case study, static code analysis

## I. INTRODUCTION

Static code analysis tools (SCATs) inspect the source code of programs to automatically pinpoint (potential) faults without actually executing these programs [1]. They complement manual software reviews and testing activities to assist in the development of reliable software systems. The main disadvantage of SCATs is the large amount of alerts (i.e., warnings, issues) being exposed to developers. The density of alerts can be as much as 40 alerts per thousand lines of code (KLOC) [2]. Although 2 alerts per KLOC on average is typical in our experience, even this alert density leads to an information overload. Around 3000 alerts are generated for an industrial scale software of size 1500 KLOC. Moreover, these alerts are subject to false positives. Empirical results have shown that effective SCATs have false positive rates ranging between 30% and 100% [3]. Hence, every alert should be (manually) inspected by developers to identify which alerts should be considered (i.e., true positives, alerts that are actionable [2]) and which should not. This process requires considerable time and effort such that the cost of using SCATs overcomes its benefits. If we assume the inspection time to be 5 min. per alert [2], [4], a developer might need up to 250 hours of time to inspect 3000 alerts.

We have observed that the inspection of SCAT alerts constitute a significant amount of time and effort during the software development lifecycle. The problem grows as we recognize a saturation effect of high number of alerts

when the software evolves. High ratio of false positives also makes the use of SCATs discouraging. SCATs often provide a prioritization (i.e., severity measure) for the reported alerts. Sometimes this information is consulted to focus on the highly prioritized alerts only. The majority of the alerts might be ignored this way. Although this approach can save some time and effort, one can miss the opportunity to fix some important defects in early development stages.

To address this problem, the output of SCATs can be (automatically) post-processed to classify alerts and eliminate false-positives. Numerous approaches have been proposed for automatically classifying/ranking the alerts of SCATs based on their likelihood of reporting a critical fault [5]–[10]. One of the promising approaches is the application of machine learning techniques to classify alerts based on a set of artifact characteristics [6]. In this work, we evaluate this approach in the context of an industrial case study to classify the alerts generated for a digital TV software. The system comprises millions of lines of code that has evolved for more than a decade. We created a benchmark based on this code base by manually analyzing thousands of alerts. The benchmark is used for 3 different studies. First, 10 different artifact characteristics are evaluated for identifying the most relevant attributes for classification. Second, 34 different machine learning algorithms are evaluated with respect to the accuracy, precision and recall measures. Third, several different test sets are used for evaluation, each of which reflects a different point of time (i.e., snapshot) during the software development life cycle. We obtained promising results with respect to the precision of classification, which can range from 80% up to 90%.

The remainder of this paper is organized as follows. The following section presents the industrial case. In Section III, we describe the data set and characteristics we analyzed for alert classification. Section IV discusses the case studies and results. Related studies are summarized in Section V. Finally, conclusions are provided in Section VI.

## II. INDUSTRIAL CASE: DIGITAL TV

In this section, we introduce an industrial case for classifying alerts generated for a Digital TV software system. This is an embedded soft real-time system and it has been maintained by Vestel Electronics<sup>1</sup>, which is one of the largest TV manufacturers in Europe. The software was first developed for digital video broadcasting (DVB) set-top boxes by Cabot Communications<sup>2</sup> founded in 1993. In 2001, the company

<sup>1</sup><http://www.vestel.com.tr>.

<sup>2</sup><http://www.cabot.co.uk>.

was acquired by Vestel Electronics and the software has been extended to support both digital and analog TV applications involving many different features such as video recording, media browsing and web browsing. During this time frame, hundreds of software developers have performed tens of thousands of changes resulting in a software system with over 1500 KLOC distributed in over 30000 classes/types and over 8000 files.

The software is composed of three layers: *i) application*, *ii) middleware* and *iii) driver*. The *application layer* controls the behaviour of the middleware layer. It implements an on-screen menu system and manages user interaction through remote control, front panel keys and other control interfaces.

The *middleware layer* implements the main features including service installation, date/time handling, audio and video playing, teletext/subtitle display and software updates. Both the middleware and the application layers have an object-oriented design, and they are mainly implemented in the C++ language.

Middleware is ported to different hardware platforms using a uniform interface provided by the *driver layer*. This is an interface that is implemented in the C language and it provides access to operating system primitives as well as hardware device drivers. The driver layer comprises external libraries that are developed by hardware suppliers. Hence, in our case study, we focused on the application and middleware layers only. We have the complete source code and version history for these layers. In the following section, we summarize the artifact characteristics and the data set utilized for analyzing this code base.

### III. ANALYZED ARTIFACT CHARACTERISTICS AND THE DATA SET

In Vestel, a dedicated bot server regularly checks out the project source code from an Subversion (SVN) server and runs a commercial SCAT<sup>3</sup>. For this study, we took the first 19 weekly run of SCAT over the code base in SVN. SCAT generates a list of alerts with a unique *ID* and a set of characteristics such as *severity*, *alert code*, *alert state* (*open* or *fixed*), *file name*, *method name*, and *line number*.

Software developers can inspect the generated alerts and provide feedback about them. We call this feedback the *developer idea* and we consider it as a characteristic for classifying SCAT alerts. We have selected in total 10 different characteristics to classify alerts. These characteristics are mainly pointed out in previous similar studies [2], [7], [9] although the *developer idea* has not taken much attention in the literature except only a few studies [5]. The analyzed characteristics are listed in the following.

**Severity:** An ordinal number between 1 and 4, which is assigned by SCAT for indicating the importance of the problem (1 is the highest, 4 is the lowest).

**Alert code:** An abbreviation assigned by SCAT, which indicates the type of alert. e.g., *NPD* stands for *Null Pointer Dereferencing*.

**Lifetime:** Number of periodic SCAT runs between the first discovery and the closure of an alert, if the alert is closed;

number of SCAT runs between the first discovery and the current time, if the alert is still open.

**Developer idea:** One of the four different values: *i) fix*, *ii) not a problem*, *iii) ignore*, and *iv) analyse*. The default value is *analyse*. It becomes *fix* if the developer conforms and fixes the issue reported by the alert. At the next run of SCAT, such alerts are closed automatically if the issue is really resolved. If the developer thinks that the alert is a false positive, the *developer idea* is set as *not a problem*. If the alert is not a false positive, yet the reported issue is not considered important/relevant, the *developer idea* is set as *ignore*. An alert can also be fixed and closed implicitly during the development or a bug fix process without any alert inspection. In that case, the *developer idea* keeps the value *analyse* but the *alert state* changes to be *fix* silently at the next run of SCAT.

**File name:** Source file name.

**Module name:** Root folder name.

**Open alerts:** Number of open alerts, if any.

**Total alerts:** Number of alerts, if any.

**Total alerts in module:** Number of alerts in a particular module, if any.

**Total alerts in file:** Number of alerts in a particular file, if any.

After collecting data regarding the characteristics listed above, we have manually inspected each alert associated with the source code and determined if the alerts generated by SCAT is actionable or unactionable (i.e., false positive). As such, we created a full oracle that we use in our analysis with 1147 total alerts (498 actionable and 649 unactionable). Our inspection results are listed in Table I.

TABLE I. MAPPING OF ALERT INSPECTION AND RESOLUTION.

		Manual Inspection Result		
		True Positive	False Positive	Total
Resolution	Resolved	203	62	265
	Not resolved	295	587	882
Total		498	649	1147

During the 19 weekly period of SCAT runs, some alerts were evaluated and resolved by developers independent from our study. 265 alerts are resolved in which 62 of them are false positives according to our manual inception. We observed that developers mainly tend to consider the *severity* measure provided by SCAT during the selection of alerts to be resolved. Almost 80% of the resolved alerts have the highest *severity* value. On the other hand, 882 alerts have remained unresolved, where 295 of them are true positives according to our inspection.

### IV. THE CASE STUDY AND RESULTS

In this section, we first summarize the 3 different studies we performed. Then we present our analysis in detail and discuss the results.

In the first study, we evaluated the relevance of the 10 selected characteristics as attributes for alert classification. For

<sup>3</sup>We keep the name of the tool confidential.

this purpose, we used 10 different attribute evaluator tools from the Weka (Waikato Environment for Knowledge Analysis) toolset [11]. Weka is an open source toolset, which supports common data mining features such as classification, clustering, regression, association, and attribute selection based on various machine learning algorithm implementations. In the second study, we used the full data set (i.e., alerts generated throughout the 19 runs of SCAT) and 10-folds cross-validation [11] to evaluate the accuracy of different machine learning algorithms. We tested in total 34 different machine learning algorithms implemented in Weka. In the third study, we used a training set based on the alerts generated until the end of the 5<sup>th</sup> run of SCAT only. Then, we classified alerts generated in later phases of the project. In the following, we explain our studies and the corresponding analysis in detail, together with our results.

*a) Attribute Evaluation:* In this study, we aimed to see which of the 10 selected characteristics are evaluated to be relevant for alert classification. We collected all the alerts generated for our case. We used 10 different attribute selection evaluator tools of Weka. 7 of these tools provide a merit and ranking for each of the evaluated attributes. The other 3 just perform a binary evaluation and select a sub-set of the attributes. For these 3 tools, we assigned a merit to each attribute based on the cardinality of the sub-sets. (e.g., if only 4 characteristics are selected out of 10, we assigned value 2.5 to the selected and 7.5 to the unselected ones.) Then, we calculated the mean value of merits obtained/derived from the 10 different tools for each characteristic. Figure 1 presents the results both in terms of the mean ranking value and the number of times a characteristic appears in top four ranks. Hereby, *file name*, *lifetime*, *alert code*, *developer idea* and *severity* appear to be the most relevant characteristics for classification.

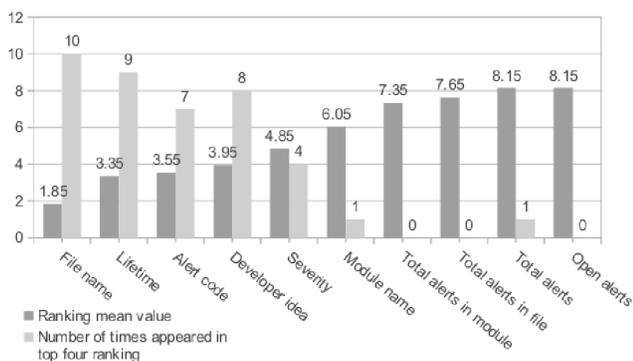


Fig. 1. Artifact characteristics ranking results (ordered).

*b) Classification with 10-folds cross validation:* In the second study, we evaluated the effectiveness of classification by applying 10-folds cross validation [6], [12] over the whole set of the generated alerts. We tested 34 different machine learning algorithms - with default settings - implemented in Weka. We have observed that the accuracy ranges between 45% and 86%. Table II lists the classifiers having the best accuracy values based on the initially selected 10 characteristics.

*c) Classification during the development life-cycle:* 10-folds cross-validation uses the full data set for both the training set and the test set. As a result, the test set includes exact values for the *lifetime* and the *developer idea* characteristics.

TABLE II. TOP CLASSIFIERS WITH RESPECT TO ACCURACY.

Classifier	Accuracy	Precision <sup>a</sup>	Recall <sup>a</sup>
Random forest [13]	86.1%	86.1% [0.84,0.88]	86.1% [0.84,0.88]
Random committee	86.4%	86.4% [0.85,0.88]	86.4% [0.84,0.88]
DTNB [14]	83.6%	83.8% [0.80,0.87]	83.6% [0.84,0.84]

<sup>a</sup> Values in brackets are particular calculations for true positive and false positive classifications respectively.

In fact, these values would not be available for alerts that are just generated by the latest SCAT run. Hence, in our third study, we took *snapshots* at certain SCAT runs. We prepared incremental test sets at these snapshots to mimic a real case during the development life cycle. We took the alerts that are generated during the first 5 SCAT runs. The training set that is prepared based on these alerts represents 92.94% of the full data set. This is due to fact that a legacy software has been used, for which the first run of SCAT already generated a high number of alerts. If an alert in the training set was resolved after the 5<sup>th</sup> run of SCAT, then we recalculated the *lifetime* value instead of using the recorded *lifetime*. In addition, we reset the *developer idea* as *analyze*, because at that point in time, this alert had not been investigated and resolved yet. Then, we created 3 test sets from the remaining alerts. These sets can be seen in Table III, where the alerts are grouped according to *SCAT run number* for the SCAT run they first appear (e.g., SCAT run # [6,10]) We reset all the *developer idea* and *lifetime* characteristics in these test sets too, as if they were new and had not been considered yet.

TABLE III. TEST SETS REFLECTING DIFFERENT POINTS OF TIME DURING THE DEVELOPMENT LIFE CYCLE.

Group	SCAT run #	True Positive	False Positive	Total
1 <sup>st</sup>	[6,10]	2	13	15
2 <sup>nd</sup>	[11,16]	9	11	20
3 <sup>rd</sup>	[17,19]	34	12	46

We trained the 34 machine learning algorithms with our training set (based on SCAT run # [1,5]) and then evaluate them with respect to the 3 test sets (Table III). Table IV presents the results for one of the classifiers (DTNB), which was one of the most accurate classifiers according our second study (Table II). We can observe a significant improvement from group 1 to group 2, and from group 2 to group 3, as well. This might be explained by the increasing number of true positives involved in the test sets (See Table III), while the number of false positives are almost equal. In general, the results turned out to be promising; the average accuracy, precision and recall is around 90% for the third group.

TABLE IV. DTNB CLASSIFICATION RESULTS FOR DIFFERENT POINTS OF TIME DURING THE DEVELOPMENT LIFE CYCLE.

Group	Accuracy	Precision <sup>a</sup>	Recall <sup>a</sup>
1 <sup>st</sup>	66.7%	80.7% [0.20,0.90]	66.7% [0.5,0.69]
2 <sup>nd</sup>	80.0%	86.2% [0.69,1.00]	80.0% [1.00,0.64]
3 <sup>rd</sup>	89.1%	90.7% [0.97,0.73]	89.1% [0.88,0.92]

<sup>a</sup> Values in brackets are particular calculations for true positive and false positive classifications respectively.

## V. RELATED WORK

There exist several case studies performed to reveal the capabilities and limitations of SCATs. Zheng et al. collect experiences from the development of a set of industrial software products. In their study, SCAT alerts are compared with respect to manual inspection results and faults found during tests. They conclude that SCATs constitute an economical value, being complementary to other verification and validation techniques [15]. Boogerd et al. [16] present an industrial case study for identifying correlations between faults and violations detected regarding the coding standard rules. They conclude that true positive rates may highly differ from project to project.

Some other studies focus on improving the usefulness of SCATs by ranking or classifying alerts [2]. An adaptive ranking model [5] is proposed for adjusting the ranking factor of each alert type based on its statistical contribution to the previous rankings and the feedback of developers for the previously ranked alerts. A benchmark [17] is introduced based on a set of Java programs for comparing classifiers with respect to their effectiveness. In particular, 3 Java programs are used for testing different machine learning algorithms with a wide range of artifact characteristics [6] to identify the most effective subset(s) of characteristics for alert classification. Kim and Ernst propose an approach for exploiting “historical information” to prioritize alerts [7]. Hereby, they mine the software change history for alerts that are removed during bug fixes. Alert categories are prioritized based on fixes applied on the corresponding alerts. Liang et al. also employ bug fix history for alert classification [9] and differentiate bug fixes as either “project-specific” or “generic”. They conclude that one may improve precision by using history regarding the “generic” bug fixes only.

Previous studies make use of a single set of alerts over the whole project life time. In this work, we took several *snapshots* from the alert history and created different training/test sets, in which artifact characteristics (e.g., *lifetime*) reflect the corresponding point of time during the software development life cycle. As another difference, previous related studies mostly focus on Java programs, whereas we work on embedded software and a C/C++ code base. To the best of our knowledge, the use of machine learning techniques for classifying SCAT alerts have not been evaluated in the context of industrial case studies for this application domain before.

## VI. CONCLUSION AND FUTURE WORK

In this work, we evaluated the application of machine learning techniques to automatically classify SCAT alerts. We created a data set by manually inspecting thousands of alerts generated for a digital TV software. Then, we performed 3 different studies for *i*) ranking the relevance of 10 different characteristics by 10 different attribute evaluators, *ii*) evaluating 34 different classifiers with 10-folds cross validation, and *iii*) evaluating the effectiveness of classification at different points of time during the software development life-cycle. The characteristics *lifetime*, *developer idea*, *file name*, *alert code* and *severity* were identified as the most relevant attributes. We obtained promising results with respect to the precision of classification, which can go up to 90%. Hence, we conclude that the use of machine learning techniques can be a viable approach for automated classification of SCAT alerts.

In this study, we created a benchmark by using the output of the first 19 weekly runs of the SCAT. In our future work, we plan to extend our data set with the consequent SCAT runs of the same software project. We also plan to incorporate additional artifact characteristics based on our observations. For instance, the *developer idea* is a subjective attribute that can take very different values based on the insights, experience, sensitivities and ambitions of different developers. Therefore, the *developer idea* can be combined with the *developer name* to increase the precision of classification.

## REFERENCES

- [1] R. Tischer, R. Schaufler, and C. Payne, “Static analysis of programs as an aid for debugging,” *SIGPLAN Notices*, vol. 18, no. 8, pp. 155–158, 1983.
- [2] S. Heckman and L. Williams, “A systematic literature review of actionable alert identification techniques for automated static code analysis,” *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.
- [3] T. Kremenek and D. Engler, “Z-ranking: using statistical analysis to counter the impact of static analysis approximations,” in *Proceedings of the 10th international conference on Static analysis*, 2003, pp. 295–315.
- [4] “Effective management of static analysis vulnerabilities and defects,” White Paper, Coverity Inc., 2009.
- [5] S. S. Heckman, “Adaptively ranking alerts generated from automated static analysis,” *Crossroads*, vol. 14, no. 1, p. 7:17:11, Dec. 2007.
- [6] —, “A systematic model building process for predicting actionable static analysis alerts,” Ph.D., North Carolina State University, United States – North Carolina, 2009.
- [7] S. Kim and M. D. Ernst, “Prioritizing warning categories by analyzing software history,” in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR ’07. Washington, DC, USA: IEEE Computer Society, 2007, p. 27.
- [8] —, “Which warnings should I fix first?” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, p. 4554.
- [9] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, “Automatic construction of an effective training set for prioritizing static analysis warnings,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE ’10. New York, NY, USA: ACM, 2010, p. 93102.
- [10] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, “A framework to compare alert ranking algorithms,” in *2012 19th Working Conference on Reverse Engineering (WCRE)*, Oct. 2012, pp. 277–285.
- [11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The WEKA data mining software: An update,” *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.
- [12] K. Yi, H. Choi, J. Kim, and Y. Kim, “An empirical study on classification methods for alarms from a bug-finding static C analyzer,” *Information Processing Letters*, vol. 102, no. 23, pp. 118–123, 2007.
- [13] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, 3rd ed. Morgan Kaufmann, Jan. 2011.
- [14] M. A. Hall and E. Frank, “Combining naive bayes and decision tables,” Florida, USA, 2008, pp. 318–319.
- [15] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, “On the value of static analysis for fault detection in software,” *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, Apr. 2006.
- [16] C. Boogerd and L. Moonen, “Assessing the value of coding standards: An empirical study,” in *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, Oct. 2008, pp. 277–286.
- [17] S. Heckman and L. Williams, “On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM ’08. New York, NY, USA: ACM, 2008, p. 4150.

# Mining Telecom System Logs to Facilitate Debugging Tasks

Alf Larsson  
 PLF System management  
 Ericsson, Research & Development  
 Stockholm, Sweden  
 Alf.Larsson@ericsson.com

Abdelwahab Hamou-Lhadj  
 SBA Research Lab  
 ECE, Concordia University  
 Montreal, Canada  
 abdelw@ece.concordia.ca

**Abstract.** Telecommunication systems are monitored continuously to ensure quality and continuity of service. When an error or an abnormal behaviour occurs, software engineers resort to the analysis of the generated logs for troubleshooting. The problem is that, even for a small system, the log data generated after running the system for a period of time can be considerably large. There is a need to automatically mine important information from this data. There exist studies that aim to do just that, but their focus has been mainly on software applications, paying little attention to network information used by telecom systems. In this paper, we show how data mining techniques, more particularly the ones based on mining frequent itemsets, can be used to extract patterns that characterize the main behaviour of the traced scenarios. We show the effectiveness of our approach through a representative study conducted in an industrial setting.

**Keywords—** System logs, event correlation, troubleshooting of telecom systems, mining algorithms.

## I. INTRODUCTION

Ericsson is one of the largest telecom companies in the world. It has a much diversified product portfolio comprising of various network components. These components work together and are usually distributed in nature. When errors or abnormal behaviours occur, software engineers turn to the analysis of logs, generated by monitoring and tracing the system's activities. Logs, however, tend to be overwhelmingly large, which hinders any viable analysis unless adequate (and practical) tool support is provided [5, 6].

Log analysis is a broad topic and varies in scope depending on the application domain. In this paper, we focus on the problem of extracting meaningful patterns from system logs to help software engineers understand the main behaviour of the traced scenario. The ultimate goal is to facilitate debugging and other maintenance tasks. Consider, for example, the simple scenario of transferring a file over FTP (File Transfer Protocol) between two network sites. The generated log file is bound to noise and network interferences (as it is almost always the case in industrial systems). Knowing which events are most relevant to the file transfer itself is a challenging task. But when performed properly, it can reduce significantly the time and effort it takes to software engineers to understand and troubleshoot the system in case the transfer fails.

At Ericsson, a common approach is to look at the occurrence of events and relate them using timestamp information. Due to noise and interference in the data, this type of analysis has limited ability to uncover correct and complete behaviour. Hence, the process often requires heavy involvement of domain experts.

In this paper, we investigate the use of data mining techniques for identifying and analyzing important events and patterns in large system logs with minimum intervention of the users.

## II. APPROACH

Our approach is shown in Figure 1. It encompasses two main phases: Pattern Generation and Validation, and Pattern Matching. The first phase is a learning phase in which we apply data mining techniques to extract behavioural patterns from large logs. The extracted patterns are presented to domain experts for validation. Domain experts can choose to assign a context (a description and any other relevant information) to the valid patterns. The patterns with their description are then saved in a database.

During the pattern matching phase (the second step), we use the pattern database to correlate events in a random set of logs generated from systems in operation using pattern matching techniques. We also support the possibility to correlate these patterns. This is particularly useful if the traced feature involves several scenarios. Software engineers can see how these scenarios are interrelated. These phases are explained in more details in the subsequent sections, preceded with a subsection on log generation.

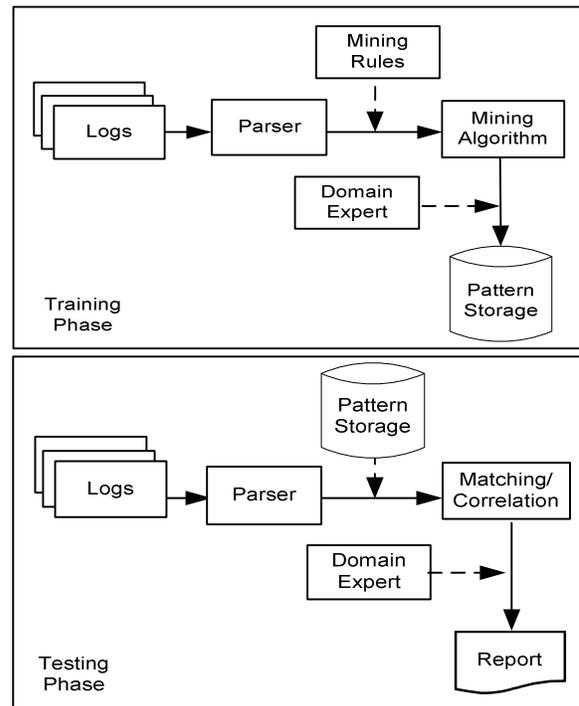


Figure 1. Overview of the approach

### A. Collection of System Logs

We generate logs by exercising the system with a usage scenario of interest. Our strategy is to run the scenario several times with different background noise and feed the resulting log files to a data mining algorithm to automatically extract the

common sequences of events. Our hypothesis is that the events that are common to the generated log files are the ones that are also the most relevant. This approach is similar to Software Reconnaissance introduced by Wilde et al. in [12] and further improved by many researchers (see [4] for a survey). The authors compared traces generated from routine call traces to identify the components that implement a particular scenario. This contributes to solving a problem known as feature location in code; identifying the most relevant components that implement a specific feature.

The main difference is that we use a data mining algorithm instead of correlating traces using graph theory. Also, to our knowledge, software reconnaissance and other feature location techniques have not been applied to network logs. This type of run-time information differs greatly from traces of control flow. Network logs tend to be more fine-grained and the information cannot be easily mapped to the source code. Many feature locations techniques heavily rely on the source code to find relevant information.

### B. Learning Phase: Pattern Generation

There exist several data mining algorithms that extract patterns from large data. Examples include Apriori [1], Frequent Pattern tree (FP-tree) [2], FP-Growth [10, 11], etc. The one we chose to use in this paper is MAFIA [3]. MAFIA stands for Mining Maximal Frequent Itemsets Algorithm. We selected MAFIA because of its time efficiency for extracting long patterns compared to its counterparts [3].

MAFIA starts by building a lattice tree that represents the lexicographic ordering of the items in an itemset. We will detail this process in the subsequent paragraphs. The algorithm then applies a depth-first search algorithm with pruning techniques to detect maximal frequent itemsets that have a support greater or equal than a certain threshold. The support of an itemset represents the number of times it appears in the itemset database. To illustrate how MAFIA works, let us revisit the example provided in [3]. In this example,  $I$ , the item set, contains four items  $I = \{1, 2, 3, 4\}$ . A database of itemsets,  $T$ , is a multiset of subsets of  $I$ . The objective of the algorithm is to find the maximal frequent itemsets in  $T$ . For example, the result of applying MAFIA to  $T = \{\{1234\}, \{123\}, \{134\}, \{234\}\}$  is  $\{123\}$  (with minimum support = 2).

The algorithm starts by building a lattice in such a way that the top itemset is the empty set and each lower level  $k$  contains all itemsets of length  $k$ . The itemsets are ordered according to the lexicographic ordering relationship. The lexicographic subset lattice generated from  $I$  is shown in Figure 2. The first level ( $k = 0$ ) is the empty set. The next level contains itemsets of length 1, ordered in the lexicographic way  $\{1\} < \{2\}$ , etc. The next level (where the effect of ordering is more noticeable) contains items of length 2. In this level, Itemsets  $\{12\}$ ,  $\{13\}$ ,  $\{14\}$ , generated from extending  $\{1\}$ , are also ordered in the lexicographic manner, etc.

One simple way to find maximal frequent itemsets is to apply a naïve depth-first algorithm and count the number of occurrences of each itemset. An itemset with a support greater or equal to the minimum support is added to the maximal itemset database (the output of the algorithm), given that a superset has not already been discovered. The problem with a simple depth-first algorithm is that it tends to be unnecessarily slow. This is because it counts the frequency of all itemsets in the tree despite the fact that some subtrees can be quickly

filtered out earlier in the process if a different reordering is used. For example, given the subtree rooted at  $P = \{1\}$ , counting the support of  $\{12\}$ ,  $\{13\}$ ,  $\{14\}$ ,  $P$ 's children, first will reveal that only  $\{12\}$  is frequent (it appears twice in the database). Items 3 and 4 can then be filtered out so no new itemsets need to be counted in the subtree rooted at  $P$ . In other words,  $\{123\}$ ,  $\{124\}$ , and  $\{234\}$  do not need to be counted which will save time. Based on this, the MAFIA authors [3] present various pruning and reordering algorithms to increase the performance of the algorithm by reducing the search space. An implementation of MAFIA is made publicly available by the authors on <http://himalaya-tools.sourceforge.net/Mafia/>.

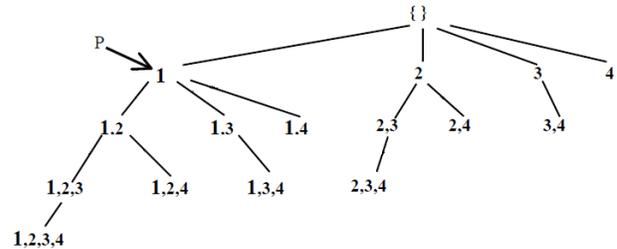


Figure 2. Lexicographic ordering lattice (from [3])

To apply MAFIA to log events, we introduce the following definitions. We call an instance of a given scenario a ‘window’. We distinguish each instance with a unique window identifier (window\_id). We save the generated logs in a database table. Each column consists of a specific attribute of a log event. We assign a unique integer transaction id to each distinct attribute. The idea is to use the ids for lexicographic ordering. While assigning transaction ids, window limit is not taken into consideration. Even if a log entry is found across multiple windows, it is assigned the same transaction id. Figure 3 shows an example of two scenarios, represented as windows, with their events (depicted using A, B, C...). The event attributes are assigned transactions ids. In this example,  $I = \{1, 2, 3, 4, 5, 6, 7\}$  and  $T = \{\{121234\}, \{1251267\}\}$ . The result of the algorithm (with minimum support = 2), is Itemset  $\{12\}$  which corresponds to the pattern AB.

Window ID	Window 1	Window 2
Sample logs for each window	A B A B C D	A B E A B D F G
Transaction ID	1 2 1 2 3 4	1 2 5 1 2 6 7

Figure 3. Example of scenarios represented as windows

We want to note that, in order to obtain a pattern representing a specific behaviour, we do not consider all attributes of a log event during the mining process. For example, a typical log event generated from file transfer operation will involve the timestamp, protocol, sender’s IP, receiver’s IP, and other information. Considering the timestamp will end up eliminating the very possibility of obtaining a pattern by making each event unique because timestamp varies from one event to another. The decision on which attributes to select is left to the user. It is always recommended to use attributes that represent a generalized behaviour. The more attributes we use, the more restrictive the pattern detection approach is.

### C. Pattern Validation and Context Assignment

Once we extract the patterns, we present them to domain experts at Ericsson for validation. This is usually done in a semi-automatic way using a tool we have developed for this

research (see the case study for snapshots). A typical task of the domain expert is to go through the pattern, assess its quality, and remove unnecessary data if need be. There might be situations where the domain expert considers the quality of the pattern to be poor (e.g., it lacks key events). In this case, he or she can request either to re-run the pattern mining process by adjusting its input (adding other attributes) or run the scenario again with additional background noise to clearly distinguish the behavioural pattern events from other events.

Once the domain expert deems the pattern to be valid, he or she assigns a context, which is a high-level description comprised of the pattern name, the context in which the pattern appeared (e.g., the network topology, the type of communication used, the communication protocol, etc.). The pattern is then saved in the pattern library.

#### D. Testing Phase: Pattern Matching

We have developed a simple pattern matching algorithm to identify the event patterns in logs generated from a system in operation. The algorithm simply matches the log events to the patterns in the database. Our matching algorithm operates as follows: Given a sequence of events  $s_1$  and a pattern  $p_1$  (in the pattern database),  $s_1$  and  $p_1$  are considered similar if all events in  $p_1$  appear in  $s_1$ . In other words, it is enough for  $s_1$  to contain the events that appear in a pattern to be considered as a candidate pattern. An alternative solution will be to consider exact matching but this would turn out to be too restrictive because of noise in the data. Future studies should focus on measuring similarity based on a certain threshold.

#### E. Testing Phase: Pattern Correlation

We define pattern correlation as the process of identifying a relation between the extracted patterns in a given scenario or a set of scenarios. This task is important for debugging and performance analysis since it can help software engineers identify what is happening in the system when multiple operations occur (e.g., sending an FTP file while at the same time using HTTP).

Patterns are correlated using two methods: attribute-based correlation and time-based correlation. In the attribute-based correlation method, the event attributes are used to find relation among patterns. For example, if a user wants to know which events happened in two patterns on a particular IP address, an attribute-based filtering mechanism can be employed to identify those events. The resulting output will contain only those logs from both patterns which belong to the selected IP. The time-based correlation (the second method) allows users to see which patterns appear within a particular timeframe.

### III. EVALUATION

#### A. Target System

We chose CPP (Connectivity Packet Platform) as the target system, which is a proprietary carrier-class technology developed by Ericsson [9]. It has been positioned for access and transport products in mobile and fixed networks. Typical applications on current versions of CPP include third-generation nodes RBSs (Radio Base Stations), RNCs (Radio Network Controllers), media gateways, and packet-data service nodes/home agents. CPP was first developed for ATM (Asynchronous Transfer Mode) and TDM (Time Division Multiplexing) transport.

A CPP node contains two parts, an application part and a platform part. The application part handles the software and

application-specific hardware. The platform part handles common functions such as internal communication, supervision, synchronization, and processor structure.

#### B. Usage Scenarios and Log Generation

We experimented with various scenarios. Due to the proprietary nature of the system, we choose, in this paper, to present two scenarios: inter-frequency handover (IFHO) and the setup of the radio access bearer (RAB). The results are representative of our findings. Each of these scenarios was performed across various RBSs, where each RBS was working on a set of cells available giving a wide variety of logs.

The log generation tool we used in this paper is the Trace and Error (T & E) package, which is a built-in capability in CPP, used often by software engineers to integrate, verify, and troubleshoot CPP applications [9]. T&E supports two functionalities: the tracing functionality and the error handling functionality. The tracing functionality helps the system and functional behaviors to be traced and reported at software development. The error functionality helps to log fault conditions. The T & E log shows a history of recorded trace and error events on the system.

#### C. Learning Phase: Pattern Generation and Validation

##### Scenario 1: IFHO

To generate pattern for IFHO, we run the scenario several times with different background noise. We generated a log file for each run. The size of log files varies from 3 to 7 GB. An IFHO event has many attributes including the event ID, DeviceFrom, DeviceTo, LoadModule, Message, MessageType, MessageText, and Parameters. We fed the log files to the pattern generation component of our approach. We selected the attributes “MessageType” and “MessageText” as the main attributes for the pattern generation process.

serial...	timeStamp	deviceFrom	deviceTo	loadModu...	message	load...	message...	messageText
1	17:47:50	UE	RNC1	1600	RncLmUe...	35	(RRC)	measurementReport
2	17:47:50	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
3	17:47:50	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
4	17:47:50	RNC1	UE	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
5	17:47:50	UE	RNC1	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
7	17:47:50	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
8	17:47:53	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkSetupRequest...
9	17:47:53	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkSetupRespons...
10	17:47:53	RNC1	UE	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
11	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkRestoreIndicati...
12	17:47:54	UE	RNC1	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
19	17:47:54	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	DedicatedMeasurement...
20	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	DedicatedMeasurement...
21	17:47:54	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkDeletionRequest...
22	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkDeletionRespo...

Figure 4. The IFHO extracted pattern

A domain expert at Ericsson analyzed the resulting pattern and removed some events including repeated signals and heartbeat type messages. These events are considered as noise and can occur at any instant. It took around one hour for the domain expert to clean up the automatically extracted pattern. We believe that this step could be automated (at least at a certain extent) in the future by studying what constitute noise in such systems and build a predefined list of events that can be removed before applying the mining algorithm. We do not expect, however, to completely discard the domain expert from the process. In fact, we believe that domain expert feedback is very useful during the whole process. The final pattern for IFHO consists of 15 events as shown in Figure 4 (note that we do not show some of the event attributes to save space). The

pattern is then saved in the pattern database under the name IFHO pattern.

#### Scenario 2: Radio Access Bearer (RAB) Setup

We followed the same process as for the previous scenario. We run RAB several times with various background noises. The size of the log files varies from 4 GB to 7 GB. The pattern mining algorithm generated a pattern. We gave this pattern to a domain expert who (as before) removed additional data (considered as noise). The resulting pattern contains 31 events and it is partially shown in Figure 5. The pattern is then saved under the contextual name: RAB set-up.

Pattern to be Stored						
amp	deviceFrom	deviceTo	loadMo...	message	load...	message...
41	RNC1	RBS	1400	FncLmUe...	1771	(NEAF) RadioLinkSetupRequestFDD
41	RBS	RNC1	1400	FncLmUe...	1771	(NEAF) RadioLinkSetupResponseFDD
41	RNC1	UE	1400	FncLmUe...	1771	(RRC) rrcConnectionSetup
41	RBS	RNC1	1400	FncLmUe...	1771	(NEAF) RadioLinkRestoreIndication
41	UE	RNC1	1400	FncLmUe...	1771	(RRC) rrcConnectionSetupComplete
41	UE	RNC1	1400	FncLmUe...	1771	(RRC) InitialDirectTransfer
41	RNC1	CN	1400	FncLmUe...	1771	(RANAP) InitialUE-Message
41	CN	RNC1	1400	FncLmUe...	1771	(RANAP) CommonID
41	CN	RNC1	1400	FncLmUe...	1771	(RANAP) SecurityModeCommand
41	UE	RNC1	1400	FncLmUe...	1771	(RRC) securityModeComplete
41	RNC1	CN	1400	FncLmUe...	1771	(RANAP) SecurityModeComplete
42	CN	RNC1	1400	FncLmUe...	1771	(RANAP) RAB-AssignmentRequest
42	RNC1	RBS	1400	FncLmUe...	1771	(NEAF) RadioLinkReconfigurationPrepareFDD
42	RBS	RNC1	1400	FncLmUe...	1771	(NEAF) RadioLinkReconfigurationReady
42	RNC1	UE	1400	FncLmUe...	1771	(RRC) radioBearerSetup
42	RNC1	RBS	1400	FncLmUe...	1771	(NEAF) RadioLinkReconfigurationCommit

Figure 5. The RAB set-up pattern

#### D. Testing Phase: Pattern Identification and Correlation

Once we identified the patterns, we used them to find patterns in the system during operation. For this purpose, we started by correlating patterns based on their attributes. To do that, we needed a log file which had a combination of scenarios. We chose a scenario that combines a set of different telecommunication sub-scenarios including IFHO, soft handover, softer handover, RAB set-up, etc. We run the scenario as it would be in real world (i.e. with background noise). The generated log file was fed to the pattern matching and correlation component of the framework. We were able to automatically identify the IFHP and RAB patterns using the pattern database built during the learning process.

Once we had the pattern highlighted, we were able to use attribute and time based correlation techniques to gain insight into what is happening in the scenario. For example, we were able to identify the most frequent destination site for IFHO messages. We conducted similar experiments using time correlation by identifying the patterns that occur within a specific timeframe. We needed for this task to do some preprocessing steps to align the time generated from parallel systems. The correlation, in this case, showed all complete patterns obtained for IFHO and RAB between these time intervals. This was helpful for following the flow of messages exchanged between different network sites.

We have shown the results to Ericsson software engineers working on troubleshooting tasks. The feedback we received shows that the approach holds real promise in simplifying the analysis of telecommunication logs, and reducing the time and effort spent on understanding their content.

## IV. CONCLUSIONS

In this paper, we demonstrated the potential of using data mining techniques, more particularly the MAFIA approach, to extract useful information from telecom logs. The objective is to help software engineers analyze these logs more efficiently

and precisely. Based on the feedback we received from software engineers at Ericsson, the approach is helpful and promising. In particular, they report that this technique (1) reduces the manual effort put into identifying relevant events required for debugging, and (2) increases the precision of relevant event identification. As future work, we intend to continue exploring the application of data mining approaches to alternative types of log analysis. We will also investigate what constitute noise in this type of data to further reduce the time spent by domain experts to identify patterns. Some techniques that can be useful to explore in this context are the ones presented in [7], in which the authors discuss the impact of utilities (noise) on the size of traces.

**Acknowledgment:** Thank you to software engineers at Ericsson, Stockholm, for active participation and providing continuous feedback.

## V. REFERENCES

- [1]. R. C. Agrawal, T. Imielinski, and R. Srikant, "Mining association rules between sets of items in large databases," *In Proc. of the ACM International conference on Management of data*, pp. 207-216, 1993.
- [2]. R. C. Agarwal, C. Aggarwal, and V.V.V. Prasad, "A tree projection algorithm for generation of frequent itemsets," *J. of Parallel and Distributed Computing*, pp. 350-371, 2001.
- [3]. D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A maximal frequent itemset algorithm for transactional databases," *In Proc. of the 17th International Conference on Data Engineering*, pp. 443 - 452, 2001.
- [4]. B. Dit, M. Revelle, M. Gethers, and D. Poshvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, 25(1), pp. 53-95, 2013.
- [5]. A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," *Ph.D. Dissertation, School of Information Technology and Engineering (SITE)*, University of Ottawa.
- [6]. A. Hamou-Lhadj, and T. Lethbridge, "Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools," *In Proc. of the 10th International Conference on Engineering of Complex Computer Systems*, pp. 559-568, 2005.
- [7]. A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities," *ECOOP International Workshop on Practical Problems of Programming in the Large, LNCS, Vol 3344, Springer-Verlag*, pp. 10-22, 2004.
- [8]. G. Jakobson, L. Lewis, and J. Buford, "An Approach to Integrated Cognitive Fusion," *In Proc. of 7th International Conference on Information Fusion*, 2004.
- [9]. L-Ö. Kling, Å. Lindholm, L. Marklund and G. B. Nilsson, "CPP. Ericsson Review No. 2", 2002. Available online: [http://www.ericsson.com/ericsson/corpinfo/publications/review/2002\\_02/files/2002023.pdf](http://www.ericsson.com/ericsson/corpinfo/publications/review/2002_02/files/2002023.pdf)
- [10]. B. S. Kumar and K.V.Rukmani, "Implementation of Web Usage Mining Using APRIORI and FP Growth Algorithms," *Journal of Advanced Net and App*, 1(6), pp. 400-404, 2010.
- [11]. Y-C. Li, C-C. Chang, "A New FP-Tree Algorithm for Mining Frequent Itemsets," *Springer Lecture Notes in Computer Science Volume 3309*, pp 266-277, 2004.
- [12]. N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, 7(1), pp.49-62, 1995.

# Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study

Dusica Marijan, Arnaud Gotlieb, Sagar Sen  
 Certus Software V&V Centre  
 Simula Research Laboratory  
 Email: dusica, arnaud, sagar@simula.no

**Abstract**—Regression testing in continuous integration environment is bounded by tight time constraints. To satisfy time constraints and achieve testing goals, test cases must be efficiently ordered in execution. Prioritization techniques are commonly used to order test cases to reflect their importance according to one or more criteria. Reduced time to test or high fault detection rate are such important criteria. In this paper, we present a case study of a test prioritization approach ROCKET (Prioritization for Continuous Regression Testing) to improve the efficiency of continuous regression testing of industrial video conferencing software. ROCKET orders test cases based on historical failure data, test execution time and domain-specific heuristics. It uses a weighted function to compute test priority. The weights are higher if tests uncover regression faults in recent iterations of software testing and reduce time to detection of faults. The results of the study show that the test cases prioritized using ROCKET (1) provide faster fault detection, and (2) increase regression fault detection rate, revealing 30% more faults for 20% of the test suite executed, comparing to manually prioritized test cases.

**Keywords**—software testing; continuous integration; regression testing; test case prioritization; history-based prioritization

## I. INTRODUCTION

One key aspect of efficient continuous integration software development is a short feedback loop from code commits to test execution reports. In a limited testing time, an important challenge to address is a trade-off between (a) selecting test cases that have high fault detection ability and (b) maximizing the number of test cases that can be executed in available time. This challenge stems from our experience in testing industrial video conferencing systems (VCS), in collaboration with our partner. The VCS is developed in a *continuous integration* environment. Changes made by developers are merged to the mainline VCS codebase on a daily basis. Testing is performed each time a new change is proposed to the VCS codebase. The software system is built and regression tested for every single change before it is committed to the mainline. Execution of a single test case for the VCS requires about 30 minutes on an average and a maximum of 45 minutes. Executing a complete set of 100 test cases for one VCS product would require at least 2 days. Therefore, we ask how can we select and prioritize a subset of test cases that can detect most of the regression faults, while being executed within a time limit? This is the question we address in this paper with the approach called Prioritization for Continuous Regression Testing (ROCKET). We demonstrate ROCKET to efficiently prioritize test cases for our industrial VCS case study.

As a major advantage in many industrial settings, ROCKET approach does not require source code. It takes four inputs: (1)

a test suite as a set of test cases, (2) a failure status and (3) execution time for each test case for several last test executions, and (4) an upper bound for the total time of testing. The prioritization algorithm computes a weight for each test case based on the distance of its failures from the current execution and the test execution time. The algorithm embeds domain-specific heuristics pertaining to a specific industrial setting. Therefore, ROCKET prioritizes the set of test cases based on test historical information and time limitations, using domain-specific heuristics.

We validate ROCKET in regression testing of commercial VCS by comparing to manual ordering performed by test engineers and random test ordering, and demonstrate that ROCKET improves effectiveness and time to detection of regression faults. In particular, the results show that test cases prioritized using ROCKET detect 30% more regression faults with 20% of the test suite executed, comparing to manually prioritized test cases. The results further show that test cases prioritized using ROCKET at the same time decrease test execution time for 40% with 20% of the test suite executed, comparing to manually prioritized test cases.

The main contributions of the paper are:

- An algorithm with tool implementation for ordering test cases in regression testing based on historical failure data, testing time constraints and domain-specific heuristics, so that the test cases with shorter time to execute and that revealed regression faults execute earlier.
- A case study to validate the effectiveness of the approach in continuous regression testing of industrial video conferencing software.

The paper is organized as follows. Section II provides background, gives motivation and reviews related work on regression testing. In Section III we describe regression test prioritization for continuous integration and our algorithm ROCKET. Section IV describes a case study of ROCKET in regression testing of industrial video conferencing software and presents the results of the study. In Section V we draw conclusion and give directions of future work.

## II. BACKGROUND AND RELATED WORK

In this section we highlight some of the challenges in continuous regression testing in industrial setting. We also review prior work on prioritization for regression testing and relate it to our industrial partner's context.

### A. Continuous Regression Testing in Practice

The following scenario illustrates the challenges imposed on a VCS test engineer performing continuous regression testing. AB is a test engineer whose task is to regression test the latest release of VCS software product, before the software will be integrated to the mainline codebase. AB is given a collection of about 100 test cases, a test execution log containing failure history information and x-hour time frame to complete the task (time frame changes from task to task). The test cases are system-level, and testing does not require source code. This task imposes two challenges on AB. First, AB has to select test cases to detect as many regression faults in software modifications as possible. Second, testing is firmly restricted to a specific duration and AB has to select and execute tests to satisfy that constraint. His objective is to select as many shorter tests as possible, in order to increase the number of executed tests and possibly the number of detected faults.

### B. Prioritization for Regression Testing

Test case prioritization has been extensively studied in literature. One class of the proposed prioritization techniques requires source code [1], [2] and such techniques were inapplicable to our needs. Sherriff presents the approach to prioritize tests by analysing effects of changes through singular value decomposition [3]. However, the approach does not consider time constraints common for regression testing in continuous integration environment. Srikanth proposes a technique that prioritizes tests based on the combination of fault detection rate over time and minimal test setup time, when switching configurations in execution [4]. This approach however does not consider time taken to execute tests. Bryce presents a metric for cost-based prioritization [5], using combinatorial interaction coverage as objective. Do presents a cost model for test prioritization and shows how different time constraints affect prioritization: time to setup testing, time to identify and repair obsolete tests, human time to inspect results [6], [7]. Our work aims at prioritization with time constraints, but only in test execution. Similarly, Walcott proposes a technique to reorder test suits in the presence of time constraints [8]. Still, this techniques considers that all tests have the same execution time. In addition to prioritization based on historical fault data and test execution cost, for our industrial partner it was important to incorporate domain-specific heuristics in the prioritization technique, due to specific industrial setting.

## III. PRIORITIZATION FOR CONTINUOUS REGRESSION TESTING

At the basic level, the goal of regression test case prioritization is to find the execution order for the given set of test cases that optimizes a given objective function. The objective function in continuous integration environment of our industrial partner is (1) selecting test cases with highest consecutive failure rate for the given number of executions, and (2) maximizing the number of executed test cases while satisfying given testing time constraints.

In this context, we define our objective function  $g$  as follows. For the given set of test cases  $S = \{S_1, S_2, \dots, S_n\}$  and the failure status for each test case in  $S$  over the last  $m$  successive executions (if the test case has been executed before),

$FS = \{\{f_{s_{1,1}}, f_{s_{2,1}}, \dots, f_{s_{m,1}}\}, \{f_{s_{1,2}}, f_{s_{2,2}}, \dots, f_{s_{m,2}}\}, \dots, \{f_{s_{1,n}}, f_{s_{2,n}}, \dots, f_{s_{m,n}}\}\}$ , we calculate the cumulative priority for each test case  $P = \{p_{S_1}, p_{S_2}, \dots, p_{S_n}\}$ . Given the  $P$  and the test case execution time  $Te = \{Te_1, Te_2, \dots, Te_n\}$ , we define the objective function as follows:

$$g = (\text{maximize}(p), \text{minimize}(Te))$$

Now, we define the problem of regression test case prioritization as finding the order of test cases from  $S$ , such that  $(\forall S_i)(i = 1..n)g(S_i) \geq (g(S_{i+1}))$ .

One idea behind our objective function is that given a short test execution time during regression testing, an effective prioritization criterion should select test cases that proved to fail in the previous test executions following chronological manner. The highest failure weight corresponds to the failure exposed in the (*current* - 1) execution and the failure in every precedent execution is weighted lower than the failure in its successive execution. While re-executing the test cases that failed in precedent intermediate execution is mandatory in regression testing, very often faults detected in the second or third last execution and corrected reoccur later, due to “quick bug fixes” or masking effects that prevent fault detection. Similarly, a successful precedent intermediate execution for a test case lowers its failure impact (i.e. priority). Another idea behind our objective function is that given a short test execution time during regression testing, in addition to selecting test cases that failed previously, an effective prioritization criterion should select test cases that execute quickly, so to increase the number of executed test cases, as every test case can potentially detect faults. If a test case takes a lot of time to execute, even if it revealed failure in the precedent intermediate execution, executing this test case will prevent other failing test cases with the same failure impact (weight) but with shorter execution time to execute. The following example justifies this prioritization objective. There are five test cases (T1 to T5) shown in Table 1, with the distribution of regression faults in five consecutive test executions (E1 to E5). For the sake of simplicity, a test case is considered to reveal the same fault across the executions. Each test case has a time unit (test execution time). One possible order of test cases that maximizes the coverage of regression faults from previous executions is  $Po1 = [T1, T4, T3, T5, T2]$ . Consider the case where the upper bound for executing tests is 7 time units. In this case only T1 will be executed, checking against only one fault. Now, if we order test cases with respect to historical fault data and execution time, e.g.  $Po2 = [T5, T3, T4, T2, T1]$ , four test cases can execute in given 7 time units, T5, T3, T2 and T4, checking against four faults. If higher priority is given to shorter test cases with the same high failure impact, the prioritization improves the fault detection effectiveness of testing.

TABLE I. TEST CASE PRIORITIZATION FOR CONTINUOUS REGRESSION TESTING.

Test case	E1	E2	E3	E4	E5	Exe time
T1	×	×			×	5
T2			×			2
T3		×			×	1
T4			×	×		3
T5	×	×				1

### A. ROCKET Algorithm

The algorithm input is fourfold: a set of  $n$  test cases to prioritize  $S = \{S_1, S_2, \dots, S_n\}$ , the execution time for each test  $Te = \{Te_1, Te_2, \dots, Te_n\}$ , a failure status for each test case in  $S$  over the last  $m$  successive executions  $FS = \{FS_1, FS_2, \dots, FS_n\}$ , where  $FS_i = \{fs_1, fs_2, \dots, fs_m\}$ , and available test execution time. The algorithm starts by assigning *zero* priority to all test cases. Next, based on the  $FS$ , failure matrix  $MF$  of size  $m \times n$  is created and filled such that

$$MF[i, j] = \begin{cases} 1, & \text{if } S_j \text{ passed in } (current - i) \text{ execution} \\ -1, & \text{if } S_j \text{ failed in } (current - i) \text{ execution} \end{cases}$$

The failure matrix is shown in Figure 1. Field values in the matrix specify for each test case from a test suite of size  $n$  whether the test case passed or failed during the last  $m$  successive executions. If the test case failed,  $MF[i, j]$  is  $-1$ , otherwise  $MF[i, j]$  is 1. Each row  $i$  in the matrix represents the distance of a test-suite execution from the current execution ( $i = 0$ ). Distances are assigned weights  $\omega_i$  that reflect the impact of the failure occurred in the test execution that is  $i$  steps far from the current test execution. The weight is assigned based on the following domain-specific heuristic:

$$\omega_i = \begin{cases} 0.7, & \text{if } i = 1 \\ 0.2, & \text{if } i = 2 \\ 0.1, & \text{if } i \geq 3 \end{cases}$$

$$MF_{m,n} = \begin{pmatrix} -1 & 1 & \dots & 1 \\ 1 & 1 & \dots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & -1 \end{pmatrix}$$

Fig. 1. Test suite failure matrix.

Given the failure matrix  $MF$  and the test-suite execution distance weighting heuristic, the algorithm calculates the cumulative priority for each test case  $PS = \{ps_1, ps_2, \dots, ps_n\}$ , as follows:

$$ps_i = \sum_{i=1..m} MF[i, j] * \omega_i$$

Next, the algorithm creates a matrix  $MP_{s \times t}$  that classifies test cases from  $S$  into  $t$  classes, based on the calculated priority values.  $s$  is the size of the largest test case class (in terms of the number of test cases). All test cases in the same class have the same priority value, which increases by 1 for the successive class. While all test cases from the same class are equally relevant for the current test-suite execution, they differ by their execution time. Given the matrix  $MP_{s \times t}$  and the test cases execution time  $Te = \{Te_1, Te_2, \dots, Te_n\}$ , the algorithm checks for each test case if its execution time exceeds overall available testing time. If it does, the test case is assigned the priority value  $t + 1$ . Otherwise, the algorithm increases the priority of a test case  $ps_i$  by its execution time value normalized to  $[0, 1]$ , with respect to the maximal test case execution time  $T_{max}$  in  $S$ :

$$ps_i = \begin{cases} t + 1, & Te_i \geq T_{max} \\ ps_i + \frac{Te_i}{T_{max}}, & otherwise \end{cases}$$

ROCKET has been implemented in a tool developed in collaboration with our industrial partner's test department. The tool is aimed for automating test selection and scheduling in continuous integration.

## IV. INDUSTRIAL CASE STUDY

In this section we present a case study to validate the effectiveness of ROCKET in continuous regression testing of industrial software. We designed the experiments to answer the following research questions:

- RQ1: Can ROCKET-prioritized test suite increase the number of test cases executed in limited testing time, compared to manually prioritized one?
- RQ2: Can ROCKET-prioritized test suite at the same time increase the number of detected regression faults when only part of the test suite has been executed, compared with manually ordered test suite?

### A. Software Studied

The studied VCS belongs to a class of configurable systems (10 products) that share many commonalities in structure and functionality and differ in features used to customize the software for specific user needs. The VCS testing team consists of 10 engineers that are responsible for testing final product releases with monthly frequency, and regression testing of feature updates at a daily base. Updates made to software features that are hardware dependent affect more products in a class and need to be tested for all affected products.

### B. Methodology

In the experiments we used the data collected during 5 consecutive test case executions for the VCS ( $E1$  to  $E5$ ). Data include: a test-suite (the same test suite was used in all 5 executions), failing test cases for each execution and execution time for each test case. We analysed the data and built a matrix that shows how are failing test cases distributed over five consecutive regression test executions. For example, the test case T1 from Table 1 failed in test execution  $E1$  and  $E2$ . Afterwards, the fault was fixed and T1 passed in execution  $E3$  and  $E4$ , but due to regressions T1 failed again in execution  $E5$ . Using the failure information from the previous 5 test executions we ordered the test cases using ROCKET. To answer RQ1, we measured how much time will take to execute partial test suites for ROCKET-prioritized and manually-prioritized suites. To answer RQ2, we measured regression fault detection capability, in terms of the number of faults detected by partial test suits, for both prioritization approaches.

Additionally, we compared prioritization effectiveness of ROCKET with manual prioritization and random ordering of test cases using the Cost-cognizant weighted Average Percentage of Faults Detected measure (APFDC) [9]. APFDC rewards test case orders proportionally to their rate of "units-of-fault-severity-detected-per-unit-test-cost". In our case, the cost of test cases is determined by their execution time. We calculated and compared APFDC for the test cases prioritized using ROCKET, manually prioritized and randomly ordered test sets (we used a median value for 100 sets of randomly ordered test cases).

### C. Results

Figure 2a) compares the execution time for the partial manually-prioritized and ROCKET-prioritized test suits, starting at 20% of the initial test suite size, with increments of 20%. In other words, the experiment analyses how many test cases will execute in a limited testing time for these two approaches. For the first 20% of the test suite prioritized using ROCKET, execution time is 40% less than for the 20% of manually prioritized test suite. For the first 40% of the test suite prioritized using ROCKET, execution time is still 28% less than the time required to execute 40% of manually prioritized suite. These results indirectly show that ROCKET maximises the number of test cases that can run in available testing time, compared to manual approach, positively answering RQ1. As the size of a test suite grows, the difference in test execution time for the two approaches decreases and, as expected, equals zero after the whole test suite has been executed.

Figure 2b) compares the number of detected faults for the partial manually-prioritized and ROCKET-prioritized test suits, starting at 20% of the suite total size, with increments of 20%. For 20% of the test suite executed, the test cases prioritized by ROCKET are able to detect 30% more regression faults comparing to manually prioritized test cases; 13 faults detected by ROCKET-prioritized test cases versus 10 faults detected by manually-prioritized test cases. These results show that ROCKET has higher regression fault detection rate compared to manual prioritization, positively answering RQ2.

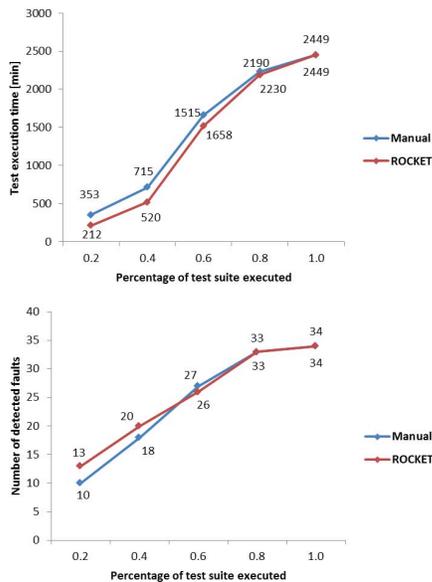


Fig. 2. a) Test execution time and b) Fault detection for manually-prioritized and ROCKET-prioritized test suite.

When using APFDC measure to compare the effectiveness in prioritizing regression tests for ROCKET approach versus manual prioritization and random ordering of test cases, we found that ROCKET outperforms other two prioritization techniques. In particular,  $APFDC_{ROCKET} = 17.09$ ,  $APFD_{manual} = 15.81$ , and  $APFD_{median\_random} = 13.85$ . These results show that ROCKET has higher rate of regression

fault detection per unit of test case cost (test execution time).

In summary, the results of the case study show that ROCKET efficiently prioritizes test cases for faster and more efficient regression fault detection, maximizing the number of executed test cases in limited period of time. This means that in cases where time required to execute complete test suite exceeds available testing time, using ROCKET will assure that as much test cases that test regression faults will execute.

### D. Threats to Validity

In the study we used only one VCS software product to evaluate ROCKET with one test team, being a threat to external validity of our findings. However, the used VCS belongs to a class of configurable systems that are characterized by a high level of commonality between products. Therefore, we assumed that the VCS we used is representative of all VCS products and that regression testing process in other test teams does not significantly differ.

## V. CONCLUSION

In this paper we presented the approach that prioritizes test cases for continuous regression testing, ROCKET, so that the tests that take shorter time to execute and have higher regression fault detection capability execute earlier. When applied to industrial continuous regression testing of video conferencing software, ROCKET showed to improve the efficiency and time to detection of regression faults, compared with manual and random test ordering. In future work we will perform more thorough evaluation of ROCKET with another testing team and on more VCS products, and especially for a longer period of usage, after the tool featuring ROCKET has been completely adopted and internalized with our partner. Later, we will extend the objective function to multiple prioritization criteria, such as the cost to fix the failure and the cost of switching test cases in execution that require manual intervention.

## ACKNOWLEDGMENT

The authors thank to Marius Liaen from Cisco. This work is supported by the Research Council of Norway.

## REFERENCES

- [1] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Int. symp. on Soft. testing and analysis*, 2002.
- [2] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC*, 2003, pp. 128–137.
- [3] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *Proc. of the 18th IEEE Int. Symposium on Software Reliability*, 2007.
- [4] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing field failures in system configurable software: cost-based prioritization," in *ISRE*, 2009.
- [5] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *Int. Journal Systems Assurance Eng. and Management*, pp. 126–134, 2011.
- [6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *Proc. of the 16th Int. Symp. on Foundations of software eng.*, 2008.
- [7] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *ISTTA*, 2008.
- [8] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *ISSTA*, 2006.
- [9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd Int. Conference on Software Engineering*, 2001.

# Improving statistical approach for memory leak detection using machine learning

Vladimir Šor  
Plumbr OÜ  
Software Technology and  
Applications Competence Center,  
Ülikooli 2  
Tartu, Estonia

Tarvo Treier  
Tallinn University of Technology  
Ehitajate tee 5  
Tallinn, Estonia

Satish Narayana Srirama  
University of Tartu  
Liivi 2, Tartu, Estonia

**Abstract**—Memory leaks are major problems in all kinds of applications, depleting their performance, even if they run on platforms with automatic memory management, such as Java Virtual Machine. In addition, memory leaks contribute to software aging, increasing the complexity of software maintenance. So far memory leak detection was considered to be a part of development process, rather than part of software maintenance. To detect slow memory leaks as a part of quality assurance process or in production environments statistical approach for memory leak detection was implemented and deployed in a commercial tool called Plumbr. It showed promising results in terms of leak detection precision and recall, however, even better detection quality was desired. To achieve this improvement goal, classification algorithms were applied to the statistical data, which was gathered from customer environments where Plumbr was deployed. This paper presents the challenges which had to be solved, method that was used to generate features for supervised learning and the results of the corresponding experiments.

## I. INTRODUCTION

Memory leaks can be a major problem in all kinds of applications, depleting their performance and up-time, even if these applications run on platforms with automatic memory management, such as Java Virtual Machine (JVM). Automatic memory management relies on garbage collector to reclaim memory from objects that are no longer used. A memory leak in garbage collected languages happens when a portion of object graph, which is no longer used by the application code, is still strongly reachable, so that garbage collector cannot reclaim the memory occupied by such unused object graph.

Memory leaks can be rapid or slow. While rapid memory leaks will exhaust memory quickly, slow memory leaks contribute to software aging which will still exhaust memory eventually. Memory related software aging is studied for example in [1], [2], [3]. As rapid leaks show up quickly and grow fast they can be easily diagnosed in the development environment. Slow memory leaks, in contrast, require fair amount of time, days or weeks, to become noticeable.

Process of finding and diagnosing memory leaks in JVM is covered by a number of researchers. On a broad scale approaches can be divided into offline, online and hybrid. Offline approaches analyze heap dumps (e.g., [4], [5], [6], [7]). Online approaches use either instrumentation techniques available in the Java Virtual Machine to instrument the running code and monitor certain parts (e.g., [8], [9], [10], [11]) or

add some specific monitoring code directly to the JVM itself (e.g., [12], [13] modify garbage collector). Hybrid approaches combine data for the analysis from both online and offline sources (e.g., [14], [15], [16]).

In current study we improve statistical approach for memory leak detection [15] implemented in the commercial tool called Plumbr (<http://plumbr.eu>). The leak detection method used in the tool was chosen for several reasons – it can be used in existing production environments (because of using standard APIs and moderate overhead [15], [17]) to analyze software aging, it provides promising results in terms of detection precision and due to its hybrid nature, it can generate reports combining both online and offline data which can be read by non troubleshooting experts. Also the implementation of the method in Plumbr can provide feedback even before a memory leak will become a threat to the stability and performance of the application. However, as we'll demonstrate, statistical method can be substantially improved to provide even better memory leak detection quality by introducing more observable parameters which can be derived from already existing data set without inducing any additional performance overhead.

The paper is organized as follows: in section II the statistical approach and its shortcomings which had to be improved are briefly described. Section III describes efforts while applying classification algorithms for the memory leak detection problem. Experiment results and conclusions are presented in sections IV and V, respectively.

## II. STATISTICAL APPROACH FOR MEMORY LEAK DETECTION

In the core of the statistical approach for memory leak detection lies *weak generational hypothesis* – an observation that most newly created objects live for a very short period of time [18]. This observation is fundamental for generational garbage collectors [19], which are used in modern JVMs.

Main idea of the generational garbage collection is to handle objects based on their age and depending on the age use the garbage collector of the appropriate type to get best performance. The age of objects is measured as a number of garbage collection cycles (*generations*) which an object has survived. Statistical approach for memory leak detection makes use of this property to track how many different generations of live instances of a class are currently alive (we further call

this property *difGen*). The intuition for the method is that in case of a leaking class<sup>1</sup> *difGen* is growing unbounded, as leaking objects are continuously created but, due to a leak, are not collected. On the other hand, *difGen* of non-leaking objects will remain under certain value, which is specific to the combination of the application and the selection of garbage collectors.

It is clear, that performing such monitoring on a class level will generate many false positives, as some classes (e.g., String or Integer) are used in many places and thus can create a lot of noise. To mitigate such cases, actual monitoring in the implementation happens with *allocation* granularity. This means that algorithm distinguishes instances of a class created in different places. In theory it may miss some leaks, but, as we'll show in section III, our experiments indicated that such limitation performs very well in practice. As an additional feature, such monitoring approach will give allocation points for leaking objects, which may help fixing the leak. Such allocation information is not available when using heap dump analysis for memory leak detection.

The main challenge of the approach is to detect the unbounded growth of *difGen* for some set of allocations over time, as soon as possible. Proof of concept solution presented in [17] used simple threshold technique to distinguish between normal allocations and leaking allocations – if a set of *difGen* values for all allocations in the application can be separated into two clusters which are separated by some threshold value, then the cluster of allocations with greater *difGens* is leaking. Although, even this simple technique already showed good leak detection qualities [17], it has room for several improvements.

The most complex task was to choose the threshold which would separate two clusters of allocations. If the threshold is too low, then algorithm will detect too many false positives, if the threshold is too high, there will be too many false negatives and the danger that the application will run out of memory before algorithm will detect anything at all.

In addition, several factors affecting the quality of detection were identified. For example, application uptime and load affect how clusters of allocations are formed – in some cases it can be clearly seen that some allocations are clearly leaking, however other allocations in the applications have such *difGen* values that it is not possible to divide them into clusters using simple threshold approach. Detailed analysis of described method will be published as a separate publication.

Aforementioned observations led to the need for more attributes to base leak detection on.

### III. IMPLEMENTING MACHINE LEARNING

As was shown in section II, additional attributes could improve statistical memory leak detection. When using one or two attributes it is easy to analyze them by hand, but when using multiple attributes and considering their combinations, manual analysis becomes unfeasible.

To use a wider range of statistical metrics supervised learning type of machine learning algorithms was used. To

apply supervised learning, a set of examples of java allocations and their leakage statuses is required. Possible leakage statuses of a class are discrete: *leaking* and *not leaking*. If the outcome of the supervised learning is discrete, as in our case, then the function that associates the attributes' values and the leakage status is called a classifier and in that case supervised learning algorithms are also called *classification algorithms*.

The set of examples of java classes was used to train the classification algorithm. Depending on the type of classification algorithm the training result is formed as a set of rules, decision tree or other model to store the relationships between values of attributes and the leakage status. After training of the classification algorithm, generated relationships are used to predict leakage status of a new allocation based on the its respective attributes.

Attribute identification is the first task in the implementation of machine learning. Attributes of an allocation, which can be used for learning, can be any features which can be used to characterize the allocation, e.g., name of the class, number of live objects, *difGen* value, number of all generations and ratios of combinations of other attributes. Further on, a specification of an attribute and its value is called a *feature*.

To select attributes following data was used: identifier of the allocation and the numbers of generations in which instances of this allocations are still alive. Such statistic data was stored after each full garbage collection when it is guaranteed that no garbage is left in the heap. Statistical data also included the leakage decision from default statistical approach.

The full data set consisted of statistical data from 10,894 application runs, collected from 974 different application owners, who were using Plumb. To determine learning attributes, 44 applications from the full data set were chosen so that they would be as different as possible, based on their allocation and object age distribution. Difference between applications was derived from visual evaluation of plotted distribution of live generations of allocations over time. From previous analysis it was known, that 11 of these 44 selected applications have been given incorrect diagnosis by the statistical approach.

We used two human experts who were able to verify whether the leakage status of the class is leaking or not leaking and correlate their decision with the decision of the default approach. Human troubleshooting experts were using professionally similar memory leak detection approach based on the *difGen* value, however, they also used 'gut feeling' which was based on yet unidentified attributes. The main goal at this stage was to determine which additional attributes were they using, without formal specification. To give their resolutions, experts were allowed to use only the data available in the collected statistical data (also in visualized form). During reevaluation of these 44 applications, the decision process of experts was tracked by letting them document their decision process to identify such 'gut feeling' attributes.

Evaluation process was done in 2 iterations. During the first iteration experts worked independently. After the first iteration of determination, approximately 10% of resolutions were conflicting. In the second iteration experts worked together with the goal to resolve conflicts. After the second iteration we got final common determination of each class by experts. The result of the final determination was that 20 applications

---

<sup>1</sup>Leaking class is the one, instances of which are leaking

from 44, contained allocations with the leakage status *leaking*. The total number of allocations in all applications was approximately 130,000 and 0,3% of them were leaking based on expert resolution.

When making their decisions and resolving conflicts, experts were creating a log, which was further analysed and additional features were determined. Some of these features were usable for human experts, but not yet measurable. One example of such feature is a warm up period of an application. During the warm up period the first initialization process is taking place, a lot of classes are loaded and instantiated and no class can be considered leaking until warm up is finished. Warm up period of application could be separate task for machine learning and in this paper we ignored such features which we were not able to measure. For the experiments considered in this paper, the following six features were used to describe an allocation for classification algorithms:

- 1) overall uptime of an application,
- 2) *diffGen* (see section II),
- 3) uniformity of *diffGen* distribution,
- 4) ratio between live objects and *diffGen*,
- 5) ratio between *diffGen* jump and *diffGen*,
- 6) ratio between number of classes with the same *diffGen* jump and number of all classes,

Ratios, instead of absolute values, were used to normalize input data and improve feature comparison for classes from different applications.

Aforementioned six attributes generate continuous values, which were further discretized based on experts reasoning – for all features discrete values in the decision process log were described as *low* and *high* or *small* and *big*. From this was concluded that for all of the features, discrete values can be divided into three groups: *small*, *medium* and *large*.

#### IV. EXPERIMENTAL RESULTS AND ANALYSIS

For experiments, java allocations with known leakage status from 200 applications were taken. Exactly half of those 200 applications contained allocations with the leakage status *leaking*. Applications were separated into two sets for training and for testing of classification algorithms as follows. For the training set 67 applications which contained leaking allocations were randomly selected and also 67 applications, which consisted only of not leaking classes. So, the training set contained 134 applications in total. The rest 66 applications were selected into testing data set.

Training set consisted of approximately 288,000 classes and 0,5% of them were leaking. Testing set consisted of approximately 159,000 classes and 0,7% of them were leaking. Both training and testing group contained a lot of classes with exactly the same values of features.

Two classification algorithms were used for experiments: C4.5 [22] and PART [23]. These algorithms were chosen because both are known to perform well over a wide range of learning tasks, thus being general enough. On the other hand, C4.5 produces a decision tree, whereas PART produces the a rule list, thus using different approaches. For both algorithms implementation from Weka toolkit [24] were used (C4.5 is called J48 in Weka). Because training set contained

lot of classes with same feature values, both classification algorithms were trained and tested twice. First time algorithms were trained with all allocations from training data set. For the second training the same training data set was used, but duplicates (allocations with the same features values) were removed. After removing duplicate classes there were only 235 classes with unique features left, which is only 0.8% of all classes in 134 applications.

After training, PART generated the rule list consisting of 11 rules and C4.5 produced the decision tree with 7 leaves (which, after converting to rules, also makes up 7 rules). Six rules out of seven were similar with the ones from PART rule list. Three out of these six were exactly the same, while other three were partially overlapping.

In both cases, classification algorithms were evaluated using allocations from the testing group. Simple *diffGen* threshold based implementation of the statistical approach described in section II was also evaluated with classes from the testing group.

Algorithm	Group	TP	FN	TN	FP
Simple threshold		0,495%	0,231%	99,086%	0,189%
C4.5	w. dups	0,489%	0,155%	99,119%	0,237%
PART	w. dups	0,480%	0,127%	99,147%	0,246%
C4.5	w.o. dups	0,493%	0,060%	99,215%	0,233%
PART	w.o. dups	0,493%	0,060%	99,215%	0,233%

TABLE I. CONFUSION TABLE

Table I shows the confusion table describing experiment results. To conserve space, columns in the table mean following. *TP* – true positive, leaking application was correctly identified. *FP* – false positive, non-leaking application was identified as leaking. *TN* – true negative, non-leaking application was correctly identified. *FN* – false negative, leaking application was identified as non-leaking.

As training was performed on an unbalanced dataset (0.5% positives and 99.5% negatives), there was the risk that a classifier can be tempted to score all samples as negative as this will yield a high score. However, from confusion matrix it can be seen that such risk has not realized. Moreover, trained algorithms show smaller number of false negatives than simple threshold method.

To evaluate combined performance of the classification we use accuracy and  $F_1$  measure [25]. The accuracy is a ratio between correct predictions and all predictions. The  $F_1$  measure combines precision and recall with an equal weight in the following form:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \quad (1)$$

We measured  $F_1$  separately for detecting *leaking* and for detecting *not leaking*. If we measure  $F_1$  for leaking classes then precision is the percentage of classes predicted as leaking that were leaking and recall is the percentage of leaking classes that were predicted as leaking. If we measure  $F_1$  for not leaking classes then precision is the percentage of classes predicted as not leaking that were not leaking and recall is the percentage of not leaking classes that were predicted as not leaking. The results of our experiments are presented in table II.

Algorithm	Group	Accuracy	$F_1$ Leak	$F_1$ Not leak	Avg $F_1$
Simple threshold		0.9958	0.7023	0.9979	0.8501
C4.5	w. dups	0.9961	0.7135	0.9980	0.8558
PART	w. dups	0.9963	0.7202	0.9981	0.8592
C4.5	w.o. dups	0.9971	0.7713	0.9985	0.8849
PART	w.o. dups	0.9971	0.7713	0.9985	0.8849

TABLE II. EXPERIMENT RESULTS

From the results in table II, it can be clearly seen that inclusion of classes with same features for training results in an  $F_1$  score practically equivalent to the simple threshold based statistical approach. Removing duplicates from training set significantly improves classification performance.

It can also be seen that using only *diffGen* attribute alone, as in simple statistical approach, is enough to predict leaks with a score of 0.70, which means that this is the most dominant attribute. Adding 5 more parameters resulted in 7% better leak prediction performance. However, from the confusion table it can be seen that the overall improvement came from the decreasing number of false positives, which was produced by simple threshold method.

## V. CONCLUSIONS

This paper has shown that machine learning techniques can be successfully used to improve statistical approach for memory leak detection. For both tried learning algorithms (C4.5 and PART) improvement in leak detection quality by at least 5%, and in best case 7%, can be seen, compared to plain statistical approach when using learning data without duplicates.

As a future work we plan to further improve detection quality by formalizing more attributes used by experts into discrete features, which we cannot measure yet, such as warm up period.

Warm up problem was shortly mentioned in section III as a feature which can affect detection, but is not yet taken into account. Automatic warm up period detection can be also a useful technique for performance testing and assessment applications.

Uniformity, as it was defined in section III, is not performing well enough in some corner cases, when number of different generations is very large and generations of live objects are very densely and uniformly spread in one part of the distribution vector, so that standard deviation of distances between data points, which is taken as a measure of uniformity, doesn't reflect anymore that the data is condensed in one part of the vector. So, either tuning of the uniformity calculation or introduction of new attributes to handle such corner cases is needed.

## REFERENCES

- Macéanddo, A., Ferreira, T., Matias, R.: The mechanics of memory-related software aging. In: Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second International Workshop on. (2010) 1–5
- Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K.S.: Analysis of software aging in a web server. IEEE Transactions on Reliability **55** (2006) 411–420
- Grottke, M., Matias, R., Trivedi, K.: The fundamentals of software aging. In: Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on. (2008) 1–6
- Reiss, S.P.: Visualizing the java heap to detect memory problems. In: VISSOFT '09. (2009)
- Chilimbi, T.M., Ganapathy, V.: Heapmd: Identifying heap-based bugs using anomaly detection. In: ASPLOS XII: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems. (2006)
- Maxwell, E.K., Back, G., Ramakrishnan, N.: Diagnosing memory leaks using graph mining on heap dumps. In: KDD. (2010) 115–124
- Aftandilian, E.E., Kelley, S., Gramazio, C., Ricci, N., Su, S.L., Guyer, S.Z.: Heapviz: interactive heap visualization for program understanding and debugging. In: Proceedings of the 5th international symposium on Software visualization. SOFTVIS '10, New York, NY, USA, ACM (2010) 53–62
- Xu, G., Bond, M.D., Qin, F., Rountev, A.: Leakchaser: Helping programmers narrow down causes of memory leaks. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2011)
- Xu, G., Rountev, A.: Precise memory leak detection for java software using container profiling. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. (2008) 151–160
- Chen, K., Chen, J.B.: Aspect-based instrumentation for locating memory leaks in java programs. In: Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. Volume 2. (2007) 23–28
- Chilimbi, T.M., Hauswirth, M.: Low-overhead memory leak detection using adaptive statistical profiling. In: 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (2004) 156–164
- Jump, M., McKinley, K.S.: Cork: dynamic memory leak detection for garbage-collected languages. In: 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07), ACM (2007) 31–38
- Bond, M.D., McKinley, K.S.: Tolerating memory leaks. In: OOP-SLA'08. (2008)
- Mitchell, N., Sevitsky, G.: Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In: ECOOP'03. (2003)
- Šor, V., Srirama, S.N.: A statistical approach for identifying memory leaks in cloud applications. In: First International Conference on Cloud Computing and Services Science (CLOSER 2011), SciTePress (2011) 623–628
- Xiao, X., Zhou, J., Zhang, C.: Tracking data structures for postmortem analysis: (nier track). In: Software Engineering (ICSE), 2011 33rd International Conference on. (2011) 896–899
- Šor, V., Srirama, S.N., Salnikov-Tarnovski, N.: Automated statistical approach for memory leak detection: Case studies. In: Lecture Notes in Computer Science. Volume 7045, Part II., Springer Verlag (2011) 635–642
- Lieberman, H., Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. Commun. ACM **26** (1983) 419–429
- Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. SIGSOFT Softw. Eng. Notes **9** (1984) 157–167
- Oracle corp. <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>: (Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning)
- Quinlan, J.R.: Learning logical definitions from relations. Machine Learning **5** (1990) 239–266 10.1007/BF00117105.
- Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
- Frank, E., Witten, I.H.: Generating accurate rule sets without global optimization. In: Proc 15th International Conference on Machine Learning. Madison, Wisconsin, Morgan Kaufmann (1998) 144–151
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. NewsL. **11** (2009) 10–18
- van Rijsbergen, C.J.: Information Retrieval. 2 edn. Butterworths (1979)

# Large-Scale Automated Refactoring Using ClangMR

Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, Zhanyong Wan  
 Google, Inc.  
 Mountain View, California 94043  
 Email: {hwright,djasper,klimek,chandlerc,wan}@google.com

**Abstract**—Maintaining large codebases can be a challenging endeavour. As new libraries, APIs and standards are introduced, old code is migrated to use them. To provide as clean and succinct an interface as possible for developers, old APIs are ideally removed as new ones are introduced. In practice, this becomes difficult as automatically finding and transforming code in a semantically correct way can be challenging, particularly as the size of a codebase increases.

In this paper, we present a real-world implementation of a system to refactor large C++ codebases efficiently. A combination of the Clang compiler framework and the MapReduce parallel processor, ClangMR enables code maintainers to easily and correctly transform large collections of code. We describe the motivation behind such a tool, its implementation and then present our experiences using it in a recent API update with Google's C++ codebase.

## I. INTRODUCTION

As software systems evolve, existing code often must be updated to allow for future feature development, removal of old or incompatible interfaces, and further maintenance tasks. Large software systems often suffer from an inability to evolve to meet new demands [1], largely due to increasing amounts of technical debt [2], and the inability of code maintainers to automatically update large portions code in a semantically safe way. Even automatic changes which may appear safe in isolation may introduce semantic conflicts that cause faults which are difficult to detect [3].

Google addresses this challenge by using ClangMR, a tool that uses semantic knowledge from the C++ abstract syntax tree (AST) to make editing decisions. ClangMR programs are written in C++ and provide a wide variety of different refactoring capabilities. ClangMR also takes advantage of the independent nature of most refactoring work by parallelizing its analysis across many computers simultaneously by using the MapReduce framework [4]. This combination of knowledge, flexibility and speed allows code maintainers to perform complex transformations across millions of lines of C++ code in a matter of minutes.

While this specific implementation of ClangMR is dependent upon Google's infrastructure, a significant portion of the system is available as open source software as part of the LLVM project [5]. The Clang AST and its node traversal and matching infrastructure are all readily available for public consumption and improvements continue to be publicly released.

In the following pages, we present the basic implementation details of the ClangMR system in use at Google. We also

discuss a sample large-scale refactoring effort recently completed which modified over 35,000 function call sites across 100 million lines of code.

### A. Existing Tools

ClangMR is not the first tool designed to do complex refactorings over C++, but it is unique in its flexibility, speed, and use in industrial applications.

While useful in simple cases, traditional regular-expression-based matching tools lack the semantic knowledge that complex transformations often require. For instance, these tools can not distinguish calls to similarly-named methods which are members of different classes, making them unsuitable for large-scale semantically-safe refactoring. Such naming conflicts also frequently increase as the size of a codebase grows, leading to a lack of scalability.

Many integrated development environments, such as Eclipse, provide a limited set of refactoring tools. While these tools take advantage of compile-time knowledge, they are generally limited to a single file or package, and only support the operations built into the tool. Since these tools run on a developer's local workstation, processing large collections of source code is often intractable.

Other tools, such as Pivot [6], may be versatile, but have difficulty scaling to many millions of lines of code. Some tools that do scale, such as those described by Kumar, et al in [7], perform specific types of transformations at scale, but lack the versatility of ClangMR. While these are useful in theory, we have yet to see existing tools demonstrated on a large, real-world corpus of production-quality code.

## II. MOTIVATION

Google maintains a large mixed-language codebase in a single repository, with a significant portion written in C++. Like many large software systems, this codebase continues to evolve as new APIs, idioms and standards are introduced. While new code may use the improved techniques, large amounts of legacy code still uses old APIs and standards.

To help maintain as small an API surface as reasonable, the developers and teams responsible for introducing new core APIs are also tasked with removing old ones and migrating existing callers to the new abstractions. Google engineers are also actively engaged in source code rejuvenation efforts to migrate custom implementations to the C++11 standard [8].

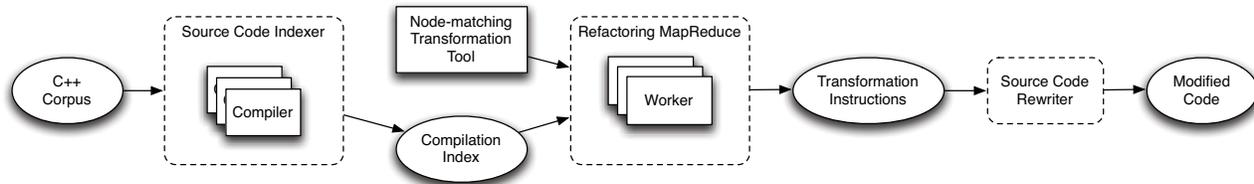


Fig. 1: ClangMR processing pipeline

ClangMR has seen wide use within Google for the past two years, and it continues to evolve into a more powerful and easy-to-use tool. Recent refactoring projects include:

- Updating legacy C++98 code to take advantage of features in the C++11 standard.
- Removing redundant explicit type conversions.
- Updating callers to improved APIs, such as string manipulation and file handling.

While these may seem trivial in isolation, performing these migrations on millions of lines of code would not be practical without ClangMR. In general, ClangMR helps reduce the accumulated technical debt of a diverse codebase built over the course of more than a decade. An example of a specific migration effort is discussed in depth below.

### III. IMPLEMENTATION

The ClangMR implementation consists of three parts:

- an indexer which describes how to compile the C++ codebase into a collection of ASTs
- a transformation-specific node matching tool which builds ASTs from the index, matches applicable AST nodes, and outputs editing instructions
- a source code refactoring which consumes editing instructions and modifies the files in a local version control client to effect the desired transformations

Of the three separate components, only the transformation-specific node matching tool is specific to an individual transformation. The compilation index can be consumed by multiple node matching tools, and the refactoring can operate on their standardized output. This architecture means a single code maintainer needs only implement an appropriate matcher for the desired transform, rather than the entire pipeline. An overview of the pipeline is shown in Figure 1, and the individual components are discussed below.

#### A. Source Code Indexer

A daily task builds a denormalized index of the acts of compilation—the commandlines used, the various files read, and the filesystem layouts of those files. Storing the entire precomputed ASTs would be space-prohibitive, but this index serves to provide a way to quickly construct those ASTs from a snapshot in source code history. Determining these compilation steps is largely independent for individual files, so this process can be parallelized across a number of different machines in Google’s standard build cluster.

#### B. Node Matcher

The node matcher uses the most recent compilation index to produce ASTs for the entire collection of source code. Because this intermediate AST is only required for node traversal, it can be stored in memory for the duration of the traversal. Even though the size of the AST can be quite large, traversing the AST demonstrates high memory locality, so it is quite fast. Experience shows that it is roughly as fast to compile C++ code into a memory-held AST as it is to read a completely annotated AST out of distant storage.

Because each translation unit has a separate root node in the AST, and each source file is generally an independent translation unit, the node matcher can operate on separate translation units in parallel. At Google, we use the MapReduce framework [4], but ClangMR could be adapted to use other suitable parallelization systems.

Most of the node matching executes outside the care or control of the programmer performing the refactoring. As input, a developer provides an appropriate node matching expression, and a callback to be invoked when that expression is matched. In practice, these tend to be relatively small: a few hundred lines of C++ code. The ClangMR infrastructure handles running the node matching algorithm and invoking the callbacks on the appropriate nodes.

The node matching infrastructure first reads the index from bigtable, uses it to recursively traverse the AST nodes, and then invokes any callbacks that have been registered for matched nodes. Any output produced by the callbacks is then serialized to disk for use by the refactoring.

1) *Node Matching Expression*: Developers use node matching expressions to register a callback with the ClangMR processor. These expressions may match a variety of node types, and can be qualified with various logical filters and traversal operations. Examples of node matching expressions used at Google are shown in Figure 2, and a complete reference is available on the Clang website [9].

2) *Callbacks*: When the preprocessor matches a node in the AST, it invokes the provided callback with the node and some context about where it was found, such as the source location. The callback is written in C++, allowing it to query the properties of the node and its context and make complex decisions about what edits, if any, can be applied. They may also decide to not make any edits. This technique allows for much more powerful transformations than pure textual substitution.

```
StatementMatcher matcher =
  callExpr( allOf(
    argumentCountIs(1),
    callee( functionDecl( hasName(
      ":: Foo:: Bar" ) ) ) ) ) )
  .bind("call");
```

(a) Match all calls to `Foo::Bar` which have a single argument

```
TypeLocMatcher matcher =
  loc( qualType( hasDeclaration(
    recordDecl( hasName(
      ":: scoped_array" ) ) ) ) ) )
  .bind("loc");
```

(b) Match all `scoped_array` typed variable declarations

Fig. 2: Example node matching expressions

As output, each callback may generate a set of instructions on how to transform the code on a textual level. Similar to a text-based patch, these instructions describe edits to the target source file as a series of byte-level offsets and additions or deletions. These instructions are then serialized to disk, and used as input to the source code refactorer.

An example callback implementation is shown in Figure 3. Much of the error checking and boilerplate has been omitted, but combined with the matcher in Figure 2a, this example renames all calls to `Foo::Bar` to `Foo::Baz`, independent of the name of the instance variable, or whether it is called directly or by pointer or reference.

### C. Refactorer

The source refactorer reads the list of edit commands generated by the node matcher callbacks and filters out any duplicate, overlapping or conflicting edit instructions before editing the source code in the local version control client. Each edit is processed serially in the version control client on the local workstation of the developer, which is synchronized to the version of code stored in the compilation index.

Even though it is local and serial, in practice, this step is relatively quick, and edits spanning thousands of files are performed on the order of tens of seconds. A final pass with `ClangFormat`, a Clang-based formatting tool [10], ensures the resulting code meets formatting and style guide recommendations.

### D. Limitations

While ClangMR enables a large class of refactoring operations at scale, it does have limitations. ClangMR can only refactor changes which are self-contained within translation units. Large sets of changes still require tedious manual review—though review tools are improving to allow faster automated review of large changes. Finally, ClangMR requires learning some nuances of the Clang AST, which requires developer investment.

```
void Refactor(const MatchFinder::MatchResult& res) {
  Clang::CXXMemberCallExpr* call =
    res.Nodes.getStmtAs<clang::CXXMemberCallExpr>(
      "call");

  const clang::MemberExpr* member_expr =
    llvm::dyn_cast<clang::MemberExpr>(
      call->getCallee());

  EditState state(res, call);
  state.ReplaceToken(member_expr->getMemberLoc(),
    "Baz");

  Report(&state);
}
```

Fig. 3: Example node matcher callback

In spite of these limitations, ClangMR enables engineers to make significant semantically-correct changes to large C++ codebases.

## IV. PRACTICAL APPLICATION

In this section, we present an actual large-scale transformation done at Google using ClangMR and demonstrate the technical advantages to this approach. While not the largest or most complex refactoring performed done at Google, this example demonstrates the versatility of the AST-based refactoring approach.

### A. Splitting Strings

Google’s internal software libraries have historically provided a number of methods for splitting strings. One of the most common APIs is known as `SplitStringUsing`, shown in Figure 4. As the name implies, `SplitStringUsing` parses a string of characters using a set of delimiters, and inserts the resulting substrings into the provided vector of strings.

Google engineers recently introduced a single method to unify and consolidate the various split-related APIs. Known as `strings::Split`, it is shown in Figure 5. This new API is appropriately parameterized to handle the use cases of most existing split functions in a single interface. While a complete discussion of `strings::Split` is out-of-scope for this paper, suffice it to say that the new API was well received by Google engineers, led to reduced numbers of bugs and has been proposed for the next iteration of the ISO C++ standard [11].

After the new API debuted, most Google developers were not anxious to invest the effort to migrate their currently-functioning code to `strings::Split`. Due to the semantic difference between the APIs, any automatic transformation would require semantic knowledge, not just a strict textual substitution. At the time, there were roughly 45,000 callers of `SplitStringUsing`, and migrating them by hand was infeasible. One software engineer was attempting to convert these callers manually using a combination of inspection and editor macros, but his efforts could not keep up with an

```

void SplitStringUsing(const string& full,
                    const char* delimiters,
                    vector<string>* result);

void foo() {
    string input;
    vector<string> output;
    ...
    SplitStringUsing(input, "-", &output);
}

```

Fig. 4: SplitStringUsing example

```

namespace strings {
template <typename Delimiter, Predicate>
Split(StringPiece text, Delimiter d, Predicate p);

struct SkipEmpty {
    bool operator()(StringPiece sp) const {
        return !sp.empty();
    }
};

void foo() {
    string input;
    ...
    vector<string> output =
        strings::Split(input, "-",
                      strings::SkipEmpty());
}

```

Fig. 5: strings::Split example

evolving codebase. ClangMR allowed us to migrate the bulk of existing callers and encourage engineers to use the new API in new code.

1) *Node Matching Implementation*: Instead of using a node matching expression to decide which kinds of calls were transformable, the expression simply matched *all* calls to `SplitStringUsing`, and relied upon logic in the callback to determine if the transformation was safe or not. A “safe” transformation meets the following criteria:

- The output variable declaration was in the same scope as the call to `SplitStringUsing`.
- The output variable was not referenced between its declaration and the call to `SplitStringUsing`.

Both of these criteria were easily examined using the context available in the AST provided by ClangMR. In some complex situations, we chose to defer the edits to be done manually.

Because ClangMR allows the examinations of literal values known at compile-time, we were also able to determine how to rewrite the actual function calls themselves. The default behavior of `SplitStringUsing` is to use any of the provided characters as a delimiter, but doing so with `strings::Split` this requires a separate `Delimiter` argument, which could be omitted in the case of only one delimiter character. By examining the literal delimiter arguments to the old function call, we could simplify a large number of

common cases when rewriting to the new code.

2) *Experiences*: The initial ClangMR program transformed about 35,000 callers of `SplitStringUsing` to `strings::Split`, and these changes were mailed for review in 3,100 separate chunks, though not all simultaneously. These were often reviewed quite quickly, with an 80th-percentile review time of just over two minutes. The bulk of reviews were completed over two months, with a small number requiring another month to complete.

One benefit of using ClangMR for this work was that the transformation could be repeated. During the course of this effort, we frequently re-ran the tool to find any additional uses which had been added since the initial run. This made it easy stay current with an ever-changing codebase.

## V. CONCLUSION

In this paper, we presented the design and implementation of ClangMR, a highly parallelized, semantically-aware refactoring tool based upon the Clang compiler running on MapReduce. We also discussed an example application of a real-world large-scale transformation using the ClangMR system to update callers of deprecated APIs.

ClangMR allows for fast and versatile refactoring of large C++ codebases, and has been applied to many problems within Google, enabling maintainers to keep millions of lines of C++ code nimble and accessible to thousands of engineers.

## REFERENCES

- [1] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1–12, 2001.
- [2] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, “An enterprise perspective on technical debt,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 35–38.
- [3] G. Thione and D. Perry, “Parallel changes: detecting semantic interferences,” in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, vol. 1, 2005, pp. 47–56 Vol. 2.
- [4] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251254.1251264>
- [5] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.
- [6] B. Stroustrup and G. Dos Reis, “Supporting sell for high-performance computing,” in *Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 458–465.
- [7] A. Kumar, A. Sutton, and B. Stroustrup, “Rejuvenating c++ programs through demacofication,” in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 98–107.
- [8] P. Pirkelbauer, D. Dechev, and B. Stroustrup, “Source code rejuvenation is not refactoring,” in *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM ’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 639–650.
- [9] (2013) AST matcher reference. [Online]. Available: <http://clang.llvm.org/docs/LibASTMatchersReference.html>
- [10] (2013) ClangFormat. [Online]. Available: <http://clang.llvm.org/docs/ClangFormat.html>
- [11] G. Miller. (2013) std::split(): An algorithm for splitting strings. [Online]. Available: <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2013/n3510.html>

# Assuming Software Maintenance of a Large, Embedded Legacy System from the Original Developer

William L. Miller, Ph.D.  
Georgia Tech Research Institute  
Electronic Systems Laboratory  
Atlanta, Georgia, USA  
bill.miller@gtri.gatech.edu

Mr. Bruce L. Woodmansee  
CECOM, SEC  
Aberdeen Proving Ground, MD USA  
Department of the Army Civilian  
bruce.l.woodmansee.civ@mail.mil

Lawrence B. Compton  
Georgia Tech Research Institute  
Electronic Systems Laboratory  
Atlanta, Georgia, USA  
larry.compton@gtri.gatech.edu

**Abstract**—Transferring software maintenance and support for a mission critical system from one organization to another demonstrates the importance of having a complete and accurate documentation and anticipating the potential end-of-life issues for software development tools. The authors present the challenges and lessons learned during the transfer of responsibility for all future software maintenance for a large, real-time, mission critical system from the original developer to an organization that was not involved in the original development or any prior maintenance of the software application.

**Keywords**—documentation, lessons learned, end-of-life, life cycle cost

## I. INTRODUCTION

In the summer of 2009, the U.S. Army contracted Georgia Tech Research Institute (GTRI) to aid them in the assumption of the software maintenance for the Operational Flight Program (OFP) for one of their major defensive Electronic Warfare (EW) systems from the Original Equipment Manufacturer (OEM). This system resides on fixed and rotary wing aircraft. This original engagement by the Army was the start of a multi-year effort to transition the ability to completely support all software maintenance aspects of a very complicated embedded software application from one organization to another. This paper describes some of the significant issues and lessons learned during the transition.

The overall process for transferring the maintenance of the OFP first involved completely setting up GTRI as the software maintenance facility then later transferring the tools, processes, documentation, knowledge and software to the U.S. Army. As the task of initially establishing a facility for the software maintenance for this system, (that had been deployed nearly 6 years earlier, but had begun development 10 years prior to deployment) progressed it became apparent that the major challenges would not be in creating processes or

developing a team that would be capable of technically sustaining the software. Most of the transfer issues have centered on insufficient information or information in the wrong format. One would think that it would be relatively easy to obtain the correctly versioned documentation, but this proved to much more difficult than originally anticipated.

## II. TRANSITION CHALLENGES

### A. The Software Development Environment

The problems with the transfer began very quickly as GTRI attempted to create a software development environment (SDE) for the OFP that replicated that of the OEM. Because the system had initially been developed over 15 years ago, many of the components were past the end of life for availability and supportability. This included the target processor, the compiler and the host development environment.

Embedded systems have a unique coupling relationship between the target processor, the compiler and the host development system. Once a target processor is selected, a hardware system is built around it. The creation of the hardware system is an exhaustive, expensive and time consuming process involving functional and environmental testing. For a system that has been deployed to 2400 total aircraft (24 different types of aircraft ) performing a target processor upgrade is a hugely involved activity that typically only happens once during the entire life of a system. The compiler for the OFP must obviously produce code that runs on the target processor. The software maintenance organization should, but doesn't always, upgrade the development environment as new releases are made available by the compiler vendor. Unfortunately, as the system ages, it will at some point become likely that the compiler vendor will stop issuing updates. Another potential issue is that, the compiler company may get purchased or it may just go out of

business. A third potential issue could occur if the OEM does not apply software upgrades as they are issued by the compiler vendor. What this means for the software maintenance organization is that the software development environment becomes frozen; it may not even be possible to upgrade the host computer or operating system. While not ideal, but tolerable for the original developer, this can, and did present a huge problem for an organization assuming the maintenance function. The act of building up a software development environment capable of producing code that is identical to that produced by the OEM was non-trivial.

#### *B. Finding a Compiler*

Procuring the compiler for the OFP was the initial challenge that the team faced. The source language for the OFP was Ada-83, targeted to a microprocessor that is out of production. The compiler was hosted on an obsolete platform running an obsolete operating system. By 2010, not one of these items was commercially available. The company that had originally produced the compiler no longer existed. While it would have been possible to copy an image of the compiler being used by the OEM, the question as to whether or not it would be legal needed to be answered. This entailed tracing the ownership of the original company. It turned out that the original compiler manufacturer had been purchased by another company in the early 1990's. Then around 2000, the purchasing company was acquired by a large US IT vendor. As it turned out, a bit of negotiation with the vendor's lawyers and a few (actually a lot) of dollars resulted in the ability for the Army to obtain an image of the compiler from the OEM for its own use.

#### *C. The host development system*

After the issue of the obtaining a compiler was resolved, it was necessary to purchase the appropriate host computer system for the compiler to run on. The problem here was that neither the computer nor the computer's operating system were commercially available. It was necessary to get imaginative about where to look for working, obsolete computer systems. The solution for this, simply enough, was finding a number of the systems with the appropriate operating system on eBay. Because of the obsolescence of the processor and the very low price on eBay, a large number of them were purchased to include extra systems as spares for future requirements.

#### *D. Validating the Software Development Environment*

Once the job of acquiring the tools was complete, it was necessary to assemble the tools and verify that their functionality was exactly equivalent to that of the OEM. We determined that best way to do this would be to create a binary image of the system with the new environment and compare it, bit-for-bit, with the binary image delivered by the OEM. The concept, in principle is an easy one. The importance of successfully conducting a binary comparison with the OEM-delivered image may, at first, seem overstated. However, lives depend upon the performance of this system, and the system

performance has very critical timing attributes. Any changes to the system performance at all would require a complete retesting of the system which would necessarily include flight and live-fire testing (i.e., testing that requires the aircraft to operate in a threatening environment) to ensure the system remained within the required specification. This type of testing is extremely expensive and can run into the millions of dollars. With all this being said, the importance of the binary comparison becomes more obvious. Creating a system that was precisely, bit-for-bit, identical to the OEM-delivered software was a non-trivial event. The OEM-delivered software consisted of 2,000+ source files, but no make files or build scripts. Not having any of the OEM's build scripts or make files required GTRI to incrementally reverse engineer these tools based upon the delivered binary image. This activity alone took 2 engineers nearly 6 months. While tedious, time consuming and somewhat expensive, this course of action was much less expensive and technically less risky than creating a new binary and conducting new full flight and live fire testing of the resultant software.

#### *E. Assimilating the Software*

Once the software development environment was fully documented and validated, it was possible for GTRI engineers to provide the Army instructions on how to create and use an identical system at its software support facility in. Creating the second software development environment based upon a completely document tool package was, as expected, much easier that engineering the first system. The recreation of the system was conducted by Army engineers using the tools and documents created by GTRI. This step verified the completeness of the package created by GTRI.

The preceding discussion only addresses the SDE for the OFP. Additionally, the OEM had created a second software development environment for the OFP BootROM. Although changes to the BootROM software by the OEM were much less frequent than those made to the OFP, it was important to assume maintenance capability for all aspects of this EW system. As the initial investigation began, it became obvious that the documentation describing the tools for building the BootROM as well as the description of compiling and building the loader were out of date and highly inaccurate. Ultimately, it was necessary to conduct a complete audit of a recreation of the BootROM by the OEM to insure that the correct tools and procedures were being used, such that an identical BootROM could be created by the Army.

The process of procuring the software development environment and using the environment to create a functional binary image took approximately 1 year. The reader should keep in mind that after one year GTRI had simply accomplished the task of recreating an environment that enabled the building of a functional binary image. Assimilating the knowledge of the application itself, to the point where it was possible to confidently modify the application proved to be an additional 2+ year effort by a joint U.S. Army and GTRI team. When compared to building a software development environment, assimilating the software

proved to be a task of equal if not greater challenge, This challenge centered on the understanding the design of the embedded application and its supporting documentation.

#### *F. Documentation*

In total, there are in excess of 500 different documents that are required to understand and totally support this software application. 500 documents may seem excessive. But the reader must consider that this equipment is deployed on more than 24 different type of vehicles (e.g., UH60, AH64, etc.). Each vehicle type has a number of documents supporting it (e.g., EW system to vehicle interface descriptions, which are different for every vehicle type and documents describing the interface between this EW system and other EW systems). Almost immediately, two problems related to the documentation were identified: the U.S. Army had not received every one of the documents and the documents received were not always current, accurate or of the correct version. As soon as it was determined that documents were missing or not correct, the effort was to identify the correct version every document referenced in every other document and compile a comprehensive list of documents, drawings, reports. This began a long procurement process that is still underway.

Most of the design documents (approximately 120 different documents) that are typically updated during a software revision cycle were ultimately delivered by the OEM to the U.S. Army (however, as of this writing, there are still documents that have not been delivered). Unfortunately, for most of the delivered documents, they were delivered in the wrong format. The documents which for the most part, were heavily table-based were delivered in PDF format. While deliveries of documents in this format are great for reference, they were inadequate to be used as a basis for on-going support because they could not effectively be automatically converted from PDF to Microsoft Word because of all the tables. Consequently, it was necessary to manually recreate approximately 18,000 pages of MS Word documentation from the PDF documents. While not technically difficult, it was time consuming and expensive, keeping 4 clerks busy for nearly 8 months.

#### *G. Test Equipment*

Another item that affected the transition was the test equipment. Because of the complexity and critical real time aspects of this application, integration and functional testing prior to flight and live fire testing is vital. Providing a simulated operating environment requires a large and sophisticated test stand. For this application, the OEM constructed a test stand that would play back pre-recorded flight data, causing the processor and software to believe that it was mounted and operating in a flying vehicle. This OEM-produced system suffered from the same issues as the processor and the software in that it was produced with parts that had become obsolete. Because having a test system is absolutely required and the current system was beyond repair,

the Army contracted GTRI to design and build a functional replacement for the OEM's test system. Defining, designing and building a test system is not a difficult task. However, because the OEM's test system had been organically developed over many years, a firm and comprehensive set of requirements for the test system did not exist. Consequently, it was necessary for GTRI to extract/infer requirements from a number of sources such as test data recordings and test plans. Again, we have another example where the transition project suffered from a lack of documentation. In order to validate the operation of the redesigned test equipment against the legacy OEM produced system, GTRI designed and executed a comprehensive comparative test plan for system requirements and flight data playback. Finally, over 300 hardware-in-the-loop software functional qualification test cases were rewritten for use on the new test equipment. The redesigned test cases were crosschecked to ensure continued traceability to the software requirements under test. The GTRI efforts to redesign the software test infrastructure enhance test hardware scalability and sustainability and provide a flexible platform for software unit test.

### III. LESSONS LEARNED

The project to create this EW System was large, involved and expensive. It entailed hundreds of people involved in the creation of a new hardware system, a new software system and an entire infrastructure to support it for many years. At its inception, the concept of transitioning support from one organization to another was most likely never considered. In the future, the procuring activity should evaluate the possibility that the original OEM may not always have the responsibility for maintenance of the software. For a system that contains software that could potentially exist for 20 years or more, there are significant considerations to take into account. Companies go out of business, companies get purchased, business relationships change, software tools become obsolete and many other issues can arise or evolve that can affect the ability of an organization to effectively maintain a software system.

For large and complex systems, such as this EW system, there are a number of lessons that were learned, things that should be done differently on future large and long-lived projects. Some of the lessons apply to activities surrounding the creation of the initial contract. Some of the lessons apply to the maintaining organization. To be effective, these lessons must be considered and applied at the front end of a large project. Whereas they might appear to add significant and unnecessary costs to the project, they should be viewed as critical risk mitigation expenses. It is likely that the total cost of ownership over the life of the system would be reduced if these risk mitigation approaches are implemented.

With regard to the software development environment(s), ideally, the organization performing software maintenance would insure that upgrades are conducted periodically.

Upgrading an SDE (hardware and/or software) for an embedded system upon which lives depend is a very significant undertaking. It is possible/likely that there will be performance and timing differences. These differences will require significant and costly testing. The cost of this testing should not be a factor in conducting upgrades; it should only be a factor in the frequency of the upgrades. The more difficult issue that will certainly arise for a long-lived system is the ultimate end-of-life for an SDE component. It is impossible to predict the options that will be available at the time an SDE component is no longer supported by a vendor. However, when the situation arises, staying with the component is probably not one of the options that should be considered, unless the system under consideration is near its end of life as well.

A critical activity that was ignored during the life of this program, and by the way, is frequently ignored in the life of many long-life programs is an occasional, complete and exhaustive audit of the software compilation and build process against the extant software documentation. This audit would be a formally witnessed activity conducted by a technically competent but independent group/agency. This audit would insure that every tool version is accurately identified, that every script and library are controlled and delivered and that by following the developers "directions" a binary match can be obtained. This is not only good engineering practice, it would have aided greatly in establishing a functional SDE during the transition.

With regard to system documentation, the lessons learned are twofold. First, if an outside entity is developing and maintaining the software for you, insure that you have every document, and insure that every document is accurate for every release. Secondly, insure that you get an editable electronic version of every document at every release.

With regard to test equipment, it should be treated the same as the primary hardware and software. Hardware, software and development environments for the test equipment should be kept current. Moreover, complete design and implementation support documentation should be developed, maintained and delivered. And, there should be an occasional audit of the software build activities for the test equipment as should also be done for the operational software.

A significant issue for this system that should have been addressed at some point prior to the transition of the maintenance activity was the number of system components that were simultaneously obsolete and past end of life. While it might be possible to stockpile hardware components or purchase software licenses that are transferable in the future, these strategies only perpetuate an obsolete implementation. Should a system survive long enough to outlive the life of some of its components, it's will have been determined to be a critical and important system that will most likely continue to exist and provide a valuable service. When multiple key components of such a system become obsolete, there will be a time when it will be necessary to redesign the system based upon contemporary tools and components. There is no firm guideline for how many obsolete major system components would suggest a reimplementation is appropriate, however, the obsolescence of two major components should be considered as an event to trigger the process of considering a reimplementation. There are times when a re-implementation is conducted. However, the authors of this paper are unaware of any EW systems that are re-implemented simply to overcome obsolescence issues. Re-implementation is typically justified as a system performance improvement activity. For systems such as the one described in this paper, system performance improvements, might be deferred until a newly implemented baseline hardware and software system exists.

#### IV. SUMMARY

Assuming responsibility of an application is a much bigger task than simply learning how the application works. The assumption of an application should begin during the initial procurement of the system to insure that all necessary tools and documentation are identified as deliverable. Periodic upgrades to development environments and test equipment should be required. Finally, if the system is going to exist for a long period of time, it might be wise to consider that the original developing company might not be the ultimate sustainer.

# The Adventure of Developing a Software Application on a Pre-release Platform: Features and Learned Lessons

Clairton Siebra  
Informatics Center  
Federal University of Paraiba  
Joao Pessoa, PB - Brazil  
clairton@di.ufpb.br

Angelica Mascaro  
CIn/Samsung Project  
Federal University of Pernambuco  
Recife, PE - Brazil  
aam3@cin.ufpe.br

Fabio Q. B. Silva, Andre L. M. Santos  
Informatics Center  
Federal University of Pernambuco  
Recife, PE - Brazil  
{fabio,alms}@cin.ufpe.br

**Abstract**—From September 2011 to May 2012, Microsoft delivered three pre-releases of Windows 8. Its principal goal was to present the features of this operating system (OS) and motivate the development of new applications to this platform. However, as any pre-release version, several parts of the code and related documentations were incomplete. Furthermore, its APIs were very prone to be changed until the final release. Even considering such risks, the company X decided to face these challenges and create an application to be embedded into Windows 8 tablets as soon as this OS was released. This paper discusses interesting aspects of this adventure and some lessons that may be important to development teams that intend to trail this kind of project. Metrics are also used to better characterize some of these aspects.

**Keywords**—, software development, risks management, metrics

## I. INTRODUCTION

In the software engineering literature, the term *pre-release* means an early version of a software product that may not contain all of the features that are planned for the final version [1]. Typically, software goes through two stages of testing before it is considered finished. The first stage, called alpha testing, is often performed only by users within the organization developing the software. The second stage, called beta testing, generally involves a limited number of external users. However, in some few cases, these versions are released in public domains. This approach is very common in the computer games market, as a way to awaken curiosity and create an expectation about the final release.

This pre-release strategy is also appropriate to operating systems (OS) makers, once applications developers can have an early contact with new features of an OS and use such features to develop new applications early. However, for any pre-release version, developers must be aware about some issues. For example:

- Incomplete packages and APIs: pre-release versions are incomplete and do not implement all functions that they intend to have. For example, some functions may present only their interfaces rather than implemented classes;
- Unstable implementations: classes are in ongoing development and even complete classes may be changed;
- Partial evaluation: several parts of the code are not fully evaluated;

- Incomplete documentation: documents are mostly incomplete or non-existent.

We see there are several issues, and consequently risks, that an application development team should consider when they intend to develop to a pre-released OS. However, even considering such issues, the time to market aspect justifies the risks. In commerce, time to market (TTM) is the length of time that a product takes from its conception until its delivery for sale. TTM is important in industries where products are outmoded quickly. A common assumption is that TTM matters most for first-of-a-kind products, but actually *the leader often has the advantage of time, while the clock is clearly running for the followers* [2].

Based on this idea, if a software application is ready or almost ready at the release moment of its target OS, then this application will certainly have advantages in the market. This was the main motivation of company X in going for the development of a new application to Windows 8 before its final release. This paper discusses some facts and lessons learned from this experience, so that they can support other teams that intend to go for such kind of development.

The remainder of this paper is structured as follows. Section II specifies a simple decision framework that illustrates the variables that must be considered before starting an application development for a pre-release OS. Section III summarizes the features of both our initial scenario and application that was developed. Furthermore, the development cycles are described, focusing on important decisions and metrics collected during this process. Finally, Section IV analyses the collected information, while Section V concludes this paper with the main learned lessons.

## II. DECISION FRAMEWORK

As discussed in Section I, the motivation to face the risks of an application development for a pre-release OS is associated with the TTM aspect. Figure 1 illustrates this idea. According to this schema, two strategies for developing an application could be carried out: *A* and *B*. The strategy *A* approximately starts at same time when the operating system is pre-released. Differently, the strategy *B* starts approximately when the final version of the OS is released. The development time of strategies *A* and *B* are respectively given by  $t_A$  and  $t_B$ . The time  $t_{adj}$  represents the time required to adjust the application *A* to the OS final version.

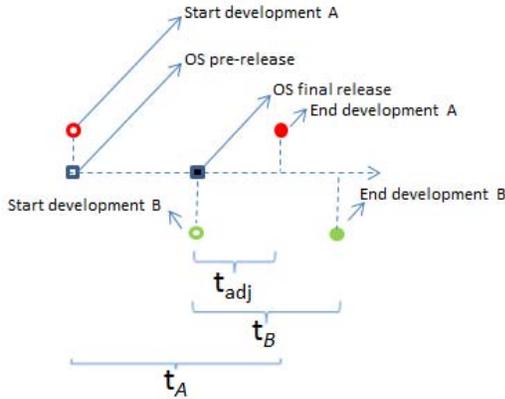


Figure 1. Decision framework schema.

To actually have advantages in using the strategy  $A$ , the next two main propositions must hold:

- $t_B \gg t_{adj}$ , this means,  $t_{adj}$  must be very shorter than  $t_B$ ;
- $t_{adj}$  must be as short as possible, so that  $t_A \approx t_B$ ;

Next section discusses the development process of a new application for Windows 8 and analyses its features considering this framework.

### III. DEVELOPMENT PROCESS

This section starts by characterizing the initial scenario of development. Then, we analyze the development strategy used by the team, considering the framework in Figure 1. This strategy is then presented, together with some metrics that characterize the main problems in this development process. The main idea is to relate such problems with particular aspects of the pre-release OS.

#### A. Characterization of Initial Scenario

The software application of this study intendsto support functions related to *QR codes* [3], such as their creation and decodification. This application was specified according to the Windows 8 Metro style [4], considering the norms and patterns of this platform, such as optional support to touch screen. Some functions of this application are:

- Reading and generation of QR code using the device camera;
- Integration of QR code functions with some native applications, such as Bing Maps and Calendar;
- Support for several languages.

The development unit of company X, at this time, had about 50 members, who were involved in Android and J2ME (Java Micro Edition) implementations. This means, the team members did not have experience in the Windows platform. Thus, a completely new team was created with 13 members, with experience in Windows development, composed of: 1 Scrum Master, 1 senior software engineer, 5 software engineers II, 4 software engineers I (juniors) and 1 internal. None of the members had experience with the Windows 8 platform, once this OS had been just pre- released.

The Agile methodology SCRUM [5] was used in this project and its concepts were used to lead the execution of the project cycles. The functional requirements were defined

during the planning stage and the goals and deliverable of each stage were defined and ordered according to priorities defined by development and business teams. The project started at February 2012 and finished at October 2012.

#### B. Development Strategy

The main challenge of this QR Code application was to finish the project in a hard deadline, which should be close to the OS and related devices (tablets) final releases, thus reducing  $t_{adj}$ . In this context, a back forwarding planning was carried out, considering this hard deadline. Considering the use of Scrum, the next steps were carried out:

- Definition of a preliminary report about the application requirements;
- The requirements are mapped into stories. Each story is analyzed by the team in terms of required effort, so that they could be allocated into sprints. A total of 7 sprints were initially estimated;
- The business team organizes the stories, according to their business value, in a sequence of implementations. Thus, less important stories are allocated in final sprints;
- The beginning of each sprint defines how many stories will be considered. At this moment, the priorities can be modified. However, during the sprint (each sprint was specified to 10 work days), the priority must be maintained;
- At the end of each sprint, the business and development teams review the results of the sprint. If some part of the sprint is not part of the final package, then the development team must list the related reasons and problems;

At the final of the project, five sprints were actually performed, once two sprints (6 and 7) were excluded to maintain the hard deadline. Thus, all the requirements associated with the last two sprints were also removed. Next section discusses four sprints of this project, stressing the main reasons for delays.

#### C. Characterization of Sprints

Table I shows a summary of the first sprint. The team had specified 10 initial tasks for this sprint. However, new tasks were created to support the implementation of the stories selected to this sprint (Stories 1 and 2).

TABLE I. SUMMARY OF SPRINT 1 REGARDING ITS TASKS

Sprint #1	Done	New	Invalid	Impediment	Remain	Expected
Initial # of tasks	10					
Total tasks	39					
19-03-2012	4	6	0	1	12	35.1
20-03-2012	3	1	0	0	10	31.2
21-03-2012	1	0	0	0	9	27.3
22-03-2012	4	2	0	0	7	23.4
23-03-2012	1	2	0	0	8	19.5
26-03-2012	3	3	0	0	8	15.6
27-03-2012	2	2	0	0	8	11.7
28-03-2012	3	10	0	0	15	7.8
29-03-2012	4	2	1	0	12	3.9
30-03-2012	3	1	0	0	10	0

For example, in the eighth line (28.03.2012), 10 new tasks were created. Considering an average of 3.9 tasks/day, the total of 29 new tasks increased the development time by about 8 days. The graph below (Figure 2) shows the same information from a different perspective. Observe that the *Remain line* is approximately constant and this indicates that the tasks are not been consumed. In fact, at the final of the sprint, 10 tasks remained to be implemented.

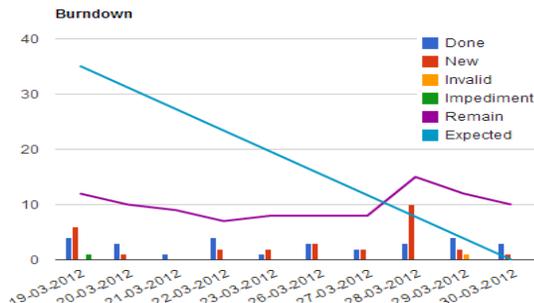


Figure 2. Evolution of tasks over Sprint 1.

The second Sprint planned a more realistic initial number of tasks (Table II), once only 5 new tasks were created. This fact may indicate a better knowledge of the team about the OS and its limitations. The low number of remaining tasks (2 tasks) also corroborates this affirmation. The third Sprint (Table III) had a higher number of new tasks than the previous sprint, however the team was able to control this increase and only 1 task remained to the next sprint.

TABLE II. SUMMARY OF SPRINT 2 REGARDING ITS TASKS

Sprint #2							
Initial # of tasks	26	Num of work days (sprint):		10			
Total tasks	31	Task/day:		3.1			
	Done	New	Invalid	Impediment	Remain	Expected	
02-04-2012	1	0	0	0	25	27.9	
03-04-2012	3	1	0	0	23	24.8	
04-04-2012	2	2	1	0	22	21.7	
05-04-2012	3	0	1	0	18	18.6	
06-04-2012	0	0	0	0	18	15.5	
09-04-2012	1	0	0	0	17	12.4	
10-04-2012	2	1	0	0	16	9.3	
11-04-2012	1	0	0	0	15	6.2	
12-04-2012	2	1	0	0	14	3.1	
01-04-2013	12	0	0	0	2	0	

TABLE III. SUMMARY OF SPRINT 3 REGARDING ITS TASKS

Sprint #3						18-04-2012	
Initial # of tasks	17	Num of work days (sprint):		10			
Total tasks	26	Task/day:		2.6			
	Done	New	Invalid	Impediment	Remain	Expected	
04-16-2012	0	0	0	0	17	23.4	
04-17-2012	0	0	0	0	17	20.8	
04-18-2012	2	1	1	0	15	18.2	
04-19-2012	2	4	0	0	17	15.6	
04-20-2012	5	2	0	0	14	13	
04-23-2012	3	1	0	0	12	10.4	
04-24-2012	0	1	1	0	12	7.8	
04-24-2012	0	0	0	0	12	5.2	
04-24-2012	11	0	0	0	1	2.6	
04-24-2012	0	0	0	0	1	0	

Sprint 4 illustrates an interesting situation of unpredicted dependence. Consider a task  $t_1$  and a set of tasks  $T_\Omega$ , where the tasks  $\{t_1, \dots, t_n\} \subseteq T_\Omega$ . Also consider that the

implementation of  $T_\Omega$  depends on the implementation of  $r_1$ . Then, if  $r_1$  becomes invalid, then the set  $T_\Omega$  cannot be implemented. The solution is to create a new task  $t_{new}$  or a set of new tasks  $T_{new}$  that implement an alternative solution. Table IV describes a similar kind of unpredicted dependence. Observe that several tasks are only performed at the end of the sprint, while 5 new tasks are created at the first half of this sprint.

TABLE IV. TD CONCEPTS, DEFINITIONS AND REFERENCES

Sprint #4						5/2/2012	
Initial # of tasks	19	Num of work days (sprint):		10			
Total tasks	29	Task/day:		2.9			
	Done	New	Invalid	Impediment	Remain	Expected	
05-02-2012	0	0	0	0	19	26.1	
05-03-2012	0	0	0	0	19	23.2	
05-04-2012	0	1	1	0	19	20.3	
05-07-2012	0	3	0	0	22	17.4	
05-08-2012	0	1	2	0	21	14.5	
05-09-2012	1	0	0	0	20	11.6	
05-10-2012	3	0	2	0	15	8.7	
05-11-2012	10	5	0	0	10	5.8	
05-14-2012	0	0	0	0	10	2.9	
05-15-2012	0	0	0	0	10	0	

This situation is illustrated by the unusual graph below (Figure 3), where the line that represents the remainder tasks has a considerable increase before starting its normal decreasing behavior.

We classify this situation as unusual because when tasks are specified in Scrum, they should be implementable considering the current state of the system. Thus, it is uncommon that a sprint starts with the creation of new tasks. All these problems resulted in a remainder number of 10 tasks at the end of this sprint.

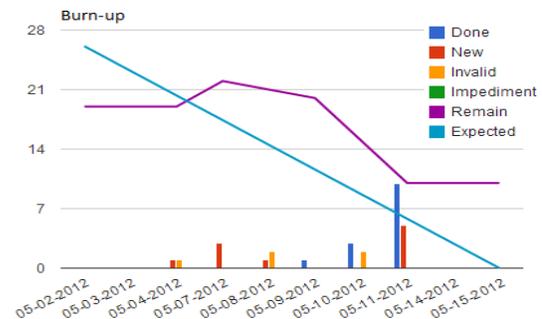


Figure 3. Evolution of tasks over Sprint 4.

At this point is important to consider the semantic of *Invalid* tasks. Such kind of task is an activity that generated effort, however without any kind of contribution to the application. For example, after some study or implementation, developers conclude that the implementation of such task is not possible. Thus, this situation is an important source of delays.

#### IV. INFORMATION ANALYSIS

In this section we try to relate some problems, identified in the previous section, to aspects of a pre-release SO. These problems are illustrated by the next observations:

- $O_1$ : pre-release OS aspects account for the high number of new tasks in Sprint 1;

- $O_2$ : pre-release OS aspects account for the behavior of the *Burn-up* graph (Figure 3) in Sprint 4;
- $O_3$ : pre-release OS aspects account for one or more invalid tasks;

For this analysis, we have used other development documents, exchanged emails and interviews with members of the development team.

#### A. Analysis of Observation 1 ( $O_1$ )

According to the project leaders, as the OS was in ongoing development, the lack of formal documentations led the team to spend a considerable time in investigating the technology using, for example, informal sites and forums. A communication channel with Microsoft was also established. Due to their complexity, all these efforts were transformed into tasks. In a normal Scrum project, the study required by a task is part of the own task, once this study does not generally require so much effort.

Another important fact is related to the task concept of "Done". In Scrum, when a task is qualified as "Done", it cannot return to any other state. Several tasks that were finished in this project presented integration problems. Then, new tasks were created to complete such "finished" tasks. The principal problem in this scenario was the limited knowledge about the technology and some of its application program interfaces (e.g. API about camera manipulation). Such interfaces did not present the expected behavior, affecting the process of integration.

#### B. Analysis of Observation 2 ( $O_2$ )

The main problem of Sprint 4 was the use of incomplete APIs, such as the API of contact manipulation. This incompleteness was only identified during the studies that were part of the Sprint 4. Thus, the team was not able to implement several functions initially defined and the first half of this sprint was used to create the basis for implementing some of these functions. This sprint in fact generated a considerable extra work, so that the initial scope of this project was reduced.

#### C. Analysis of Observation 3 ( $O_3$ )

According to the interviews, some APIs, such as the Windows API C#, had significant modifications until the final OS release. These modifications were the reason for invalidating several tasks. Furthermore, tasks planned to carry out unitary tests in functions related to user interfaces in Sprints 1, 2 and 3 were also invalidated because this part of the OS was not working. When the user interface API was released, it presented changes and the unitary tests had to be adapted. Thus, the team had a considerable waste of time.

### V. CONCLUSIONS

From this experience, we can stress three main learned lessons that may assist other groups that intend to join this kind of development. First, the team needs to define alternatives to the lack of document/information. For example, they could establish a communication channel with the producers that they depend on. Second, it is important to

avoid as much as possible APIs in a very early stage of development, once they are prone to be modified. In fact, this feature (API maturity) could be considered during the prioritization of requirements. In this case, requirements that depend on early APIs will be left to latter sprints and, at this time, such APIs may be in a more mature stage. Finally, as the unexpected events are more common when we deal with pre-release systems, project managers should define estimations that consider a higher frequency of such events. To illustrate this aspect, let's calculate the total of work days that were spent with new and invalid tasks (Table V).

TABLE V. ESTIMATION OF TIME USED IN NEW AND INVALID TASKS

Sprint	New tasks	Inv. tasks	Tasks/days	Days
1	29	1	3,9	10,00
2	5	2	3,1	2,26
3	9	2	2,6	4,23
4	10	5	2,9	5,17
Total:				21,66

This table stresses that about half of the time (21,66 from 40 days) was spent with unexpected events (new and invalid tasks). Another interesting information is that these "extra 21,66" days could be enough to implement the two removed sprints (Sprints 6 and 7). Finally, this software was probably one of the first QR code related applications to run in this new OS for tablets. Thus, even considering the reduction of the functional scope (7 to 5 sprints), this effort to develop for a pre-release OS version certainly brought a differential in the competitive market of mobiles.

#### ACKNOWLEDGMENT

The results presented in this paper have been developed as part of a collaborative project between Samsung Institute for Development of Informatics (Samsung/SIDI) and Federal University of Pernambuco (CIn/UFPE), financed by Samsung Eletronica da Amazonia Ltda., under the auspices of the Brazilian Federal Law of Informatics no. 8248/91. Professor Fabio Q. B. da Silva holds a research grant from the Brazilian National Research Council (CNPq), process #314523/2009-0. This work was also supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq, grants 573964/2008-4.

#### REFERENCES

- [1] M. A. Cusumano, D. B. Yoffie, "Software development on Internet time", *Computer*, October 1999, 32(10): 60-69.
- [2] L. Wu, R. Matta, T. J. Lowe, "Updating a modular product: How to set time to market and component quality", *IEEE Transactions on Engineering Management*, May 2009, 56(2): 298-311.
- [3] M. W. Islam, S. Alzahir, "A novel QR code guided image stenographic technique", 2013 IEEE International Conference on Consumer Electronics (ICCE), January 2013.
- [4] S. E. Arnold, "Windows Metro: User Experience Over Findability", *Information Today*, December 2011, 28(11): 20-21.
- [5] K. Schwaber, J. Sutherland, "Scrum Guide", 2013, available in [www.scrum.org](http://www.scrum.org)

# Analysis of Multi-Dimensional Code Couplings

Fabian Beck

VISUS, University of Stuttgart, Germany

Email: fabian.beck@visus.uni-stuttgart.de

**Abstract**—Software systems consist of hundreds or thousands of files, which are usually not independent of each other but coupled. While it is obvious that structural dependencies like method calls or data accesses create couplings, there also exist other, more indirect forms of coupling that should be considered when modifying, extending, or debugging a system. In contrast to most previous research, in this work, code coupling is considered as a multi-dimensional construct: several forms of structural couplings are contrasted to couplings based on the history and the semantics of the source code entities. The work proposes two novel visualization techniques, which allow for exploring and visually comparing different concepts of coupling. Based on an empirical study on open source systems, the work further provides insights into the relationship between concepts of coupling and the modularization of software; first evidence on the usage of modularization principles can be derived thereof. Finally, a new application for adapting the modularization of a software system—component extraction—is introduced and tested with varying coupling data. This work summarizes the doctoral thesis of the author, suggests directions for future research, and reports lessons learned.

## I. INTRODUCTION

*If changing one module in a program requires changing another module, then coupling exists.*

—Martin Fowler, 2001 [1]

Defining coupling by those hypothetical changes, multi-faceted reasons exist why entities of a software system need to be changed together—coupling is indicated by different observable features of the system:

- **Structure:** Changing the functionality or interface of a code entity may induce changing another because the code depends on functionality defined somewhere else.
- **History:** The history of a software project might reveal hidden dependencies: code entities changed together in the past are likely to still change together in the future.
- **Semantics:** When adapting the specification of a software system, all entities that relate to the modified semantics need to be changed, which are not necessarily connected by structure or history.

These observable connections between entities of code are called *code couplings* in this work. The term serves as a superordinate to other terms often used in a similar context, such as *code dependencies*, *relationships*, *associations*, *links*, *similarities*, or *connections*. Each form of observable relationship introduces a different *concept of code coupling*.

Usually, coupling is not treated as a multi-dimensional but one-dimensional construct, often consisting of only structural

dependencies. There exist indeed some works that analyze different concepts of coupling, but more on an exemplary level than applying and comparing a larger number of concepts systematically. In contrast, this work is considering code coupling as a multi-dimensional construct and aims at providing the means for systematically analyzing these couplings. Hence, the central goal of this work is to provide a better understanding of different concepts of code coupling in particular context of modularity.

This work summarizes and reflects the doctoral thesis of the author [2], which was conducted under the supervision of Stephan Diehl at the University of Trier, Germany between February 2009 and February 2013. Stephan Diehl and Martin Pinzger (University of Klagenfurt, Austria) acted as examiners for the thesis. The work was part of the DFG project “Analyse mehrdimensionaler Kopplung zur Unterstützung des Software-Entwicklungsprozesses” (grant number: DI 728/8-1/2). The remainder of this papers presents a synopsis of the results of the thesis (Section II), discusses directions for future research (Section III), and reports some lessons learned (Section IV).

## II. SYNOPSIS

Formally, code couplings are modeled as a graph structure consisting of code entities as nodes and couplings as weighted, directed edges. Multiple concepts of coupling are treated as different types of edges. The modularization of a system additionally introduces a hierarchy on the nodes. The final data model is a construct called *modularized multi-dimensional coupling graph*, which aggregates, for a single software system, all data relevant in this work. Focusing on object-oriented Java systems, interfaces and classes form the code entities and the package structure provides a hierarchical organization.

The 17 concepts of coupling considered in this work are divided by three categories: *structure*, *history*, and *semantics*. Goal for selecting specific concepts was to cover a broad spectrum of relevant concepts that also could be retrieved with acceptable effort. The specific concepts are listed in Table I: First, the structural couplings of the system are covered by direct inheritance dependencies (*SD.Inh*), aggregation dependencies (*SD.Agg*), and usage dependencies (*SD.Use*); further structural dependencies are considered by indirect variants of these three concepts—code entities are coupled if they depend on the same other project-external (*FO.InhE*, *FO.AggE*, *FO.UseE*) or project-internal (*FO.InhI*, *FO.AggI*, *FO.UseI*) code entities. Second, history is reflected in two variants of evolutionary coupling recording co-changing code entities in the past (*EC.Sup*, *EC.Conf*), and two variants of code ownership coupling connecting the files having similar authors (*CO.Bin*, *CO.Prop*). Third, semantic relations in code are

TABLE I. CONSIDERED CONCEPTS OF COUPLING.

Group	Subgroup	Concepts of Coupling
structure	structural dependencies	<i>SD.Inh, SD.Agg, SD.Use</i>
	fan-out (external)	<i>FO.InhE, FO.AggE, FO.UseE</i>
	fan-out (internal)	<i>FO.InhI, FO.AggI, FO.UseI</i>
history	evolutionary coupling	<i>EC.Sup, EC.Conf</i>
	code ownership	<i>CO.Bin, CO.Prop</i>
semantics	code clones	<i>CC.I, CC.II</i>
	semantic similarity	<i>SS.Tfidf, SS.LSI</i>

TABLE II. OUTLINE OF THE THESIS AND PRIOR PUBLICATIONS.

Part	Objective	Publication
(Feasibility Study)		[3], [4]
A. Visual Comparison of Code Couplings	Visualize different concepts of code coupling in modular software systems.	
1) Node-Link Approach		[5], [6]
2) Matrix Approach		[7], [8]
3) Evaluation and Comparison		[9]
B. Empirically Analyzing the Congruence	Analyze the congruence between concepts of coupling and modularity.	[10] <sup>a</sup>
C. Component Extraction	Leverage the multi-dimensionality of code coupling for modularizing software systems.	[11], [12]

<sup>a</sup>awarded with the *Joseph A. Schumpeter-Preis* of the *Fachbereich IV, University of Trier*, 2012

expressed through two variants of coupling based on shared code clones of code entities (*CC.I, CC.II*) and two variants of measuring the similarity of the vocabulary used in source code (*SS.Tfidf, SS.LSI*).

The thesis falls into three main parts addressing different objectives (Table II provides an overview and references prior publication): The first part investigates the scalable visual comparison of code couplings; two novel visualization approaches are proposed for interactively exploring and comparing code couplings (Section II-A). Second, the congruence of coupling structures and modularizations is analyzed in an empirical study in order to investigate how modularity principles are used in practice (Section II-B). Third, it is tried to leverage the multi-dimensionality of coupling for shaping software components that can be extracted for future, independent development (Section II-C). As a starting point, however, a feasibility study, which is not reported in further detail here, was conducted where two concepts of coupling were contrasted and successfully combined in the application of automatically modularizing software systems.

#### A. Visual Comparison of Code Couplings

Making code coupling information explorable through visualization is challenging because software systems are complex and easily contain hundreds of entities and thousands of couplings. The representation and comparison of multiple concepts of coupling further complicates the task. In particular, the combination of both challenges—scalability and multi-dimensionality—had not been studied sufficiently. To tackle this combination of challenges, the thesis presents two novel visualization approaches that focus on different aspects of the problem and are based on two different graph visualization paradigms.

1) *Node-Link Approach*: The first visualization technique works with on juxtaposed node-link diagrams, however, varies significantly from traditional node-link approaches by applying

a special layout: nodes are arranged on linear vertical axes and each node is split into two ports assembling outgoing and incoming edges. This unusual layout targets at fitting scalable node-link diagrams into small stripes that can be arranged side-by-side enabling graph comparison. An icicle plot is attached reflecting the hierarchical structure of the data. Figure 1 (left) provides an impression of this visualization contrasting three direct structural concepts of coupling (*SD.Inh, SD.Agg, SD.Use*) for the *JFtp* project.

The implemented visualization tool allows manipulating the visualization to retrieve details and to ease the visual comparison process. Basic interactions like focusing and highlighting enlarge parts of the visualization or visually distinguish them from others. While the juxtaposed subdiagrams already support a visual comparison of concepts of coupling, interaction provides the possibility to further enhance the comparison abilities: the tool allows for moving subdiagrams by drag-and-drop; additionally, an advanced highlighting technique of entities enables the comparison of graph features and the possibility to merge concepts of coupling by set union, intersection, and difference helps finding common structures or outliers. Some design alternatives are explored for the approach, among them, different edge bundling strategies [13], [14] and edge splatting [6]. Typical visual patterns are discussed, which help interpreting the visualization. A case study suggests how the approach can be used in different practical software engineering scenarios.

2) *Matrix-Based Approach*: The modularization of a software system is visualized along with code couplings in the presented node-link approach. But there are scenarios where more than one modularization of a system exist: alternative modularizations could be different versions of the system, modularizations created by applying other modularization criteria, or modularizations generated by clustering. Though approaches for visually comparing multiple hierarchies such as modularizations are available [13], [15], these approaches are not suitable for concurrently visualizing code couplings. Due to the need for comparing hierarchies, this approach is visually very different to the previously presented node-link visualization technique; it is based on an adjacency matrix representation of the coupling graphs instead. The matrix is augmented with two different hierarchies, attached to the sides of the matrix as icicle plots. Grayscale rectangles in the background of the matrix encode the similarity of modules between the two hierarchies. The resulting visualization is shown in Figure 1 (right), again visualizing the *JFtp* project: the package structure on the left is contrasted to a clustered modularization at the top; further, two concepts of coupling are visualized within the matrix using different colors.

Like the node-link approach, the introduced matrix approach is implemented as an interactive visualization tool. Besides basic interactions like zooming and selecting, the level of detail for the two modularizations can be adapted interactively: by expanding and collapsing modules in the two hierarchies matching parts of the hierarchy can be retrieved. While it might be acceptable to manually find such similarities small projects such as *JFtp*, the task becomes tedious for larger projects. Hence, an automatic algorithms is proposed that automatically finds matching levels of detail. To further increase the readability of the visualization, an ordering al-

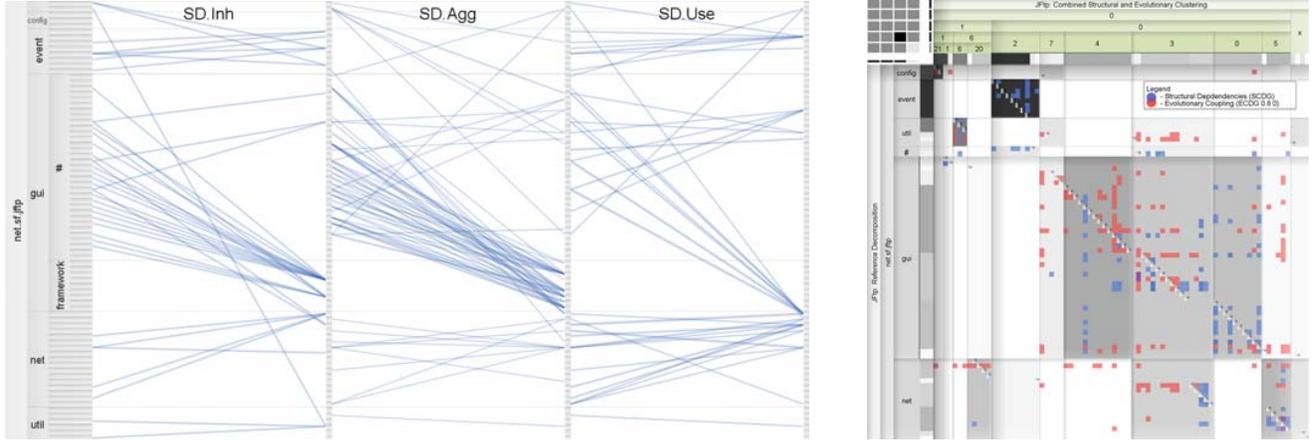


Fig. 1. Two novel visualization techniques for understanding multi-dimensional code couplings; left: node-link visualization for the comparison of concepts of couplings; right: matrix-based visualization for comparing two modularizations along with two concepts of coupling; both depict the JFtp project.

gorithm of modules and code entities is introduced that sorts equivalent entities and similar modules to the diagonal of the matrix. Level-of-detail and sorting algorithms were evaluated with respect to layout and interaction improvements based on metrics and examples. A case study provides first evidence on the usefulness of the approach for exploring and restructuring software systems.

3) *Evaluation and Comparison*: Additional to the case studies, a user study was conducted for exploring how software developers compare different concepts of coupling with the help of visualizations. In particular, the node-link approach presented in this work was contrasted to a matrix approach by Abuthawabeh and Zeckzer [16], which is similar to the matrix approach introduced above—directly comparing the two visualization approaches of the thesis was not possible due to their different focus and abilities. With the help of the visualizations, eight developers analyzed software projects in a realistic scenario based on an explorative study design. A largely qualitative analysis of the results showed that both visualizations can be used for the same program comprehension tasks such as finding key entities, understanding a package, or identifying high-level couplings. Visual comparison of concepts code couplings was integral part of these tasks.

### B. Empirically Analyzing the Congruence

In their general theory on modularity, Baldwin and Clark [17] argue that complex systems, such as software systems, need to be modularized to make their complexity manageable. In research literature, multiple desirable attributes are connected to a good modularization of a software system, for instance, *comprehensibility* [17], [18], *abstraction* [17], *changeability* [19], [18], *independent development* [17], [18], *reusability* [19], or *flexibility* [17]. In order to modularize a system, software developers group those code entities that are similar, connected, dependent, related, or in other words *coupled* by any concept of coupling. Hence, there is a multitude of criteria for modularization: for instance, those code entities are placed in the same module that structurally depend on each other (*low coupling and high cohesion*) [20], that are connected to the same design decisions (*information hiding*) [18], or that

TABLE III. CONJECTURED RELATIONSHIPS BETWEEN MODULARITY PRINCIPLES AND CONCEPTS OF COUPLING; MEASURED CONGRUENCE OF COUPLING AND MODULARITY IN THE EXPERIMENT.

Modularity Principle	Concepts of Coupling	Rationale	Congruence
<i>low coupling and high cohesion</i> [20]	SD.Inh, SD.Agg, SD.Use	directly addressing direct structural dependencies	high
<i>information hiding</i> [18]	FO.Inh, FO.Agg, FO.UseI	depending on the same design decision (hidden in other code entities)	high
	EC.Sup, EC.Conf	locality of co-changes; past changes predict future changes	high
<i>Conway's law</i> [21]	CO.Bin, CO.Prop	organization structure similar to product (modularization) $\Rightarrow$ ownership locality	low
<i>domain knowledge</i> [22]	SS.TIdf, SS.LSI	domain-related terms are reflected in the vocabulary	medium
<i>separation of concerns</i> [23]	FO.InhE, FO.AggE, FO.UseE	usage of the same functionality	low–medium
[not classified]	CC.I, CC.II	multiple aspects	medium–high

are related by the communication structure of the development team (*Conway's law*) [21].

These modularity principles, more or less, are connected to couplings between code entities. Abstracting from the concrete principles, in each case, pairs of coupled entities should be placed into the same module while pairs of non-coupled entities can be spread across different modules. This implies that, in a good modularization, there exists certain congruence between code couplings and modularity: couplings are assumed to mostly connect entities in the same module rather than in different modules. Different modularity principles can be mapped to different concepts of code coupling as summarized in Table III. For instance, the principle of information hiding assumes the locality of co-changes, which is reflected in a high congruence between evolutionary couplings and the existing modularization of the system.

Although the conjectured relationships are different and are founded on diverse rationale, they share a specific characteristic: assumed that the relationships are correct, the particular modularity principle is met if the code couplings of the respective concept are largely *local* with respect to modularity. *Local*, in this context, means that the couplings mainly connect code entities contained in the same or similar modules instead of entities from very different modules. The *coupling-modularity congruence* is introduced as a measure for empirically in-

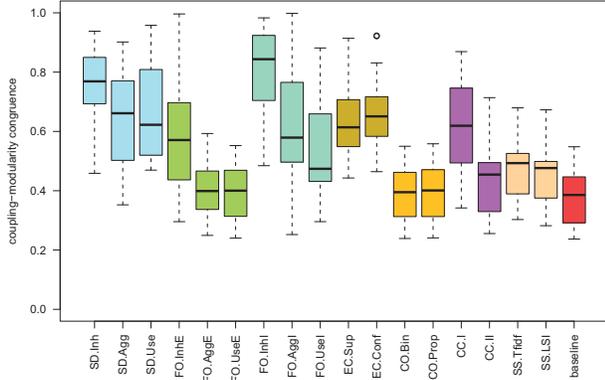


Fig. 2. Boxplot diagram depicting the coupling-modularity congruence for different concepts of coupling across all studied projects.

vestigating this locality. It quantifies the conformance of the respective coupling structure to the modularization, hence, compares a graph structure to a hierarchy. Based on module cohesion and coupling, the congruence between the coupling structure and the modularity of the system is defined: while a high overall module cohesion indicates a high congruence, a high module coupling hints at a lower congruence; in broad strokes, the measure divides module cohesion by the sum of module cohesion and coupling.

In an empirical study based on 16 open source Java projects, the congruence is investigated step by step. First, by showing that the concepts of coupling are sufficiently independent of each other (except variants of the same underlying concept), it is substantiated that the concepts really cover different aspects of coupling. Second, the concepts of coupling show quite different congruence to the modularizations of the analyzed systems. As depicted in Figure 2, direct and indirect structural dependencies, evolutionary couplings, and code clone couplings provide high congruence values; the highest values are reached by fan-out inheritance similarity (*FO.InhI*) and direct structural inheritance dependencies (*SD.Inh*). Finally, no distinguished variance is observed for the congruence patterns of different types of modules such as *gui*, *util*, or *io* packages. Applying the mapping of concepts of code couplings and modularity principles, the results hint at a particular impact of the principle of *low coupling and high cohesion* and of *information hiding* (Table III, Congruence).

### C. Component Extraction

Code couplings are used for automatically clustering software systems [24]. Many software clustering approaches, however, share the problems that they ignore existing modularization, change every part of the system ruthlessly, and are not interactive [25], [26]. Considering these issues, a novel modularization approach is proposed that focuses on the semi-automatic extraction of a single component instead of modularizing the whole project fully automatically. The target is enabling future independent development of the extracted component. Similar as software clustering, component extraction is closely related to code couplings—future independent development of components requires that components are decoupled. While perfect segregation is the idealistic but unreachable goal,

component extraction should at least try to minimize coupling. A related technique is component extraction refactoring [27].

As a starting point, it is assumed that the developers already have some rough idea which part of the software they want to extract: they identify two sets of key entities (i.e., classes and interfaces), one providing seeds for the extracted component, the other specifying the parts of the program that should not be extracted. A component extraction algorithm computes a minimal cut in the coupling graph between the two sets. The result is the basis for proposing a component to extract and for creating a so-called *contract*, which acts as an interface between the component and the remaining system. Being an interactive approach, users may modify the set of key entities based on the proposed result and rerun the algorithm.

Interacting with the component extraction result requires an intuitive and scalable user interface that represents the proposed decomposition and relevant couplings. Two versions were developed: the first prototype was evaluated in small user study; based on the experience from this study, an improved interface was designed as an IDE plug-in (Figure 3). The two user interfaces mainly differ in the approach used for representing couplings between the components: while the prototype employed explicit visual links between entities, the IDE plug-in works with aggregated lists of couplings to enhance scalability. Both interfaces share a split main view with three columns—the original system on the left, the extracted component on the right, and the contract in between.

Finally, it is evaluated what concepts of coupling are most suitable for component extraction and how they might be combined to further improve the results. The component extraction algorithm is tested with multiple concepts of coupling similar to the empirical study on modularity (Section II-B). The observed results are largely consistent to the results reported for the congruence between the concepts of coupling and modularity: *SD.Inh* and *FO.InhI* perform best, followed by other concepts of direct structural coupling and evolutionary coupling. Surprisingly, however, the merging of concepts does not improve the results of the approach reasonably, but acts more as a way to average the outcome.

### III. FUTURE RESEARCH CHALLENGES

The described work considers code coupling as a multi-dimensional construct. This shift from one-dimensional to multi-dimensional couplings opens up a wealth of questions—only a few of them have been answered. A particular focus has been on how multi-dimensional couplings interact with modularity. However, other applications than modularization have stayed untouched, such as change impact analysis or bug detection where code couplings seem to have a similarly important role. Understanding couplings as multi-dimensional will probably also have an impact on these domains. While this is a general observation, more specific research challenges are the following.

**RC1: The Meaning of Coupling**—How to define coupling (or a related term) so that the definition matches the intuition of developers, is measurable, and reflects multi-dimensionality?

The definition by Fowler [1] used in this work is based on potential future co-changes (Section I). It is open and takes the

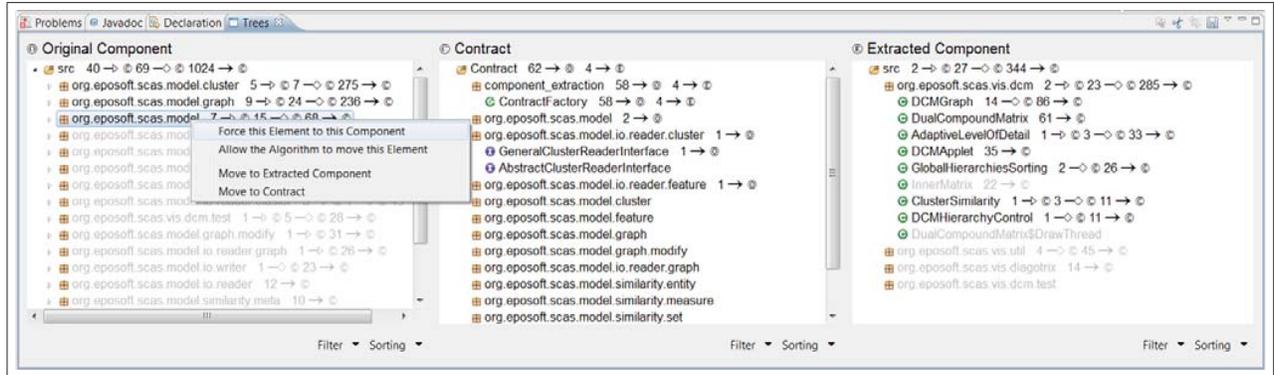


Fig. 3. Main view of the improved interface showing the extraction results for a sample project.

perspectives of developers but is difficult to map to a metric. Considering multiple concepts of couplings is possible but the selection of concepts lacks empirical evidence. In a recent study, Bavota et al. [28] partly address the above research challenge: they investigated which concept of coupling is most related to the intuition of developers. Pairs of code entities, which are (not) coupled with respect to a certain concept, were presented to developers, who rated the strength of coupling. The results suggests that, in particular, semantic similarity and structural dependencies match well the developers' idea of coupling. While this is an interesting first step, more research need to be conducted in this direction.

**RC2: Combination of Code Couplings**—How to combine multi-dimensional couplings to a single concept of coupling?

A related question to finding a consistent and measurable definition of coupling is how to combine several concepts. While dividing couplings into several concept might be appropriate in a first step, most algorithmic approaches processing couplings require a single concept. Since only using one of the concepts means throwing away much information, it is desirable to combine different data sources. In this work, simple approaches were tested based on the union of couplings: while this was successful in the feasibility study for the application of software clustering with two concepts of coupling, a similar approach for component extraction combining more concepts of coupling showed only limited success. In general, a multitude of ways exist how to combine several concepts of coupling [4]; a systematic evaluation which works best with respect to different applications has not yet been conducted.

**RC3: Interactive Software Modularization**—How to design a general recommendation tool for modularizing software systems based on multidimensional code couplings?

Component extraction was only a first try for leveraging multi-dimensional couplings in practice. It focused on a special use case of modularizing a software systems; combining concepts of coupling was only partly successful. An open research question still is how the advantages of component extraction—in particular the interactive and transparent approach—can be transferred to general software clustering. While some works already propose solutions into this direction [29], [30], using multi-dimensional couplings in such an approach is still a challenge. Multiple concepts of coupling might help better explaining developers proposed remodularizations.

#### IV. LESSONS LEARNED

Writing a doctoral dissertation, no matter how fascinating, is a long and exhausting process. In this section, I report some lessons learned divided into issues related to software engineering research (SE) and general observations on writing a doctoral thesis (DT).

**Lesson SE1:** *Changing a software system has to be controlled by the developer.*

Coming up with a good solution of a software engineering problem is often not enough: developers will not apply the proposed changes if they do not understand why they should and are not in control of the process. Taking software clustering as an example, dozens of approaches exist, but they seem to be rarely applied by developers because they work automatically [25]. While I first also experimented with software clustering in the feasibility study (see Table II), by now, I believe that only user-controlled, semi-automatic approaches like component extraction (Section II-C) or the interactive modularization approach outlined as future work (RC3) have the chance to be accepted by developers.

**Lesson SE2:** *There is not much fundamental research and theory on software engineering.*

Software engineering research seems to be very practical: hardly any paper does not present a specific application of the results. Searching for literature on modularity principles, I found mostly works from the 1970s (Section II-B), but relatively few follow-up papers with a comparable focus on theory and fundamentals. Only recently, people within the field start to rediscover this kind of research (e.g., [10], [28]).

**Lesson DT1:** *Starting to write early helps keeping focused.*

Just to take some notes on related work, I started to compile a document for the thesis early (in the first year). In retrospective, I have been lucky: working regularly with the document to extend my notes, I unwillingly thought about the structure of the thesis, the story I wanted to tell, and what work currently was most urgent. This kept me from losing focus during the four years of my dissertation.

**Lesson DT2:** *Scale introduces new complexity.*

When I began compiling the main parts of the thesis, through publishing papers, I already had some experience

on writing scientific texts. But I learned that it is not just more work to write a longer text such as a thesis, but it introduces new complexity: it is much harder to use consistent terminology and telling a story is getting multifaceted.

## V. CONCLUSION

The summarized thesis analyzes *multi-dimensional code couplings* in software projects, i.e., different *concepts of code coupling* derived from multiple data sources. The goal was taking the next step in understanding these couplings by providing means for exploration, by presenting empirical findings, and by proposing approaches for improving the modularization of software projects. While the results are firstly limited to the studied Java systems and cannot be directly generalized to other systems, this work also provides analysis tools (visualizations and software development tools) as well as methods (metrics, evaluation methods, and algorithms); these can be used for analyzing other datasets and for replicating the results. Distinguishing features of the work are the large set of concepts considered for code coupling and the focus on fundamentals instead of concrete applications scenarios.

The thesis considers code coupling as a multi-dimensional construct rather than a one-dimensional one. While, occasionally, a small set of concepts of coupling have been compared or combined in a particular application, this work is one of the first that systematically investigate the idea of multi-dimensional code couplings. For using the full potential of multi-dimensional code couplings, an open research question, however, still is how to best combine the different concepts. In general, the work can be considered as a first step towards the ambitious goal of understanding multi-dimensional code couplings or—to phrase it in the words of Fowler—how *changing one module in a program requires changing another module*.

## REFERENCES

- [1] M. Fowler, “Reducing Coupling,” *IEEE Software*, vol. 18, no. 4, pp. 102–104, 2001.
- [2] F. Beck, “Understanding Multi-Dimensional Code Couplings,” Ph.D. dissertation, University of Trier, 2013.
- [3] F. Beck and S. Diehl, “Evaluating the Impact of Software Evolution on Software Clustering,” in *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 99–108.
- [4] —, “On the Impact of Software Evolution on Software Clustering,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, 2013.
- [5] F. Beck, R. Petkov, and S. Diehl, “Visually Exploring Multi-Dimensional Code Couplings,” in *VISSOFT '11: Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2011, pp. 1–8.
- [6] M. Burch, C. Vehlou, F. Beck, S. Diehl, and D. Weiskopf, “Parallel Edge Splatting for Scalable Dynamic Graph Visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2344–2353, 2011.
- [7] F. Beck and S. Diehl, “Visual Comparison of Software Architectures,” in *SofVis '10: Proceedings of the ACM 2010 Symposium on Software Visualization*. ACM, 2010, pp. 183–192.
- [8] —, “Visual comparison of software architectures,” *Information Visualization*, vol. 12, no. 2, pp. 178–199, 2013.
- [9] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl, “Finding Structures in Multi-Type Code Couplings with Node-Link and Matrix Visualizations,” in *VISSOFT '13: Proceedings of the 1st IEEE Working Conference on Software Visualization*. IEEE Computer Society, 2013.
- [10] F. Beck and S. Diehl, “On the Congruence of Modularity and Code Coupling,” in *SIGSOFT/FSE '11 and ESEC '11: Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*. ACM, 2011, pp. 354–364.
- [11] A. Marx, F. Beck, and S. Diehl, “Computer-Aided Extraction of Software Components,” in *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 183–192.
- [12] F. Beck, A. Pavel, and S. Diehl, “Interaktive Extraktion von Software-Komponenten,” *Softwaretechnik-Trends*, vol. 32, no. 2, pp. 1–2, 2012.
- [13] D. Holten and J. J. van Wijk, “Visual Comparison of Hierarchically Organized Data,” *Computer Graphics Forum*, vol. 27, no. 3, pp. 759–766, 2008.
- [14] F. Beck, M. Puppe, P. Braun, M. Burch, and S. Diehl, “Edge Bundling without Reducing the Source to Target Traceability,” *InfoVis '11 Poster*, 2011.
- [15] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou, “TreeJuxtaposer: Scalable Tree Comparison using Focus+Context with Guaranteed Visibility,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 453–462, 2003.
- [16] A. Abuthawabeh and D. Zeckzer, “IMMV: An Interactive Multi-Matrix Visualization for Program Comprehension,” in *VISSOFT '13: Proceedings of the 1st IEEE Working Conference on Software Visualization*. IEEE Computer Society, 2013.
- [17] C. Y. Baldwin and K. B. Clark, *Design Rules, Vol. 1: The Power of Modularity*, 1st ed. The MIT Press, 2000.
- [18] D. L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [19] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 1st ed. Prentice Hall, 2002.
- [20] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured Design,” *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.
- [21] M. Conway, “How do Committees Invent?” *Datamation Journal*, vol. 14, no. 4, pp. 28–31, 1968.
- [22] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Prentice-Hall, 1988.
- [23] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns,” in *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*. ACM, 1999, pp. 107–119.
- [24] O. Maqbool and H. A. Babri, “Hierarchical Clustering for Software Architecture Recovery,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.
- [25] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, “A Reverse Engineering Approach to Subsystem Structure Identification,” *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.
- [26] M. Glorie, A. Zaidman, A. van Deursen, and L. Hofland, “Splitting a Large Software Repository for Easing Future Software Evolution—An Industrial Experience Report,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 113–141, 2009.
- [27] H. Washizaki and Y. Fukazawa, “Automated Extract Component Refactoring,” in *Extreme Programming and Agile Processes in Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2675, ch. 42, p. 1016.
- [28] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “An Empirical Study on the Developers’ Perception of Software Coupling,” in *ICSE '13: Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*. IEEE Press, 2013, pp. 692–701.
- [29] M. Hall, N. Walkinshaw, and P. McMinn, “Supervised Software Modularisation,” in *ICSM '12: Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 472–481.
- [30] R. Koschke, “An Incremental Semi-Automatic Method for Component Recovery,” in *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE, 1999, pp. 256–267.

# How Good are Code Smells for Evaluating Software Maintainability? - Results from a Comparative Case Study -

Aiko Yamashita

Mesan AS &

Simula Research Laboratory, Norway

Email: aiko@simula.no

**Abstract**—An advantage of code smells over traditional software measures is that the former are associated with an explicit set of refactorings to improve the existing design. Past research on code smells has emphasized the formalization and automated detection of code smells, but much less has been done to empirically investigate how good are code smells for evaluating software maintainability. This paper presents a summary of the findings in the thesis by Yamashita [1], which aimed at investigating the strengths and limitations of code smells for evaluating software maintainability. The study conducted comprised an outsourced maintenance project involving four Java web systems with equivalent functionality but dissimilar implementation, six software professionals, and two software companies. A main result from the study is that the usefulness of code smells differs according to the granularity level (e.g., whether the assessment is done at file or system level) and the particular operationalization of maintainability (e.g., maintainability can be measured via maintenance effort, or problems encountered during maintenance, etc). This paper summarises the most relevant findings from the thesis, discusses a series of lessons learned from conducting this study, and discusses avenues for new research in the area of code smells.

**Keywords**- Code smells; Bad smells; Empirical study; Comparative Case Study, Software maintenance; Software quality.

## I. INTRODUCTION

Code smells are indicators of software design shortcomings that can potentially decrease software maintainability. An advantage of code smells over traditional software measures (such as maintainability index or cyclomatic complexity) is that code smells are associated with an explicit set of refactoring strategies. Thus, code smells can potentially be used to support both *assessment* and *improvement* of software maintainability.

Nevertheless, it is not clear how and to which extent code smells can reflect or describe how maintainable a system is. This makes the interpretation and use of code smells somewhat difficult and hinders the possibility of conducting cost-effective refactoring. Given that refactoring represents a certain level of risk (e.g., introduction of defects) and cost (e.g., time spent by developers modifying the code and cost of regression testing), it is essential to weigh the effort and risks of eliminating versus ignoring the presence of code smells. Furthermore, it is important to understand which maintenance aspects can be addressed by code smells and which should be addressed by other means.

Insufficient information on maintenance aspects, such as severity levels and the range of effects of code smells, makes refactoring prioritization a nontrivial task. To support cost-effective refactoring, we need to increase our understanding of how code smells affect maintenance, what kinds of difficulties they cause, and how they can affect productivity in a project. If we are to use code smells to assess (and improve) maintainability, we also need to understand better *when* they are useful (e.g., the contexts in which their “predictive power” is acceptable) in predicting maintainability, and *which* aspects of maintainability they can measure best. The thesis summarized in this paper investigates how good code smells are in: (1) Reflecting system-level maintainability of software (and how they compare to other system-level assessment approaches), (2) Identifying source code files that are likely to require more maintenance effort (time) than others, (3) Discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance, and (4) Reflecting maintainability aspects that are as deemed critical by software developers. In order to answer these questions, a Comparative Case Study was conducted. The study consisted of outsourcing and observing a maintenance project involving 4 Java web systems, 6 developers and 2 software companies during seven weeks. The remainder of this paper describes how this study was conducted, what were the findings, and discusses some lessons learned from this experience. More specifically: Section 2 describes the methodology followed in the thesis, Section 3 presents the main findings from this research, and discusses some lessons learned from this experience, and Section 4 discusses potential avenues for research in the area of code smells.

## II. DESCRIPTION OF THE RESEARCH STUDY

To investigate the usefulness of code smells, a maintenance project was carried out and observed as part of a comparative case study. This project involved four Java web systems, six software professionals, and two software companies. The remainder of this section will provide details for each of the elements involved in the study, and the design of the study.

### A. The systems under study

In 2003, Simula Research Laboratory’s Software Engineering department sent out a tender for the development of a web-based information system to keep track of their

empirical studies and resulting scientific publications. Based on the submitted bids, four Norwegian consulting companies were hired to independently develop a version of the system using the same requirements and specifications. The four development projects led to the creation of four systems with the same functionality. We will refer to them as System A, System B, System C, and System D. The systems were primarily developed in Java, and they all have similar three-layered architectures. Although the systems exhibit nearly identical functionality, there were substantial differences in how they were designed and coded. The systems were deployed in 2003 over Simula Research Laboratories' Content Management System (CMS), but in 2007, due to changes in the CMS, it was not longer possible for the systems to remain operational. This provided a realistic setting for a maintenance project based on a real need for adapting and enhancing the systems.

### B. The tasks and the developers

The maintenance project involved three tasks: 1) To modify the systems so they could operate in the new CMS environment of Simula Research Laboratory (i.e., an adaptive maintenance task where a Data layer based on a relational DB was to be replaced with a series of calls to external web services), 2) To modify the authentication system by means of consuming a web service, and 3) To extend the systems with a new functionality, which would allow the users to build customisable reports. Two Eastern European software companies were hired to carry out the maintenance tasks between September and December 2008 at a total cost of 50,000 Euros. Thus, the case study was conducted in a way that resembled, as much as possible, a real-life consultancy project. The developers were recruited from a pool of 65 participants of a previously completed study on programming skill [2], and were evaluated to have sufficient English skills for the purpose of our study.

### C. Study design

Having four functionally equivalent systems with different code enabled the design of a comparative case study, with software maintenance tasks embedded within almost identical *maintenance contexts* and differing in the variable of interest: *code smells*. The design of the study therefore, enables better control over the moderator variables, such as system functionality, tasks, programming skills, and development technology to better observe the relations between *code smells* (our variable of interest) and several dependent variables (i.e., different *maintenance aspects*), in a very similar way to experimental studies. In this study, we conducted both *theoretical* and *literal* replications<sup>1</sup>, by asking each of

<sup>1</sup> In *literal replication*, cases that are similar in relation to certain variable(s) are expected to support the analysis of each and give similar results. When *theoretical replication* is used, the cases that vary on the key variable(s) are expected to have different results [3].

		Developer					
		1	2	3	4	5	6
Round	1	A	B	C	D	C	A
	2	D	A	D	C	B	B

Figure 1. Assignment of systems to developers in the case study.

the six developers to first conduct all tasks in one system and then to repeat the same maintenance tasks on a second system, resulting in 12 observations (six developers  $\times$  two systems). Figure 1 describes the order in which the systems were assigned to each developer. This assignment was done randomly.

### D. Study protocol

First, the developers were given an overview of the tasks (e.g., the motivation for the maintenance tasks and the expected activities). Then they were provided with the specification of the maintenance tasks. When needed, they could discuss the maintenance tasks with a researcher; one was always present at the site during the entire duration of the project. We had daily meetings with the developers where we tracked their progress and the problems they encountered.

Think-aloud sessions were conducted every other day at a random point during the day, and they lasted for 30 minutes. Acceptance tests and individual open interviews, which had a duration of 20-30 minutes, were conducted once all three tasks were completed. In the open-ended interviews, the developers were asked about their opinions of the system, e.g., about their experiences when maintaining it.

Eclipse was used as a development tool along with MySQL<sup>2</sup> and Apache Tomcat<sup>3</sup>. Defects were registered in Trac<sup>4</sup>, and Subversion<sup>5</sup> was used as the versioning system. A plug-in for Eclipse called Mimec [4] was installed on each developer's computer to log all the user actions performed at the graphical user interface (GUI) level with millisecond precision.

### E. Variables observed

Figure 2 describes the moderator variables (those we control in the analysis), the variables of interest (those whose relationships we analyze), and the data sources for the variables. The variables of interest within this study are as follows: (1) *Code smells*: Number of code smells and code smell density (code smells/kLOC). The definition of the smells analyzed is provided in [1]. Two commercial tools were used (Borland Together<sup>®</sup> [5] and InCode [6]) to detect code smells. (2) *Developers' perception of the maintainability of the systems*: This includes subjective and qualitative aspects of maintainability reported by each developer

<sup>2</sup> <http://www.genuitec.com>

<sup>3</sup> <http://tomcat.apache.org>

<sup>4</sup> <http://trac.edgewall.org>

<sup>5</sup> <http://subversion.apache.org>

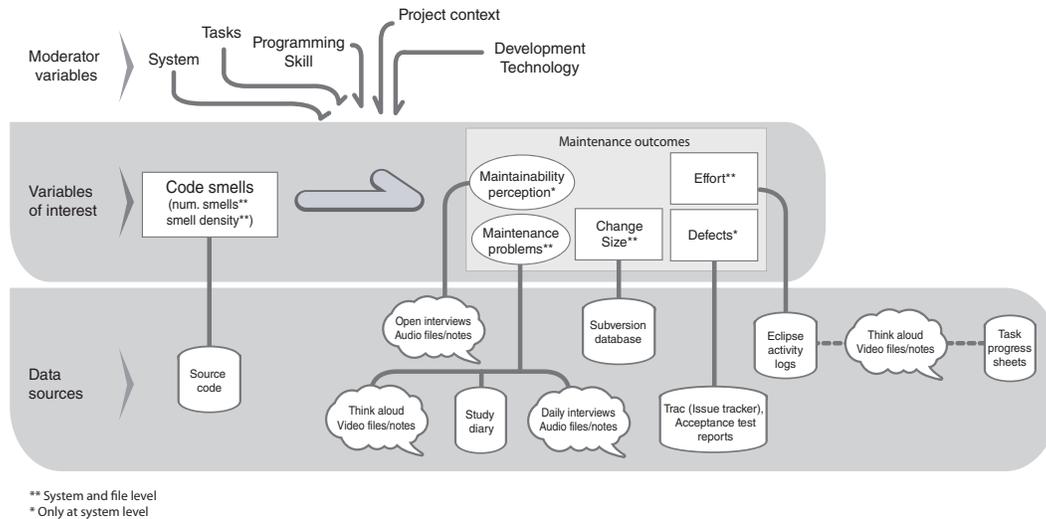


Figure 2. Illustration of variables involved in the study and the corresponding data sources.

once the three maintenance tasks of one system had been completed. (3) *Maintenance problems encountered by the developers during maintenance*: These include a qualitative aspect of the maintenance process based on problems reported through interviews or think-aloud sessions or observed by the researcher during the maintenance work. (4) *Change size*: This variable constitutes an outcome variable from the maintenance project, reflecting the sum of LOC added, changed, and deleted (i.e., file churn). (5) *Effort*: Another maintenance outcome, measured by time spent on the tasks and each of the files in the systems, (6) *Defects introduced during maintenance*: This variable is an aspect of the quality of the system after the tasks were completed.

Figure 2 also discriminates between outcomes/aspects that were observed at the system level (one asterisk) and at both system and file levels (two asterisks). The figure also distinguishes maintenance problems and maintainability perception—which are categorized as *qualitative aspects* (circles)—from change size, effort, and defects—which are categorized as *quantitative outcomes* (squares). This Figure also depicts the data sources from which each of the maintenance outcomes/aspects was derived.

#### F. Analysis Conducted

To investigate “How good are code smells in reflecting system-level maintainability of software (and how they compare to other system-level assessment approaches)”, code smells were aggregated at the system level, and each system was ranked according to the *amount of code smells* they contained and their *code smell density* (i.e., less code smells and lower smell densities mean better maintainability ranking of a system). After the systems had undergone maintenance work, the maintenance outcomes: *total effort*, and the *introduced defects* at the system level were collected per system to

rank them accordingly. To avoid the learning effect problems, we used only the data from the first round per developer. Cohen’s kappa coefficient<sup>6</sup> was used to statistically measure the degree of agreement between the code-smell-based and maintenance-outcome-based rankings. Previous maintainability assessments of the systems based on a subset of C&K metrics and expert judgments, as reported in Ref. [7], were also compared with the maintenance-outcome-based rankings to analyze the differences in accuracy between the code-smell-based, expert-judgment-based, and metrics-based approaches for maintainability assessments.

To investigate “How good code smells are on identifying source code files that are likely to require more maintenance effort (time) than others”, we focused on Java files as the unit of analysis and used *multiple regression analysis*. Effort at *file level* (effort used to view or update a file) was the variable to be explained. Variables representing the different code smells, the file size (measured in LOC), the number of revisions on a file, the system, the developer, and the round were included as independent variables. Several regression models, with different subsets of variables, were built to compare their fit and to discern the predictive capability of each of the variables considered.

To investigate “How good code smells are on discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance”, we focused on Java files as the unit of analysis and by used binary logistic regression analysis. The variable to explain was the variable “problematic,” which was *true* (1) if a file was deemed problematic during maintenance by at least one developer who worked with the file, but *false* otherwise (0). The different types of code smells, files size

<sup>6</sup> Cohen’s kappa coefficient is a statistical measure to represent inter-rater agreement for categorical items.

(measured in LOC), and change size (churn) were used as independent variables. A follow-up qualitative analysis based on the data from the interviews and the think-aloud sessions was performed (1) to support/challenge the findings from the binary logistic regression and (2) to understand better how the presence of a code smell contributed to the problems experienced by the developers during maintenance.

To investigate “How good code smells reflect maintainability aspects that are as deemed critical by software developers”, we conducted qualitative analysis, which compared the developers’ perceptions on the maintainability of the systems with the goal of identifying a set of factors relevant to maintainability. These factors were related to current definitions of code smells to observe their conceptual relatedness. The transcripts of the open-ended interviews were analyzed through *open* and *axial coding* [8]. The identified factors were summarized and compared across cases using a technique called *cross-case synthesis* [3]. The factors derived from this analysis were compared with the factors reported in a previous study [7], which were extracted via expert judgment.

### III. FINDINGS AND LEARNINGS

This section discusses the main findings from this thesis and some lessons learned along the research process.

#### A. Finding 1: Aggregated code smells are not so good indicators of system-level maintainability

System-level indicators of maintainability based on aggregated number of code smells were investigated in the four systems where a system’s maintainability was ranked according to code smell measures and compared with respect to the maintenance outcome measures – change effort, and number of defects. Given that many code smells definitions are based on size-related parameters, aggregating them at system level would not provide more information than the more traditional *system size* measured in LOC. It was found that expert-judgment-based assessment was the most flexible of all the three approaches (i.e., code smell, C&K metrics, and expert-judgment) because it considered both the *effect* of the system size and the potential *maintenance scenarios* (e.g., small versus large extensions). However, we found that if smell density (code smells/LOC) is used to compare systems of similar size, it is likely to provide a more accurate assessment than the expert-judgment based one. We conclude that an advantage of the use of code smells is that when comparing similarly sized systems, they can spot critical areas that experts may overlook. This finding is reported in detail in [9].

#### B. Finding 2: Code smells are not good indicators of effort at file level

We conducted multiple regression, including the number of different code smells per file, the file size (LOC), and

number of changes in the file, as the explanatory variables for file effort. We found that although with an  $R^2$  of 0.58, the only code smell that constituted a significant independent variable of effort ( $\alpha < 0.01$ ) in the regression model is *Refused Bequest*, which interestingly, displayed a *decreasing* effect on effort. If we exclude the code smells from this model (i.e., leaving only file size and number of changes as variables), the  $R^2$  remained at 0.58. This implies that code smells may not provide additional explanatory power than file size and the number of revisions in the context of explaining effort usage per file. This finding is reported in detail in [10]

#### C. Finding 3: Code smells may be promising indicators of problematic files during maintenance

To investigate the capability of code smells to uncover problematic codes, binary logistic regression was conducted. We built a model in which the variables related to the different code smells, the file size, and the file churn were entered in a single step. The  $R^2$  values (Hosmer & Lemeshow = 0.864, Cox & Snell = 0.233, and Nagelkerke = 0.367) confirm that our model provided a reasonably good fit of the data. In the model, the odds ratio for the code smell *ISP Violation* was the largest [Exp(B) = 7.610,  $p = 0.032$ ], which suggests that this code smell was able to explain much of the maintenance problems at the file level. The model also finds *Data Clump*<sup>7</sup> as a significant contributor [Exp(B) = 0.053,  $p = 0.029$ ], but contrary to *ISP Violation*<sup>8</sup>, this code smell indicates less maintenance problems. Some of the problems caused by *ISP Violation* were: (1) Error propagation, (2) Change propagation, (3) Difficulties identifying the task context, and (4) Confusion due to inconsistent design. This finding, is reported in detail in [11].

#### D. Finding 4: Some code smells may deserve more attention from a practical maintenance perspective

Several factors were identified as critical by the developers: appropriate technical platform, coherent naming, design suited to the problem domain, encapsulation, inheritance, (proprietary) libraries, simplicity, architecture, design consistency, duplicated code, initial defects, logic spread, and use of components, where *design consistency* was considered as one of the most important factors. We found that there are code smells capable of supporting the analysis of the several of the maintainability factors—*encapsulation, design consistency, logic spread, simplicity, and use of components*.

For example, *simplicity* is a factor traditionally addressed by static analysis means, but it is also closely related to God Class, God Method, Lazy Class, Message Chains, and Long Parameter List. Similarly, *logic spread* is related to

<sup>7</sup> Clumps of variables appearing repetitively across the code. <sup>8</sup> The violation of the *Interface Segregation Principle*, which dictates that there should not be ‘all-purpose’ interfaces with wide-spread incoming dependencies.

Feature Envy, Shotgun Surgery, and ISP Violation, and *design consistency* is related to several code smells: Alternative Classes with Different Interfaces, ISP Violation, Divergent Change, and Temporary Field. We concluded that in some cases, code smells would need to be complemented with alternative approaches, such as expert judgment (see Refs. [7, 12]) and semantic analysis techniques (for example, see Maletic et al. [13]) to achieve a comprehensive assessment of maintainability. This finding is reported in detail in [14].

*E. Finding 5: Code smells can ‘interact’ with each other, or with other types of design shortcomings*

In this study, we discovered that interaction effects occur between code smells and also between other kinds of design shortcomings, often causing more maintenance problems than when interactions do not occur. Code smells that appear together in the same artifact (file) can interact with each other, but also interaction effects can occur between code smells that are distributed across *coupled* artifacts (e.g., artifacts that display data/functional dependencies). Consequently, in practice, there is no difference between the interaction effects of *coupled smells* and the interaction effects of *collocated smells*. This finding has considerable implications for further studies on code smells, since it means that, to get a more complete understanding of the role of code smells in software maintenance, dependency analysis should be included in the code smell analysis process. This finding is reported in detail in [15].

*F. Lessons learned*

Despite the intricacies and challenges involved, this study design allowed us to conduct both *theoretical* and *literal* replication in a case study, which is often very difficult to attain. This enabled the cross-validation of the observations across cases, and strengthened the internal validity of findings derived from qualitative sources. By combining qualitative and quantitative data collection, we were also able to use a mixed method approach, which not only can identify trends or connections between variables, but also help to derive theories or explanations for those relationships.

Although the element of ‘control’ introduced a certain degree of *artificiality* in a case study, the degree of “intrusiveness” was not found to differ greatly from “normal” case studies. Considering the fact that designs as the one presented here can provide richness in details (from an in-vivo context) that cannot be achieved by experimental settings, we suggest that this approach may deserve further exploration and usage.

One particular challenge was that although it is important to adhere to protocols in normal case studies, within case studies that need a certain element of control, this becomes of paramount importance for the validity of certain results. Since the environment for studies of this nature often constitute an industrial setting, we often face situations where certain factors cannot be controlled for. Consequently,

it is important to prepare adequately beforehand, building contingency plans on potential issues from *practical* and *research* perspective. This study also showed the importance of a study protocol that can be “tested” through a pilot study. Pilot studies can allow the identification of potential threats to validity, and practical issues not contemplated in the protocol. A pilot study helps also to adjust the time that is allocated to each of the different activities (project-related and research related), which can support better management of the study, and the prioritization of data collection activities. Consequently, whenever possible, we strongly recommend to run a pilot study, even if it constitutes a down-scaled version of the final study.

Another great challenge of this study after the data was collected, was the summarisation, integration and analysis of the data. In particular, processing and indexing large amounts of log-data (e.g., the MimEc logs amounted for 1400 hours in approx. 87MB of .csv files) and qualitative data (e.g., 3000 minutes for daily interviews and 480 minutes of recorded audio from the open interviews) was found extremely time consuming. Also, the integration of the different data sources for the purposes of data triangulation had to be automated via a Java program written for that purpose. Although we counted with a structured and functional repository for storing the data from the study, more “on the fly” analysis and summarization would have been preferred, as this would have eased the navigation of the data in latter stages. In our study, the lack of human resources during the study execution limited the amount of “on the fly” analysis that could have been conducted otherwise.

A very useful approach for facilitating the indexing of data in this study was a *logbook*, where the researcher logged all her observations during the study. This was an invaluable resource to pinpoint and identify observations of interest that could be further examined in through other data sources, and to synchronise time-wise the different events during the project across the different data sources. For example, as the logbook contained the dates of the observations, it was an intuitive way to navigate the progress of the project and validate the events via multiple sources sharing the same date, such as the SVN or the progress report written by the developers.

In conclusion, an adequate balance should be seek for; for example, “how much data” should be collected, versus how much resources should be used in terms of time, staff and funds. Our experiences in this study warns researchers against focusing on a too wide research scope, in detriment of an adequate number of human resources/staff that can carry out, summarise and analyse the data in a large study.

#### IV. FUTURE WORK FOR THE COMMUNITY

The areas for future work identified through this research include the following:

1) *Interaction effects among code smells*: More focus is needed on the implications of combinations of code smells (and other types of design flaws) on maintainability instead of investigating only the effects of individual code smells (this corresponds with the ideas of Walter and Pietrzak [16]). This entails building more comprehensive symptomatic characterizations of different types of potential maintenance problems (e.g., in the form of *inter-smell relations*) and uncovering the causal mechanisms that lead to them.

2) *Study of collocated smells and coupled smells*: More focus is needed on dependency analysis alongside the analysis of interaction effects across code smells. We suggest this among others because interactions between code smells can occur across coupled files. This interaction is currently ignored due to the fact that code smells are mostly analyzed at the file level and “coupled code smells” are not identified. Also, the dependency analysis should focus on *types* of dependencies (e.g., data, functional, abstract definition, and inheritance) and their *quantifiable attributes* (e.g., intensity, spread, depth).

3) *Nature and severity of maintenance problems*: Future work should focus on quantifying the severity and the degree of the impact of different types of maintenance problems in different contexts to establish the *relative* importance and context dependency of code smells. This way, it may be possible to assess not only whether and how code smells cause maintenance problems, but also *how much* those problems matter on concrete outcomes of maintenance projects compared with other problems and in different contexts.

4) *Cost-/benefit-based definition/detection of code smells*: Further research should focus on defining and extending a catalog of design factors that have empirical evidence of their relevance on maintainability. This catalog may be used to guide further efforts in new definitions of code smells and corresponding detection methods/tools.

As a final remark, we cite Hannay et al. [17] who asserted that theory-driven research is not yet a major issue in empirical software engineering and referred to several articles that commented explicitly on the lack of relevant theory. Case study research could significantly contribute to the development of theories from observations in relevant fields and contexts (i.e., *inductive research* [3]). Runeson [18] equally argues for the adequacy of case studies in the Software Engineering field, given the commonalities of our discipline to fields as Social and Political Sciences, where the complexity of the context plays an intrinsic role on the different phenomena being investigated.

We hope that the example and experiences described here can help further on the design and conduction of “mixed approaches” not only combining ‘quantitative’ and ‘qualitative’ techniques, but also combining features from

‘case studies’ with ‘experimental’ approaches in the field of Software Engineering.

#### REFERENCES

- [1] A. Yamashita, “Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach,” Doctoral Thesis, University of Oslo, 2012.
- [2] G. R. Bergersen and J.-E. Gustafsson, “Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective,” *Journal of Individual Differences*, vol. 32, no. 4, pp. 201–209, 2011.
- [3] R. Yin, *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.
- [4] L. M. Layman, L. A. Williams, and R. St. Amant, “MimEc,” in *Int’l Ws. Cooperative and Human Aspects of Softw. Eng. (CHASE)*. New York, New York, USA: ACM Press, 2008, pp. 73–76.
- [5] Borland, “Borland Together. <http://www.borland.com/us/products/together>. Accessed 10 May 2012,” 2012.
- [6] Intooitus, “InCode. <http://www.intooitus.com/inCode.html>. Accessed 10 May 2012,” 2012.
- [7] B. C. D. Anda, “Assessing Software System Maintainability using Structural Measures and Expert Assessments,” in *IEEE Int’l Conf. Softw. Maintenance (ICSM)*, 2007, pp. 204–213.
- [8] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 1998.
- [9] A. Yamashita and S. Counsell, “Code smells as system-level indicators of maintainability: An Empirical Study,” *Journal of Systems and Software*, 2013.
- [10] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, “Quantifying the Effect of Code Smells on Maintenance Effort,” *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2013.
- [11] A. Yamashita, “Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data,” *Empirical Software Engineering*, pp. 1–33, 2013.
- [12] M. Jørgensen, “Estimation of Software Development Work Effort: Evidence on Expert Judgment and Formal Models,” *International Journal of Forecasting*, vol. 23, no. 3, pp. 449–462, 2007.
- [13] J. I. Maletic and A. Marcus, “Supporting program comprehension using semantic and structural information,” in *Int’l Conf. Softw. Eng. (ICSE)*, ser. ICSE ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 103–112.
- [14] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?” in *IEEE Int’l Conf. Softw. Maintenance (ICSM)*, 2012, pp. 306–315.
- [15] A. Yamashita and L. Moonen, “Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study,” in *Int’l Conf. Softw. Eng. (ICSE)*, 2013, pp. 682–691.
- [16] B. Walter and B. Pietrzak, “Multi-criteria Detection of Bad Smells in Code with UTA Method 2 Data Sources for Smell Detection,” in *Extreme Programming and Agile Processes in Softw. Eng. (XP)*. Springer Berlin / Heidelberg, 2005, pp. 154–161.
- [17] J. E. Hannay, D. I. K. Sjøberg, and T. Dybå, “A Systematic Review of Theory Use in Software Engineering Experiments,” *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 87–107, 2007.
- [18] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

# Refactoring Planning for Design Smell Correction: Summary, Opportunities and Lessons Learned

Javier Pérez

Ansymo group, University of Antwerp, Belgium

Email: javier.perez@ua.ac.be

**Abstract**—Complex refactoring processes, such as applying big refactorings or removing design smells are difficult to perform in practice. The complexity of these processes is partly due to their heuristic nature and to the constraints imposed by preconditions on the applicability of the individual refactorings. Developers have to find out manually how to apply a complex refactoring “recipe”, from a refactoring book, for each particular situation. In a PhD thesis, we developed an approach for tackling this problem. We described how to better write refactoring “recipes” (Refactoring Strategies) and how to compute, from them, the precise refactoring sequences for each particular situation (Refactoring Plans). Our proposal introduced, for the first time, the use of automated planning for this kind of software engineering problems. This paper presents a short summary of that PhD thesis and discuss the future work, open questions, new research opportunities arisen and the lessons learned from it.

## I. INTRODUCTION

Last decade has witnessed a large amount of contributions in Software Refactoring [1]. Refactoring operations were defined, automated and integrated into software development processes. The automatic identification of refactoring opportunities was studied as well. Closely-related to this matter, design smells were introduced, precisely defined, and automatically detected [2]. Our research was broadly aimed at improving the automation of complex refactoring processes. These can be targeted to many different goals, such as introducing or removing design patterns, improving certain software quality factors, or applying a big refactoring. Design smell correction was the complex refactoring process that motivated the work presented here.

In [2] we presented an analysis on the state of the art on design smell correction with refactorings. We identified the automated correction of design smells to be an open problem and introduced the basic ideas of our approach to tackle it. Our proposal relied on targeting these two objectives: an approach to write automated-suitable specifications of complex refactoring processes; an automated support to plan ahead the computation of complex refactoring sequences.

The first requirement was addressed by defining Refactoring Strategies: a way to write specifications of complex refactoring processes. They unified the concepts, languages and terminology used in the literature for describing how to applying a certain refactoring or how to correct a particular design smell. Refactoring Strategies allowed us to write heuristic-based complex refactoring specifications that can be automated. In order to address the second objective, we

used automated planning and particularly Hierarchical Task Network (HTN) planning for addressing the problem. Finally, the full research was developed in the PhD thesis entitled “Refactoring Planning for Design Smells Correction in Object-Oriented Software” [3] and presented to a wider audience in [4]. This paper presents a short summary of that PhD thesis and discusses the future work, open questions, new research opportunities arisen and the lessons learned.

This paper is structured as follows. Section II presents a summary of the PhD thesis. Section III discusses the future work, the open questions and the new research opportunities. Section IV describes the lessons learned along the elaboration of this study.

## II. THESIS SUMMARY

### A. The problem: Supporting complex refactoring processes

When a refactoring process is used to solve a complex problem, such as the correction of design smells, a significant amount of changes is needed. Refactorings’ preconditions can help assure behaviour preservation, but at the same time they hinder the application of complex transformation sequences, because they restrict the applicability of refactoring operations. Preconditions make it hard for the developer to perform complex refactoring processes. In these situations we have to find the precise refactoring sequence for each particular case.

Moreover, refactorings alone are useless. Moving source code entities around does not guarantee improving the design of a software system unless the redesign process is carefully planned in advance. Improving the design for a particular objective –what is called *strategic refactoring* [5]– requires refactoring tools more advanced than those currently available. Strategic refactorings can be addressed to achieve different kinds of goals such as removing a design smell, introducing or removing a design pattern or improving the testability of a system [6], [7], [8]. These kind of redesign tasks are complex, error-prone and resource consuming. They comprise long sequences of many refactorings and other additional changes.

This leads us to the problem we tried to solve: to give automated or semi-automated support to find out and schedule sequences of refactorings in the context of software design smells correction. The problem exists, among other reasons, because preconditions can disable the application of a refactoring over the current system. Even a simple design change implies using additional or alternative sequences of

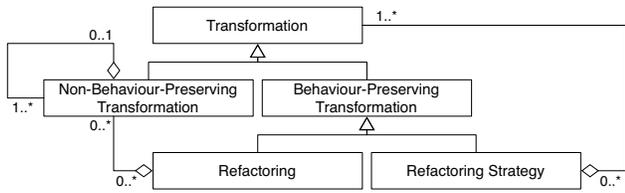


Fig. 1. The higher-level model of Refactoring Strategies

refactorings that have to be planned ahead for each particular case. This problem is important because design smells have a negative effect over software quality factors and correcting them can help reduce development and maintenance costs.

### B. Computing refactoring plans from refactoring strategies

Refactoring Strategies are automation-suitable specifications of the steps to perform complex behaviour preserving transformations. A refactoring strategy compiles the empirical knowledge on how to achieve a particular goal that needs to be reached by application of complex behaviour-preserving transformations. Several kinds of goals are possible: removing a design smell, introducing or removing a design pattern, applying a complex refactoring operation, etc. Refactoring strategies can be seen as a revised version of the reorganization strategies defined by Trifu [9].

Refactoring strategies are instantiated into Refactoring Plans, which are sequences of transformations, aimed at achieving a certain goal, that can be effectively applied over a software system in its current state, while preserving its observable behaviour. This separates the specification of a correction strategy from the executable instance of that strategy which is applicable over a system in a certain state. The computation of refactoring plans from strategies produces a sequence of refactorings whose preconditions are fulfilled at the time of their application. Automated generation of refactoring plans enables a complex behaviour-preserving transformation to be planned ahead, so the sequence of instantiated refactorings included in the refactoring plan can be safely executed in their entirety.

A refactoring strategy (see fig. 1) defines a complex behaviour-preserving transformation sequence that is composed of substrategies, refactorings and non-behaviour-preserving transformations. A refactoring strategy is organised into smaller strategy steps that can be combined with different strategy control constructs, such as alternatives, loops and invocations. The lowest-level transformations in our proposal are represented by atomic add, remove and replace operations over the basic elements in the AST-like system’s representation we use. System queries for gathering and checking information regarding the system’s current state are also part of refactoring strategies’ specifications.

### C. A refactoring planner prototype

Automated planning [10] is an artificial intelligence technique for generating sequences of actions that will achieve

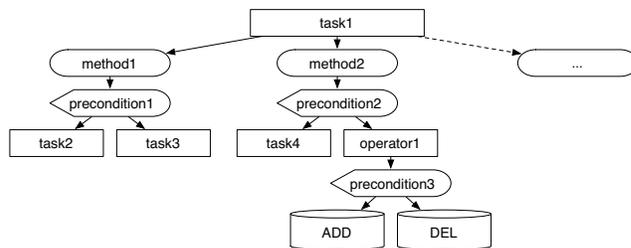


Fig. 2. A simplified model of HTNs

a certain goal when they are performed. We found that automated planning is a technique suitable to be used in the generation of refactoring plans. Among all the planning approaches we analysed, we concluded that hierarchical task network (HTN) planning provides the best balance between search-based and procedural-based strategies for the problem of refactoring planning. HTN planning [10], introduces the concept of “task”, which models actions composed by simple operators or by other tasks. Task networks allow us to include domain knowledge describing which subtasks should be performed to accomplish another one. Task networks represent “recipes” on how to achieve certain objectives.

HTN planning and forward search together make it possible to write very expressive domain definitions. This rich domain knowledge definitions can guide the planning process in a very efficient way. In HTN forward planning, tasks are reduced in the same order as the operators will be applied, therefore the current system state is always known. This enables including numerical computations, interacting with external information sources, writing complex queries that can be computed with a problem solver, etc. For our problem, the ability to perform numerical computations over the current state of the system is crucial, for example, for metrics calculation. The chance to interact with external systems permits to attach external functions and procedures, *e.g.* user queries such as asking the user for the name of a method to be newly created.

In figure 2 we represent a simplified model for HTNs. Tasks can be decomposed into other tasks by different alternative decompositions. Selecting a particular decomposition can depend on the fulfilment of a precondition. We can define two task decomposition types: methods –compound tasks that specify just decompositions– and operators –simple tasks that describe the actual transformations over the world’s state. The transformation is specified, at its lowest level, by indicating which terms the operator will add to, and delete from, the current world’s state. The application of an operator can also be restricted with a precondition. Logic expressions and predicates are used in preconditions and for deriving additional knowledge from the world’s state.

In the context of the HTN approach, we used a logical representation of the current AST of the system as the current state in the planner. HTN operators are used to represent atomic transformations over the basic elements of the AST. Refactoring strategies, simpler refactorings and

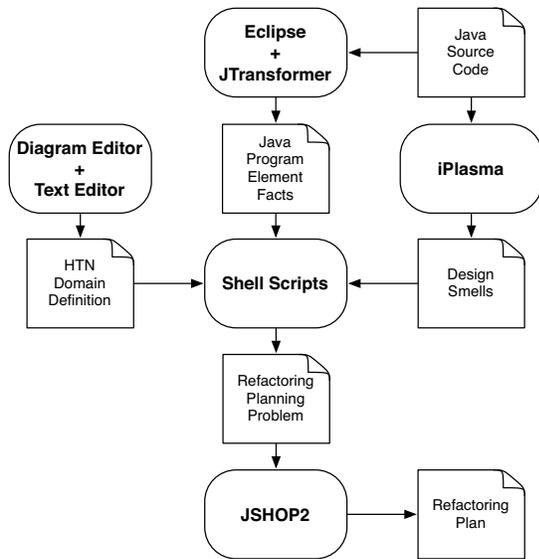


Fig. 3. Overview of the different tools used in the experiments.

non-behaviour-preserving transformations are implemented as HTN tasks, which can be further decomposed into other tasks. This decomposition construct will allow us to split strategies into simpler ones and to attach preconditions to them. Therefore, specification of dependencies and conflicts between behaviour-preserving transformations or parts of them can be written. Invocation of transformations, refactorings and refactoring strategies are described as task decompositions. Complex system queries and preconditions, are formulated over the logical terms representing the system’s AST. This is possible due to the expressiveness of the HTN forward planner we use.

Depending on the goal defined by a refactoring strategy, the task which implements it can be addressed to the application of a refactoring, the removal of a bad smell, or any subpart of these activities. The domain definition represented in the task network represents the heuristic knowledge, or “recipes”, on how to apply these transformations. In order to demonstrate the feasibility of our approach, we have developed a small HTN domain for the refactoring planning problem, which addresses “Feature Envy” and “Data Class” design smells. This domain is used with JSHOP2 [11], the HTN planner we have selected. We have also used some other tools from other researchers, in particular JTransformer [12] and iPlasma [13], for prototyping our approach. An overview of the different components integrated in the prototype is displayed in figure 3.

JSHOP2 is a HTN forward planner by the University of Maryland [11]. It supports arbitrarily complex queries that can be written as axioms in Horn-clauses Prolog style. The JSHOP2 planner includes a problem solver that computes these queries in an efficient way. Special queries, named “call terms”, allow the planner to call external functions during the planning process. In order to write problem specifications for the planner, the domain knowledge –the heuristic rules–

have to be written as HTNs in a language with LISP-like syntax. The representation of the system’s states has to be given to the planner as a set of logical terms in the same LISP-like language. A distinctive feature of JSHOP2 is its precompilation stage. When feeded a planning problem, the tool precompiles the problem and produces a Java program. This Java program contains an optimized version of the problem definition –the rules and the system’s initial state– and the planner itself. In order to solve the planning problem, this Java code has to be compiled and executed. We used the 1.0.3 version of JSHOP2.

JTransformer is a tool developed by the ROOTS group [12]. It is a Java analysing and transformation tool based in Prolog. It converts Eclipse projects into logical representations of their ASTs. They call these first-order-logic predicates program element facts (PEFs). With this tool an Eclipse project can be analysed and manipulated with complex queries and transformations written in Prolog. It is distributed as an Eclipse plugin and, in its most current version –2.8.0–, it supports the full Java 1.4, 1.5 and 1.6 specifications, except for generics. We used the 2.3.1 version of the tool.

iPlasma is a tool by the LOOSE group [13]. It is an integrated environment that performs a great variety of quality analysis over Object-Oriented systems. Among other features, it contains tools for visualisation, metrics computation and bad smells detection. We used the 6.1 version of the tool.

To search for refactoring plans we initially obtain the logical representation of the system we want to process. We rely on JTransformer to convert the system’s AST into a set of logical terms. This set builds up the initial state of the world for the planner. Additionally, we use iPlasma to detect bad smells in the system, and to generate smell-entity reports that complement the information about the current state of the system.

In order to instantiate a refactoring plan the tool should be able to perform some computations such as checking the applicability of refactorings. In order to do this, we must be able to query the state of the system and to derive additional knowledge from the set of facts that represent the current system AST. We have used Eclipse+JTransformer combined for this. We have developed over 150 system queries.

Finally, we have used a diagramming tool and a text editor to write refactoring strategies into HTNs domain knowledge. The diagramming tool has been useful for designing the task networks that have then been coded with a simple text editor. In order to feed the planner with the needed information, we have embedded the specifications of a set of refactorings into JSHOP2 task networks. In practice, this actually means to program refactorings in the JSHOP2 domain definition language. Writing and debugging the refactoring planning domain has been the most difficult activity of our approach due to the lack of a custom editor tool. However, these specifications can be easily reused. Moreover, we are currently developing a set of tools to ease writing refactoring strategies. So far, we have written refactoring strategies for: Remove Data Class, Remove Feature Envy and Move Method; and refactoring specifications

TABLE I  
SUMMARY OF THE RESULTS REGARDING THE NUMBER OF PRODUCED PLANS AND THE ELAPSED TIMES OF THE PLANNING PROCESS. TIMES ARE GIVEN IN SECONDS.

Feature Envy						
System	Smells	Plans	%	Mean $T_t$	Mean $T_c$	Mean $T_p$
1	2	1	50	29.52	23.02	6.49
2	2	1	50	30.55	21.57	8.98
3	18	3	16.67	64.1	53.71	10.4
4	26	21	80.77	107.7	103.66	4.04
5	2	0	0	253.69	182.85	70.84
6	17	9	52.94	182.03	148.05	33.98
7	21	11	52.38	413.21	303.29	109.92
8	36	13	36.11	544.79	385.46	159.32
9	45	24	53.33	1065.54	960.83	104.71
Totals	169	83	49.11			

Data Class						
System	Smells	Plans	%	Mean $T_t$	Mean $T_c$	Mean $T_p$
1	7	6	85.71	26.28	22.5	3.78
2	2	2	100	30.56	21.63	8.93
3	16	16	100	62.96	53.48	9.48
4	3	3	100	179.91	104.28	75.63
5	13	11	84.62	206.36	186.15	20.21
6	21	20	95.24	167.55	148.36	19.2
7	40	40	100	431.47	300.36	131.11
8	27	24	88.89	530.18	386.57	143.62
9	29	23	79.31	1021.41	959.76	61.65
Totals	158	145	91.77			

for 9 refactorings: Encapsulate Field, Move Method, Rename Method, Rename Field, Rename Parameter, Rename Local Variable, Remove Field, Remove Method and Remove Class.

We have used a simple graphical notation to display how the strategies have been designed and organised in the refactoring planning domain (see fig. 4). This example shows a set of strategies for removing a Data Class. A simple refactoring strategy to remove a Data Class will explore two alternatives. A trivial strategy that will simply remove the class and a more complex strategy that tries to reorganise some class members. This reorganisation is attempted with 3 substrategies: moving Feature Envy methods that are accessing the Data Class, moving other client methods accessing the class or cleaning the class from public exposed fields. This last substrategy implies to apply encapsulate field as much as possible and then to remove the unused accessor methods.

As a summary, the refactoring planning problem definition is composed of the current system PEF representation, a smell-entities listing, a collection of system queries and a set of specifications for refactorings, refactoring strategies and non-behaviour-preserving transformations. A set of shell scripts glue these pieces together by compiling and translating all the files into the JSHOP2 language and into a refactoring planning problem specification that is sent to the planner.

#### D. Validation through two use cases

In order to test our approach we have ran experiments over a set of open source systems. The motivation of the case study is to instantiate refactoring strategies, aimed at removing “Data Class” and “Feature Envy” smells, into specific refactoring plans for the systems selected.

The general results of the experiments are shown in Table I. Regarding effectiveness, the first point worth of discussion is the difference in the total number of plans produced for each design smell. The number of successfully generated plans (col. %) depends on the strategies’ knowledge base implemented. In the case of the Feature Envy case study, our prototype has been able to produce almost 50% of the requested plans, while it has generated plans for over 90% of the experiments in the Data Class case study. Despite the simplicity of the sample strategies implemented in the prototype, the results obtained are quite acceptable. Therefore, we consider them as good results. The difference in the success ratios between the Feature Envy and the Data Class case studies is due to the nature of the implemented strategies. The Feature Envy strategy, which is not described in this paper due to lack of space, can only generate a plan when the targeted method can be either removed or displaced to another class, closer to the data it accesses. The Data Class strategy has more chances to success. It can generate a plan when the targeted class can be removed, or when any method can be transferred to the class, or either when some “class cleaning” is performed by improving the encapsulation of its publicly exposed fields.

We will focus now on discussing the efficiency and scalability of the prototype. During the development of the refactoring planning domain definitions, we have noticed that, with JSHOP2, the biggest fraction of the total elapsed time ( $T_t$ ) is due to the JSHOP2 precompilation stage. This process, as stated by the authors of JSHOP2, performs some optimizations that might increase the efficiency of the planning process [14]. Nevertheless, the authors use problem domains with a small number of predicates in their initial system’s state definition. Their biggest examples contain about 600 ground literals, while our experiments range from 32.780 to 354.543. We think that the benefits of the precompilation stage may not apply to our case. The added cost of the precompilation stage, which is significant for problems like ours, might overcome the improved efficiency of the planning process. We intend to explore this further in future work. Meanwhile, we have measured precompilation and planning times separately, and we have used just planning time to study our approach.

As summarised in Table I, precompilation times ( $T_c$ ) are tightly correlated to system size, but the correlation between planning time ( $T_p$ ) and system size is less clear. Due to planning time not following a normal distribution, we could not find a correlation function. Nevertheless, even if the growth of the mean planning time were to be correlated to the targeted system size, it appears to scale for bigger systems. The mean planning time is under 3 minutes in all cases. We consider this result to be quite acceptable for our prototype in terms of efficiency and scalability. The great benefit of HTN planning is that it searches a severely pruned state space. The time results are quite good despite the refactoring planning problem having a huge search space.

As for the differences between both design smell case studies (see Table I), for some systems, the mean planning times do not seem to differ very much between the two

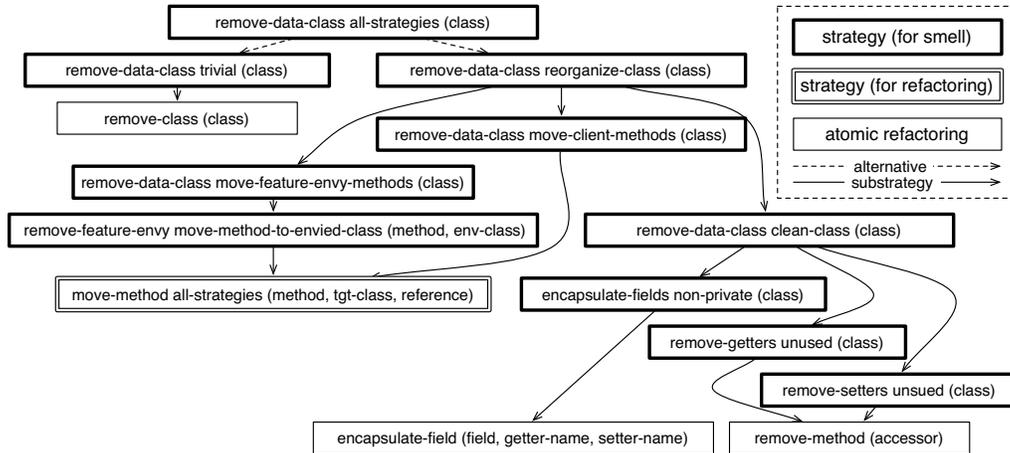


Fig. 4. Overview of a refactoring strategy for removing a *Data Class*

explored case studies –Data Class and Feature Envy. At first sight, the mean planning time is higher in the case of Feature Envy experiments for systems 1, 5 and 6. In the case of the Data Class experiments, they seem to be higher for systems 4, 7 and 9. Finally, the mean planning times for systems 2, 3 and 8 appear to be rather similar between both case studies. Therefore, there is not a clear evidence of the planning times being generally dependent on the requested strategy.

### III. FUTURE WORK AND RESEARCH OPPORTUNITIES

The work summarised in this paper comprises a novel technique that was validated by construction, with a prototype. As such, there are many future work opportunities in improving it. A better integrated and more efficient prototype could be developed, by migrating all the third-party tools we used to a single platform and language, especially, rewriting a custom version of the planner algorithm used. A more integrated tool will improve the usage of metrics within queries and preconditions. Writing and debugging HTN domain knowledge in the planner’s language is a hard task. In order to keep the internal complexity of the tool hidden, while allowing us the reuse of this specifications, we have developed a domain specific language for developing new refactoring strategies. However, the development of tools to facilitate writing refactoring strategies are certainly needed.

Additionally, refactoring strategies could be automatically collected. One way to address this problem is by reusing and reapplying refactoring “recipes” that were successful in the past, formalising the existing expertise in such a way that this knowledge can be collectively and incrementally built, improved, shared and reused. Support for reusable complex refactorings [15] and refactoring customization [16], [17] is a growing topic in the refactoring research community. We envision complex reusable refactorings that can be incrementally mined from the combined refactoring history of many projects, looking for fixed design smells and tracing them back to the

refactorings performed. This knowledge can be gathered from software repositories and by recording the developers actions as well. This proposal has been briefly introduced in [18].

In regards to the type of problem addressed, we can picture new research opportunities from some of the conclusions we extracted during this study. Further exploration of the topics mentioned in this paragraph could lead to interesting and useful opportunities. For example, problems that rely heavily on reapplying known recurrent solutions can be naturally tackled with rule, or “recipe” based techniques. By compiling the existing knowledge into rules we can pave the way to automate and support these problems. Additionally, any problem that can be solved by collecting and reapplying repeated solutions could benefit from the abundance of data and collaboration tools existing today. We think there is a big research opportunity in community driven tools that gather the developer’s practice, with different degrees of automation and awareness. Development tools could then be improved incrementally and collaboratively by with the scripts and tool customisations, collected from and shared by the developers.

Up to our knowledge, our proposal introduced for the first time the use of automated planning for this kind of software engineering problems thus, opening the door to explore other potential scenarios of application. The technique used in this PhD dissertation, automated planning, could be applied to other transformation scheduling problems in software reengineering. Some authors have already started to do so. As a sample, we can mention the use of regression planning for removing model inconsistencies [19] or even HTNs planning for composite refactoring detection [20].

### IV. LESSONS LEARNED

Our work can be classified within the more general field of development recommendation tools. One of the lessons we learned, while identifying and defining the PhD thesis problem, is that a tool should not try to fully automate a certain

task if it cannot deliver better results than manual approaches. It's generally better and more useful to give recommendations and different insights to the developers, while allowing them to carry out the ultimate decisions.

One of the crucial moments in the process of doing my PhD thesis was identifying and defining clearly the topic and problem that I wanted to address. I spent a lot of time exploring the topic of helping the developer in migrating a system's design with refactorings. This work served me to gain a good background of the concepts involved in this topic. However, the problem was not very clear in the beginning. I first tried to support the migration of a system with structural problems to an existing redesign proposal, that I needed to be completely defined with class diagrams. Moreover, it turned out to be an unrealistic problem, because this redesign proposal never existed in a real scenario. I had to step back and think carefully on the definition of a realistic problem. After reading plenty of papers and other PhD thesis about similar topics, I was finally able to formulate a realistic, interesting and feasible problem. Once I had identified the problem to address, everything was more straightforward.

Conferences, research stays and other forms of networking and collaboration also played a very important role in my thesis. A short research stay abroad, was fundamental for defining the scope and goals of the thesis. It also served me to get in touch with other researchers relevant to my topic. Asking the experts also helped me not to drift very much away from the good tracks.

Sitting down to actually write the PhD dissertation document also helped me a lot. By compiling everything into a comprehensive document, I could finally get to understand the big picture. I could determine the scope of the thesis, what I had to do and what not, by writing the introduction of the thesis, and specifying clearly which the problem was, why it was important and relevant and which were the thesis' questions and objectives. Having a general perspective of the thesis allowed me to identify the gaps, what was left to do and the side objectives that were not so relevant to cover or include. Moreover, I have found that writing things down has helped me all the time in clarifying ideas and in establishing reachable goals. Writing for conferences, workshops or even preparing slides for an internal meeting of my research group were useful to establish shorter-term milestones and deadlines.

Last but not least, I believe stubbornness also had a key role in finishing the thesis. Perseverance is a driving force that cannot be neglected. However, in my opinion, perseverance requires confidence in your chances of reaching the goal, while stubbornness pushes you forward even without that.

#### ACKNOWLEDGMENT

I want to thank my PhD thesis supervisor in Valladolid, Yania Crespo, my host in Mons during my research stay, Tom Mens, and my current promotor in Antwerp, Serge Demeyer. This work was partially funded by the spanish government (*Ministerio de Ciencia e Innovación*, project TIN2008-05675).

While writing this paper, Javier Pérez has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)".

#### REFERENCES

- [1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.
- [2] J. Pérez and Y. Crespo, "Perspectives on automated correction of bad smells," in *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. New York, NY, USA: ACM, 2009, pp. 99–108.
- [3] J. Pérez, "Refactoring planning for design smell correction in object-oriented software," Ph.D. dissertation, University of Valladolid, 2011.
- [4] J. Pérez and Y. Crespo, "Computation of refactoring plans from refactoring strategies using HTN planning," in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. Rapperswil, Switzerland: ACM, June 2012, pp. 24–31. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2328880>
- [5] C. J. Neill and P. A. Laplante, "Paying down design debt with strategic refactoring," *IEEE Computer*, vol. 39, no. 12, pp. 131–134, 2006.
- [6] K. Beck and M. Fowler, *Bad Smells in Code*, 1st ed., ser. Refactoring: Improving the Design of Existing Code. Addison-Wesley, June 1999, ch. 3.
- [7] J. Kerievsky, *Refactoring to Patterns*, ser. Addison-Wesley Signature Series. Addison-Wesley Professional, August 2004. [Online]. Available: <http://www.amazon.fr/exec/obidos/ASIN/0321213351/citeulike04-21>
- [8] M. Harman, "Refactoring as testability transformation," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 414–421.
- [9] A. Trifu, O. Seng, and T. Gensler, "Automated design flaw correction in Object-Oriented systems," in *proceedings of the 8<sup>th</sup> Conference on Software Maintenance and Reengineering*, C. Riva and G. Canfora, Eds. IEEE Computer Society Press, March 2004, pp. 174–183.
- [10] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning; Theory and Practice*. Morgan Kaufmann, 2004.
- [11] JSHOP2, Department of Computer Science, University of Maryland, <http://www.cs.umd.edu/projects/shop>.
- [12] JTransformer Engine, ROOTS group, <https://sewiki.iai.uni-bonn.de/research/jtransformer/start>.
- [13] iPlasma, LOOSE Research Group. [Online]. Available: <http://loose.upt.ro/iplasma>
- [14] O. Ilghami and D. Nau, "A general approach to synthesize problem-specific planners," University of Maryland, Tech. Rep. CS-TR-4597, UMIACS-TR-2004-40, October 2003.
- [15] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," in *33rd International Conference on Software Engineering (ICSE'2011)*, Waikiki, Honolulu, Hawaii, May 2011.
- [16] H. Li and S. Thompson, "Let's make refactoring tools user-extensible!" in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. New York, NY, USA: ACM, 2012, pp. 32–39. [Online]. Available: <http://doi.acm.org/10.1145/2328876.2328881>
- [17] M. Hills, P. Klint, and J. J. Vinju, "Scripting a refactoring with rascal and eclipse," in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. New York, NY, USA: ACM, 2012, pp. 40–49. [Online]. Available: <http://doi.acm.org/10.1145/2328876.2328882>
- [18] J. Pérez and A. Murgia, "A proposal for fixing design smells using software refactoring history," RefTest 2013: International Workshop on Refactoring & Testing, June 2013.
- [19] J. Pinna Puissant, R. Van Der Straeten, and T. Mens, "Resolving model inconsistencies using automated regression planning," *Software and Systems Modeling*, pp. 1–21, 2013. [Online]. Available: <http://dx.doi.org/10.1007/s10270-013-0317-9>
- [20] J. Huang, F. Carminati, L. Betev, C. Luzzi, Y. Lu, and D. Zhou, "Identifying composite refactorings with a scripting language," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, May 2011, pp. 267–271.

# Revealing the Effect of Coding Practices on Software Maintainability

Péter Hegedűs

*University of Szeged*

*Department of Software Engineering*

*Árpád tér 2. H-6720 Szeged, Hungary*

*hpeter@inf.u-szeged.hu*

**Abstract**—Due to its very obvious and direct connection with the costs of altering the behavior of a software, maintainability is probably the most attractive, observed and evaluated quality characteristic of the software products. There are many coding practices and techniques that may influence the maintainability of a system (e.g. design patterns, coding rules, anti-patterns, refactoring techniques).

However, the empirical evidences of the connection between coding practices and maintainability are vague due to the following reasons: i) finding instances of coding primitives like design patterns, anti-patterns, etc. precisely with reverse engineering tools is not easy; ii) the lack of mature practical quality models for objective calculation of maintainability and handling its ambiguity; iii) few empirical studies directly evaluating the connection of coding techniques and software maintainability.

The presented work focuses on solving these major problems by creating a benchmark for evaluating the performance of different reverse engineering tools and introducing a novel probabilistic approach for measuring software maintainability. By performing case studies based on new analysis methods we evince that there is a significant correlation between the design pattern density and the maintainability of a system, e.g. 0.89 Pearson correlation for JHotDraw. Moreover, preliminary studies show that applying refactoring has indeed a traceable positive impact on software maintainability as anticipated.

**Keywords**—Software Maintainability; ISO/IEC 9126; Design Patterns; Anti-patterns; Refactoring; Case Study;

## I. INTRODUCTION

According to the definition of ISO/IEC 9126 [1], maintainability is “the capability of the software product to be modified”. Based on this definition it is clear that maintainability is in direct connection with the costs of altering the behavior of a software. Hence it became a central issue in modern software industry, and lots of recommendations and counterproposals exist on how to write programs with high maintainability (e.g. design patterns [2], anti-patterns [3], refactoring techniques [4]).

These techniques are widely used well founded concepts, e.g. the common belief is that applying design patterns results in a better OO design, therefore they improve software maintainability as well, or that the existence of anti-patterns causes the system to be harder to maintain. However, there is a surprisingly low number of studies examining the relation of coding practices and maintainability directly. Additionally, there are some controversial findings, e.g. some studies suggest that the use of design patterns not necessarily result in good design [2]. Similarly to design patterns there are controversial opinions about the effects of anti-patterns, e.g.

Abbes et al. found that developers are able to deal with one type of anti-patterns while the existence of more anti-pattern types decrease their productivity significantly [3]. Refactoring is a technique to change the internal structure of a software without changing its external behavior (i.e. its functionality). The reason behind modifying the software without changing its functionality is to achieve a system that is easier to maintain in long terms, therefore concrete evidences of the positive effect of refactorings on maintainability is also of a great value.

Today, software industry is a giant business driven entirely by business needs and profit. Thus maintainability of the systems is often overshadowed by feature developments whose business value is more evident at least in short terms. As applying techniques that improve the maintainability of the code or avoiding structures that deteriorate systems has an additional cost without having a short term financial benefit, they are often neglected by the business stakeholders. By better understanding the relation of different coding practices and their relationship to the maintainability and the long term development costs, it would be possible to show the return on investment of applying these techniques and make them more appealing to the business stakeholders as well, reaching higher quality and cheaper software in general.

Section II describes our contributions to the three identified research problems of empirically analyzing the effect of coding practices on maintainability. In Section III we list the currently ongoing research tasks. We summarize the related work in Section IV and conclude the paper in Section V.

## II. CONTRIBUTIONS

### A. A benchmark for evaluating reverse engineering tools

*Aims:* Reverse engineering tools analyze the source code of a software system and produce various results, which usually point back to the original source code. Such tools are e.g. design pattern or anti-pattern miners. Most of the time these tools present their results in different formats, which makes them very difficult to compare. Moreover, the validation of these results is another major issue since it requires manual effort or a predefined golden standard data set (i.e. a benchmark). Although some initiatives has been made in the field of design pattern instance benchmarking [5], [6], these repositories contain the manually evaluated pattern instances of few systems only. In addition, this type of repository would also be needed for other reverse engineering tools not just for design pattern miners.

*Methodology:* To achieve our research goal, we implement a benchmark called BEFRIEND (BENchmark For Reverse engInEering tools workiNg on source coDe) with which the outputs of reverse engineering tools can be easily and efficiently evaluated and compared. It supports different kinds of tool families, programming languages and software systems, and it enables the users to define their own evaluation criteria. Furthermore, it is a freely available web-application open to the community.

*Accomplished contributions:* We have successfully developed BEFRIEND [7] from the benchmark for evaluating design pattern miner tools called DEEBEE. During the development of BEFRIEND, we were striving for full generalization: an arbitrary number of reverse engineering tasks (domains) can be created, the domain evaluation aspects and the setting of instance siblings can be customized, etc. For uploading the results of different tools, the benchmark provides a plug-in mechanism that supports pattern instance description languages like DPDX [8]. We already applied BEFRIEND on three reverse engineering domains: design pattern mining tools, code clone mining tools, and impact analysis approaches [9]. With the help of the built benchmark we were able to evaluate the most effective tools for the selected tasks and calculate their precision. The benchmark contains the results of five design pattern miner tools, five clone detectors and four impact analysis algorithms for 6 different systems (for details see [7] and [9]).

### B. A Probabilistic Software Quality Model

*Aims:* In order to take the right decisions in estimating the costs and risks of a software change, it is crucial for the developers and managers to be aware of the quality attributes of their software. Maintainability is an important characteristic defined in the ISO/IEC 9126 standard (superseded by ISO/IEC 25010), owing to its direct impact on development costs. Although the standard provides definitions for the quality characteristics, it does not define how they should be computed. Not being tangible notions, these characteristics are hardly expected to be representable by a single number. Existing quality models do not deal with ambiguity coming from subjective interpretations of characteristics, which depend on experience, knowledge, and intuition of experts.

Besides expressing the source code maintainability in terms of numerical values, a model is also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved in order to reach higher overall quality. We propose a novel method for drilling down to the root causes of a quality rating [10].

*Methodology:* We address the problem of handling the ambiguity coming from the subjective interpretation of maintainability in two different ways. As a possible solution, Bakota et al. provide a probabilistic approach [11] for computing high-level quality characteristics defined by the ISO/IEC 9126 [1] standard, which integrates expert knowledge, and deal with ambiguity at the same time.

The method copes with “goodness” functions, which are continuous generalizations of threshold based approaches. The computation of the high-level quality characteristics is based on a directed acyclic graph (see Figure 1) whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). The probabilistic statistical aggregation algorithm uses a benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems. My thesis uses this introduced model in various case studies and extends it in several ways.

As an alternative approach, we collect large number of subjective opinions about the quality characteristics of different source code elements from IT experts and students with various degree of expertise. The quality characteristics are those defined in the ISO/IEC 9126 standard and the evaluators rate these characteristics for a large number of source code elements on a scale from 0 to 10 (0 is the worst, 10 is the best). Using the average votes of the evaluators we can build prediction models based on machine learning techniques using source code metrics as predictors to predict the subjective opinions of humans about the different quality attributes of a software.

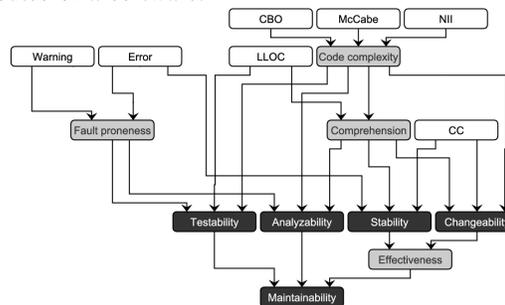


Figure 1. Java maintainability model

*Accomplished contributions:* Examining two Java systems with the introduced probabilistic quality model we found that the changes in the results of the model reflect the development activities, i.e. during development the quality decreases, during maintenance the quality increases. We also found that the goodness values computed by the model show relatively high correlations with the expert votes.

Figure 2 shows the maintainability values of the jEdit open source editor program for almost 1000 revisions. It can be seen that there are peaks in the maintainability values (caused by feature additions, refactorings, etc.) but the long term tendency shows a constant decrease according to the concept of software erosion [12].

Besides the Java language we have also created a maintainability model for C# [13] together with our industrial partners. It achieved a great acceptance from our partners. We compared the results of our model to the opinions of developers and although the average human votes were higher than the estimated values the Pearson correlation analysis gave 0.923 significant at the 0.05 level, a very high

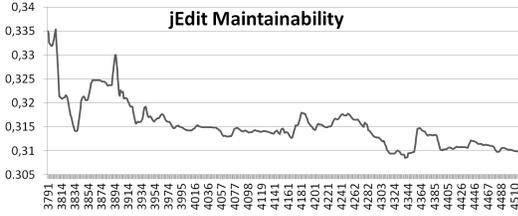


Figure 2. Maintainability trend of jEdit

correlation between the two data sets. However, in this case study the benchmark consisted of the company’s own software products, thus the model calculated the maintainability of the systems relative to each other.

We introduced an algorithm to measure also the maintainability of individual source code elements [10] (e.g. classes, methods). This allows the ranking of source code elements in such a way that the most critical elements can be listed, which enables the system maintainers to spend their resources optimally and achieve maximum improvement of the source code with minimum investment. We validated the approach by comparing the model-based maintainability ranking with the manual ranking of 191 Java methods of the jEdit open source text editor tool. The manual maintainability evaluation of the methods performed by more than 200 students showed a 0.68 Spearman’s correlation ( $p < 0.001$ ) with the model-based evaluation.

We also performed three large case studies to examine the fitness of our second approach for creating software maintainability prediction models [14], [15]. We found that metrics have the potential to predict high level quality indicators assessed by humans (the votes of the evaluators showed a relatively low deviation in general, i.e. between 0.5 and 2 on a scale of 10). The best regression model trained on our evaluation data predicts *maintainability* with 0.72 correlation and 0.83 average error.

### C. The connection of maintainability and coding practices

*Aims:* There are different coding practices that are considered to have either positive or negative impact on the maintainability of the source code. These practices are e.g. design patterns, anti-patterns, code clones, or refactoring techniques. Particularly, the belief of utilizing design patterns to create better quality software is fairly widespread; however, there is relatively little research objectively indicating that their usage is indeed beneficial. On the contrary, some studies revealed that using design patterns can be dangerous [2]. This is true for other coding techniques as well, e.g. anti-patterns and code clones considered to have negative effect on maintainability while refactoring techniques improve it, but unambiguous empirical evidence could not be found in the literature.

*Methodology:* For revealing the connection between coding practices and software maintainability we use our probabilistic maintainability model and data collected in BE-FRIEND and other public repositories [5], [6]. As a first step we analyze more than 300 revisions of JHotDraw, a

Java GUI framework whose design relies heavily on some well-known design patterns. There are also ongoing works to analyze the source code element level maintainability indices of elements playing a role in design patterns.

*Accomplished contributions:* We found that every introduced pattern instance caused an improvement in the different quality attributes [16] for JHotDraw (see Table I). Moreover, the average design pattern line density showed a very high, 0.89 Pearson correlation significant at the 0.05 level with the estimated maintainability.

To further strengthen our first results we repeated the study on 9 different open source systems using the design pattern results of 5 different tools available in the DPB [5] online benchmark. The pattern line densities showed a similarly high Pearson (between 0.59 and 0.78) and Spearman’s (between 0.68 and 0.82) correlation significant at the 0.05 level. Filtering out false positive instances according to the available public repositories significantly increased the correlation values in general (with about 10%).

Table I  
DESIGN PATTERN CHANGES OF JHOTDRAW

Revision	Pattern	Maintainability	Testability	Analyzability	Stability	Changeability
531	+3	↗	↗	↗	↗	↗
574	+1	↗	↗	↗	↗	↗
609	-1	—	—	—	—	—
716	+1	↗	↗	↗	↗	↗
758	+1	↗	↗	↗	↗	↗

### III. ONGOING RESEARCH WORK

*Refactorings:* in a preliminary study we are analyzing a half year long development period of 5 industrial systems with our probabilistic quality model. During this period a heavy refactoring phase took place in each company but some new features were added as well. Each of the refactoring activity related commits have been marked in the commit logs therefore we are able to distinguish between refactoring and feature development commits. Our preliminary investigation shows that commits with refactorings indeed have a traceable positive impact on software maintainability (i.e. the quality characteristics are improving) while the feature developments generally cause decrease in maintainability.

*Anti-patterns:* similarly to the investigation of design pattern density (see Section II-C) we analyze the effect of anti-patterns on maintainability. Besides anti-pattern frequencies we also investigate the relationship of the source code elements taking part in anti-patterns and their relative maintainability indices calculated by our drill-down approach (see Section II-B). We think that source code elements taking part in anti-patterns have lower maintainability than those of non anti-pattern elements.

### IV. RELATED RESEARCH

*Benchmarking reverse engineering tools:* Fontana et al. provides an online repository [5] of design pattern instances of different pattern miner tools. They focus only on design patterns and provide an interactive web interface for browsing, evaluating and getting statistics of instances. Guéhéneuc

also provides an online repository for pattern-like micro architectures, called p-Mart [6].

*Software maintainability models:* Heitlager et al. [17] propose an extension of the ISO/IEC 9126 model that uses source code metrics at low level as a replacement of the Maintainability Index. The technical debt based models like SQALE [18] introduce low-level rules to connect the ISO/IEC 9126 characteristics with metrics. The maintainability is calculated by the total cost which would be necessary to correct the rule violations in the system. Mordal et al. deal with different techniques to aggregate source code element level quality metrics to system level [19].

*Relation of coding practices and maintainability:* In an empirical study Prechelt et al. [20] gave groups identical maintenance tasks to perform on two different versions - with and without design patterns - of four programs. Here, the impact on maintainability was measured by completion time and correctness. Khomh and Guéhéneuc [21] used questionnaires to collect the opinions of 20 experts on how each design pattern helps or hinders them during maintenance. They bring evidence that design patterns should be used with caution during development because they may actually impede maintenance and evolution.

## V. CONCLUSIONS

My thesis aims at providing further empirical evidences of the long term benefits of applying recommended coding practices like design patterns or refactorings and avoiding problematic structures like anti-patterns or code clones. As following these coding guidelines has an additional cost without having a short term financial benefit, they are often neglected by the business stakeholders. This is a common situation according to our industrial experiences even though the developers are usually aware of the long term benefits of following these common coding practices. However, they have no strong enough arguments for investing extra effort to improve long term maintainability. With the presented empirical findings we hope that it becomes easier to present the return on investment of these practices towards business stakeholders to get the required support.

On top of that, my work tries to provide additional information for practitioners also on how, when, and to which extent it is worth to apply the different coding practices to achieve the best balance between long term maintainability and invested development efforts.

## ACKNOWLEDGMENTS

This research was funded by the TÁMOP 4.2.4. A/2-11-1-2012-0001 European grant.

## REFERENCES

- [1] ISO/IEC, *ISO/IEC 9126. Software Engineering – Product quality 6.5*. ISO/IEC, 2001.
- [2] W. B. McNatt and J. M. Bieman, “Coupling of Design Patterns: Common Practices and Their Benefits,” in *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, ser. COMPSAC ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 574–579.
- [3] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension,” in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.
- [4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [5] F. A. Fontana, A. Caracciolo, and M. Zanoni, “DPB: A Benchmark for Design Pattern Detection Tools,” in *CSMR*, 2012, pp. 235–244.
- [6] Y.-G. Guéhéneuc, “PMARt: Pattern-like Micro Architecture Repository,” in *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, July 2007.
- [7] L. J. Fülöp, P. Hegedűs, and R. Ferenc, “Befriend - a Benchmark for Evaluating Reverse Engineering Tools,” *Periodica Polytechnica - Electrical Engineering*, vol. 52, no. 3-4, pp. 153–162, 2008.
- [8] G. Kniesel, A. Binun, P. Hegedűs, L. J. Fülöp, A. Chatzigeorgiou, Y.-G. Guéhéneuc, and N. Tsantalis, “DPDX – A Common Exchange Format for Design Pattern Detection Tools,” in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, Mar. 2010, pp. 232–235.
- [9] G. Tóth, P. Hegedűs, J. Jász, A. Beszédés, and T. Gyimóthy, “Comparison of Different Impact Analysis Methods and Programmer’s Opinion – an Empirical Study,” in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ’10)*. ACM, Sep. 2010, pp. 109–118.
- [10] P. Hegedűs, T. Bakota, G. Ladányi, C. Faragó, and R. Ferenc, “A Drill-Down Approach for Measuring Maintainability at Source Code Element Level,” in *Proceedings of the 7th International Workshop on Software Quality and Maintainability*, ser. SQM, 2013.
- [11] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, “A Probabilistic Software Quality Model,” in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society, 2011, pp. 368–377.
- [12] D. L. Parnas, “Software Aging,” in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE ’94. IEEE Computer Society Press, 1994, pp. 279–287.
- [13] P. Hegedűs, “A Probabilistic Quality Model for C# – an Industrial Case Study,” *Acta Cybernetica*, vol. 21, no. 1, pp. 135–147, 2013.
- [14] P. Hegedűs, G. Ladányi, I. Siket, and R. Ferenc, “Towards Building Method Level Maintainability Models Based on Expert Evaluations,” in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. Springer, 2012, pp. 146–154.
- [15] P. Hegedűs, T. Bakota, L. Illés, G. Ladányi, R. Ferenc, and T. Gyimóthy, “Source Code Metrics and Maintainability: A Case Study,” ser. Communications in Computer and Information Science, vol. 257. Springer, 2011, pp. 272–284.
- [16] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, “Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability,” ser. Communications in Computer and Information Science, vol. 340. Springer, 2012, pp. 138–145.
- [17] I. Heitlager, T. Kuipers, and J. Visser, “A Practical Model for Measuring Maintainability,” in *Proceedings 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007*. IEEE, 2007, pp. 30–39.
- [18] J. L. Letouzey and T. Coq, “The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code,” in *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID)*. IEEE, Aug. 2010, pp. 43–48.
- [19] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, “Software Quality Metrics Aggregation in Industry,” *Journal of Software: Evolution and Process*, 2012.
- [20] L. Prechelt, B. Unger, W. Tichy, P. Brössler, and L. Votta, “A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions,” *IEEE Transactions on Software Engineering*, vol. 27, pp. 1134–1144, 2001.
- [21] F. Khomh and Y.-G. Guéhéneuc, “Do Design Patterns Impact Software Quality Positively?” in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 274–278.

# Automated S/W Reengineering for Fault-Tolerant and Energy-Efficient Distributed Execution

Young-Woo Kwon

Software Innovations Lab, Virginia Tech  
ywkwon@cs.vt.edu

**Abstract**—Reengineering extant software systems into distinct, distributed components communicating across a network has become indispensable in modern software maintenance practices. Such reengineering practices can be applied to migrate portions of an application’s functionality to the cloud, both to take advantage of superior cloud computing resources and to reduce the energy consumption of mobile applications. Executing such reengineering tasks naively results in distributed applications that suffer from two main problems: a lack of reliability and poor energy efficiency. Specifically, transformed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Effective handling of partial failure requires that the reengineering process systematically introduces special failure handling functionality. Furthermore, mainstream middleware architectures, the calls to which the reengineering process inserts, lack sufficient expressiveness and adaptivity to use the limited energy resources of mobile devices optimally. This research addresses these two fundamental reengineering problems through innovation in program transformations and distributed programming abstractions. In this paper, we discuss the general vision and contributions of this research, as well as the related state of the art, and preliminary results.

**Keywords**—Distribution refactoring, fault-tolerance, energy-efficiency, middleware, cloud computing

## I. INTRODUCTION

As distributed execution is rapidly becoming prevalent in the majority of computing domains, an increasing number of software maintenance tasks is concerned with transforming software systems into distinct, distributed components communicating across a network. These reengineering tasks include migrating centralized applications to the cloud to take advantage of cloud computing and offloading energy-intensive client functionality to remote servers to reduce energy consumption of mobile applications. Specifically, using cloud-based remote services has become a common avenue for leveraging cloud computing resources, with the benefits that include reduced costs, increased automation, greater flexibility, and enhanced mobility [1]. Furthermore, to reduce energy consumption, a mobile application can be partitioned into local and remote parts, so that energy intensive functionality is executed remotely in the cloud.

Existing reengineering practices to introduce distribution into existing software systems often lead to distributed

applications that are both unreliable and energy inefficient. First, because centralized and distributed applications have different failure modes, simply rendering a subset of a centralized application remote does not preserve the original semantics. Distributed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Although one cannot handle all the possible failures in a distributed application, some failures have well-known handling strategies. Thus, to better preserve the original execution semantics, programmers must change code to transition applications to use cloud-based remote services and add proper fault handling functionality to any application that invokes services remotely. Moreover, most distributed applications coordinate the execution of multiple remote processes through *middleware* (e.g., sockets, remote procedure calls, messaging, etc.), which provide programming and runtime support for one process to execute functionality in a different process. Because network communication is one of the largest sources of energy consumption [14], the choice and parameterization of middleware and corresponding distributed programming abstractions can reduce the amount of energy consumed by a distributed application.

This research enhances the current state of the art in reengineering software for fault-tolerant and energy-efficient distributed execution. Specifically, this research innovates in the spaces of automated program transformations and distributed programming abstractions, making the following contributions:

- **Automated program transformations for the repositioning to distributed applications:** A set of refactoring techniques facilitate the process of transforming centralized applications to use remote services. These techniques automate the program transformations required to render portions of functionality of a centralized applications as remote services and re-target the application to access the remote functionality.
- **Hardening distributed applications with resiliency against partial failures:** A declarative approach for hardening distributed applications with resiliency against partial failures at the application level. We introduce a domain-specific language for describing both

failures and application-specific hardening strategies to eliminate them, thus enabling programmers to easily integrate the state-of-the-art fault handling solutions.

- **Middleware with dynamic adaptation capabilities for energy efficiency:** Energy-aware adaptive middleware achieves energy-efficiency by means of dynamic adaptation mechanisms. These abstractions enable the programmer to express how to adapt middleware execution patterns in the presence of volatile mobile networks, so as to reduce the mobile application's energy consumption.
- **Systematic assessment of distributed applications across middleware:** A novel mechanism can accurately assess the performance, complexity, reliability, and energy consumption of distributed applications across middleware for accessing remote functionality.
- **An empirical evaluation:** We will apply our approach to a set of third-party mobile applications to show effectively reduce the energy consumed by the applications.

## II. RESEARCH OVERVIEW

The goal of this research is to address the deep conceptual challenges of reengineering existing centralized applications for fault-tolerant and energy-efficient distributed execution. Figure 1 shows how a centralized application can be transformed into a distributed application. In particular, we focus on three software reengineering goals. The first goal involves moving some of a centralized application's functionality to the cloud and adding fault-handling code to the client. The second goal involves handling the faults raised during the invocation of a cloud-based remote service via fault-tolerant middleware. The third goal involves optimizing the energy consumption in a distributed application by flexibly adapting its execution patterns at runtime based on the characteristics of mobile networks, as specified by programmers. We assume that the energy spent on executing the distributed functionality in the cloud is free, as it does not exhaust any battery power of the mobile device. In the following discussion, we describe our approach's individual parts.

### A. Refactoring Centralized Applications

A refactoring—a semantics preserving program transformation performed under programmer control [4]—has become a popular technique in modern software development. In other words, refactoring is automated: a programmer determines if a refactoring should be performed and then a refactoring engine transforms the code automatically.

The goal of this research is to alleviate the code transformation hurdles involved in adapting existing applications to take advantage of cloud-based remote services. To reduce development efforts/costs and increase programmer productivity, we express as refactorings several common program transformations that programmers perform when adapting applications to use cloud computing resources. Although the

approach is not fully automatic, programmers only determine if the source code should be transformed. The actual transformations are performed by a refactoring engine. However, because centralized and distributed applications have different failure modes, transformed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Thus, the refactoring technique adds the ability to handle partial failures. Then, detected partial failures are handled by a underlying middleware, which will be accomplished through the second goal.

1) *Proposed Approach:* The proposed approach focuses on common program transformations occurring when using cloud-based remote services that are well-amenable to be expressed as a refactoring. In particular, we will explore a set of refactoring techniques that facilitate the process of transforming centralized applications to use cloud-based remote services. These techniques automate the program transformations required to 1) rewrite a class making all its methods into remote service methods, 2) partition class methods into service methods and regular methods, rewriting all the communication between the two into remote service calls, 3) re-target all clients of the original class to access its functionality in the cloud by means of remote service calls, and 4) add fault handling functionality to client code.

Instead of individual objects, we use service components as a distribution boundary. Service Oriented Architecture (SOA) has been successfully employed to provide uniform access to a variety of computing resources across multiple application domains. Loosely coupled services may be co-located in the same address space or be geographically dispersed across the network. Among the software engineering advantages of SOA are strong encapsulation, loose coupling, ease of reusability, and standardized discovery.

The distribution refactoring browser of Figure 1 shows how the constituent components of our approach fit together. The two main parts of our approach are *Recommendation Engine* and *Refactoring Engine*. By combining the class coupling metrics collected through both static analysis and runtime monitoring, the recommendation engine suggests classes that can be converted into cloud-based services. The refactoring engine then transforms the given methods into remote service methods, leaves the remaining methods on the client, and rewrites all communication between the original and remote methods into remote service calls. By integrating these engines with a modern IDE, our approach makes it possible to use the new refactoring techniques indistinguishably from the existing ones. The approach is realized using the Open Service Gateway Initiative (OSGi), a widely used service infrastructure, codified as a platform for cloud applications by the request for proposal 133 [13]. The OSGi infrastructure can expose a regular Java class as a service, while our distribution refactoring browser generates the interfaces for accessing this service.

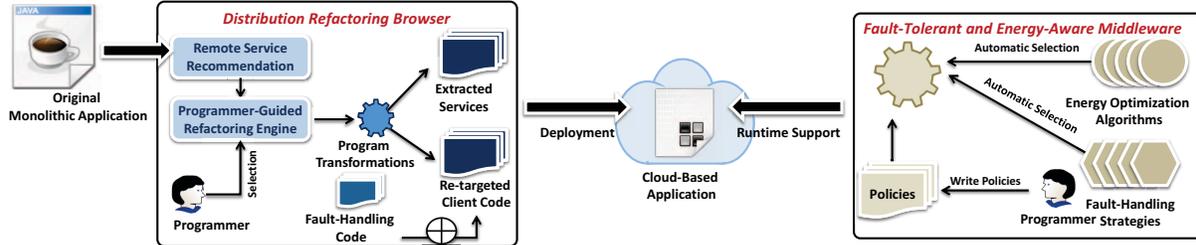


Figure 1. Automated software reengineering procedure.

### B. Hardening Distributed Applications against Failures

Refactoring a centralized program for distributed execution should preserve its original execution behavior, so that a distributed application would provide the same or equivalent functionality to the user as the original centralized version. In other words, distributing an application should only improve its performance, availability, scalability, and efficiency in furnishing the original functionality to the user. One of the most significant impediments to this goal is that centralized and distributed applications have drastically different failure modes. Distributed applications are subject to partial failure, where components of a distributed system can fail independently of each other. Many cases of partial failure are application specific and require an expert to handle properly. Therefore, we focus on *application-level fault handling* as compared to all the system level approaches.

An example of partial failure is *network volatility*, when a network suffers from an outage and then shortly becomes operational again. Despite its temporary nature, network volatility is disruptive and detrimental to the user experience. To facilitate the programming required to handle network volatility, a refactoring adds volatility handling functionality to an application. This functionality can be provided as framework components that can be reused across applications. A domain-specific language expresses the preconditions for applying this refactoring.

1) *Proposed Approach*: We propose an automated refactoring technique that adds the ability to operate disconnectedly to a distributed application. The refactoring is enabled by a component framework that manages *hardening strategies*, service components integrated with distributed applications. The hardening strategies are rendered as reusable software components, which can be developed by third-party programmers for a variety of distributed applications. Refactored into a distributed application, these components will harden it against network volatility.

The fault-tolerant middleware of Figure 1 shows how the constituent components of our approach fit together. Because in the presence of partial failures, existing middleware cannot dynamically handle the raised failures, we will innovate middleware by means of a fault diagnosis module and a strategy manager. The programmer can specify and configure the fault tolerance strategies to apply by means

of a domain-specific language. The fault diagnosis module catches raised exceptions or failures. The strategy manager associates raised exceptions with fault tolerance strategies. In response to detecting an exception, the middleware initiates the handling strategy as configured by a given script written in the domain-specific language. A strategy implementation is simply a sequence of corrective actions whose execution counteracts the effect of experiencing the fault.

### C. Adaptive Middleware to Improve Energy Efficiency

Through our prior work [7], we have studied how distributed applications consume energy and how they can maximize the energy savings. A promising energy optimization target is a middleware mechanism. Since middleware defines the patterns through which a distributed application transmits data across the network, the choice of middleware can heavily impact the amount of energy consumed by mobile applications. Therefore, middleware must take into consideration the hardware capacities of the mobile device running the application as well as the characteristics of the mobile network connecting the mobile device to the cloud.

This research will close the knowledge gap in perfective maintenance by investigating how the choice, parameterization, and optimization of middleware impact the overall energy budget of a distributed application. In addition, mainstream middleware mechanisms cannot flexibly adapt their execution patterns to minimize energy consumption in mobile execution environments [12], in which an application often switches between mobile networks with different bandwidth/latency characteristics.

1) *Proposed Approach*: We will identify the optimization opportunities with respect to energy consumption for a variety of mobile applications running on different hardware capacities. Although several prior approaches focus on identifying how mobile applications consume energy [2], [14], none of these studies have specifically focused on middleware. Our preliminary results indicate that distributed communication indeed consumes a substantial amount of the energy budget of a mobile application. Therefore, our goal is to understand which portions of distributed communication incur the highest energy costs. Then, we will create a novel middleware mechanism, called *energy-aware adaptive middleware*, which features dynamic adaptation capabilities as well as a declarative, domain-specific language that enables

the programmer to express a rich set of middleware energy optimizations and the runtime conditions under which these optimizations should be applied.

Figure 1 shows *energy-aware adaptive middleware* that minimizes energy consumption via advanced, dynamic optimizations. Enabling these optimizations requires innovation in runtime support. A dynamic adaptation mechanism selects an optimal execution patterns at runtime by leveraging the information provided by various system components, thereby enabling the middleware to optimize the energy consumption of software systems [3]. Therefore, the middleware can effectively switch energy optimizations based on the runtime conditions while identifying possible trade-offs between energy consumption and QoS.

Furthermore, the middleware enables the programmer to implement several strategies that should be used under different runtime conditions (e.g., network latency, bandwidth, volatility, etc.). At execution time, the runtime system selects the strategy to use, and dynamically switches strategies in response to the changes in the system environment. With the programmer provided several energy optimization strategies and the runtime deploying them based on the conditions in place, the proposed approach has the potential to reach the energy efficiency levels not achievable through purely static or dynamic energy optimization approaches. In the following discussion, we outline some of the general guidelines for optimizing distributed executions. In the following discussion, we give specific examples of middleware functionality and the alternatives for their implementations.

**Data compression:** The data transferred across the network may be compressed to reduce the amount of consumed energy by leveraging the trade-off between CPU processing and network transfer. Compressing the data will reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Transferring the uncompressed data will result in transferring more data, but it will require less CPU processing. The runtime conditions in place determines which strategy is optimal.

**Multi-hosting:** The multi-hosting optimization examines multiple alternate remote servers to find an optimal execution path (i.e., remote server) with respect to energy consumption. To save energy, an application should try to avoid operating over a congested network. For example, when experiencing a network congestion at `a.com`, the middleware redirects the distributed execution to `b.com`.

### III. PRELIMINARY RESULT AND RESEARCH PLAN

The research described above has focused on common reengineering tasks that take place when software systems are being adapted for distributed execution, such when transitioning to using cloud-based remote services. The completed work of this research is concerned with assessing distributed programming abstractions [11], [10], [7], hardening distributed applications [9], [5], and reducing the energy

consumption in mobile applications [6], [8]. We plan to (1) leverage static and dynamic energy optimizations via a novel middleware architecture, (2) enhance the distribution refactoring via a novel program analysis and transformation, and (3) integrate the distribution refactoring with a modern IDE to help the programmer easily produce fault-tolerant and energy-efficient distributed application. As our evaluation, we will apply the described reengineering techniques to a series of benchmarks and centralized Java applications. The expected results will indicate that our approach can effectively handle partial failures and reduce energy consumption of distributed applications.

### ACKNOWLEDGMENTS

This research is supported by the National Science Foundation through the Grant CCF-1116565.

### REFERENCES

- [1] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5<sup>th</sup> utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
- [2] T. Do, S. Rawshdeh, and W. Shi. pTop: A process-level power profiling tool. In *2<sup>nd</sup> Workshop on Power Aware Computing and Systems (HotPower'09)*, 2009.
- [3] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *22<sup>nd</sup> International Conference on Distributed Computing Systems*, 2002.
- [4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [5] Y.-W. Kwon and E. Tilevich. A declarative approach to hardening services against QoS vulnerabilities. In *2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA '11)*, 2011.
- [6] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *32<sup>nd</sup> International Conference on Distributed Computing Systems*, 2012.
- [7] Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, vol. 55, no. 9, 2013.
- [8] Y.-W. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *29<sup>th</sup> IEEE International Conference on Software Maintenance*, 2013.
- [9] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *ACM/IFIP/USENIX 10<sup>th</sup> International Middleware Conference (Middleware 2009)*, 2009.
- [10] Y.-W. Kwon, E. Tilevich, and W. Cook. Which middleware platform should you choose for your next remote service? *Service Oriented Computing and Applications*, 5:61–70, 2011.
- [11] Y.-W. Kwon, E. Tilevich, and W. R. Cook. An assessment of middleware platforms for accessing remote services. In *7<sup>th</sup> IEEE International Conference on Services Computing*, 2010.
- [12] A. Miettinen and J. Nurminen. Energy efficiency of mobile clients in cloud computing. In *2<sup>nd</sup> USENIX conference on Hot Topics in Cloud Computing*, 2010.
- [13] OSGi Alliance. Request for Proposal 133. 2010.
- [14] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *7<sup>th</sup> ACM European Conference on Computer Systems*, 2012.

## Reverse Engineering Web Sales Configurators

Ebrahim Khalil Abbasi

Advisor: Patrick Heymans

PRECISE, University of Namur, Belgium

{*eab, phe*}@info.fundp.ac.be

**Abstract**—Sales configurators are widespread Web applications. Although such applications have specific common characteristics (e.g., they manage options governed by constraints, they enforce a configuration process...), they are usually developed in an unspecific way, that is, like any other Web application. Proceeding this way leads to configurators that are suboptimal in reliability, efficiency and maintainability.

This PhD thesis is concerned with the reverse-engineering of Web sales configurators. It aims to develop a consistent set of methods, languages and tools to semi-automatically extract configuration-specific data from a Web configurator. Such data is stored in formal models (e.g., variability models, process models). These models can later be used for verification purposes (e.g., checking the completeness and correctness of the configuration constraints) as well as input for generative techniques (e.g., to re-engineer legacy configurators).

More precisely, our two main research questions are: (1) How to extract variability data from the unstructured or semi-structured Web pages of a sales configurator? (2) How to extract such data from the dynamic content created when the configurator is executing? The accuracy of the extracted data and the scalability of the delivered tools are major concerns. The PhD thesis is meant to be completed within the coming year.

**Keywords**—Web; Variability; Reverse Engineering

### I. INTRODUCTION

A *Web Sales Configurator (SC)* provides an online configuration environment for users to specify products that match their individual requirements and preferences. Customized products are often characterised by hundreds of *configuration options* presented to users who have to select the options to be included in the final product. A SC provides an interactive graphical user interface (GUI) that guides the user through the configuration process. It also verifies constraints between options, propagates user decisions, and handles conflictual decisions.

Web SCs vary significantly. They each have their own characteristics, spanning visual aspects (GUI elements) to constraint management. The Web SC of Audi appearing in Figure 1 is thus only one example. It shows a configuration environment in which different options are presented through specific widgets (radio buttons and check boxes – (A) and (B), respectively). These options can be in different states such as activated (e.g., “Privacy glass” is flagged with ✓) or unavailable (e.g., “Twin-pane UV and heat-insulating glass” is greyed out). Additionally, these options are organised in

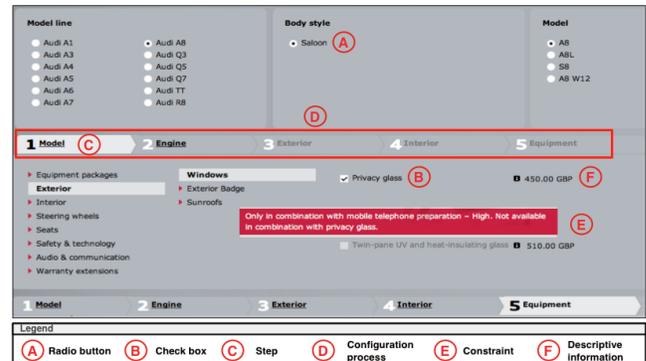


Figure 1. Audi Web SC (<http://configurator.audi.co.uk/>)

different tabs (e.g., “Equipment”) and sub-tabs (e.g., “Equipment packages”) which denote a series of steps (C) in the *configuration process* (e.g., “1. Model” is followed by “2. Engine”—(D)). A SC can also implement *constraints* between options (E). These are usually hidden to the user but they determine valid combinations of options. For instance, the selection of “Privacy glass” implies the deselection of “Twin-pane UV and heat-insulating glass”, meaning that the user cannot select the latter if the former is selected. Moreover, descriptive information (F) is sometimes associated to an option (e.g., its price).

Our previous empirical study of Web SCs reveals maintainability, usability, and reliability issues for the SCs [1]. These problems come from the fact that Web SCs are developed in an ad-hoc manner and no rigorous method is applied to implement critical functions such as consistency checking. To migrate legacy Web SCs to more reliable, efficient, and maintainable solutions, we offer to systematically *re-engineer* those applications. This encompasses two main activities: (1) reverse engineering a legacy Web SC and encoding the extracted data into dedicated formalisms, and then (2) forward engineering new improved SC [2].

Reverse engineering a Web SC is the process of extracting configuration options, their associated descriptive information, and constraints, altogether called *variability data*, from the Web pages of the SC, and then constructing a variability model, for instance, a *feature model* [3]. Once the feature model of the Web SC is built, it can be used in the forward-engineering process to generate a customized and easily maintainable user interface with an underlying reliable reasoning engine [2]. The study of the forward-engineering

process is outside the scope of this PhD thesis.

In this PhD thesis, we proposed the notion of *variability data extraction patterns* (*vde* patterns) to extract structured variability data from the unstructured or semi-structured pages of Web SCs. We developed a Firebug<sup>1</sup> extension to support this process. The Firebug extension consists of two major components: (1) a *Web Wrapper* that seeks and finds code fragments in a Web page that structurally match the given *vde* patterns and extracts the required data from them, and (2) a *Web Crawler* that explores the “configuration space” (i.e., all objects representing configuration-specific data) and simulates users’ configuration actions. The Crawler generates dynamic variability data which are then extracted by the Wrapper.

We proceed as follows. Section II formulates the research questions. Section III explains the proposed research method and Section IV reports on the progress we have made. Section V concludes the paper and presents the contributions of this thesis.

## II. RESEARCH QUESTIONS

The specific goal of the proposed research is to reverse engineer feature models from Web SCs. Building a complete feature model requires analysing both the client and server sides of a SC<sup>2</sup>. We choose to start this journey by investigating the visible part of SCs: the *Web client*. We analyse the client side because (1) it is the entry point for customer orders and most of the variability data is somehow represented in Web pages, (2) large portions of the client-side code are publicly available, and (3) the techniques to reverse engineer Web clients and Web servers differ significantly.

The two main research questions addressed in this PhD thesis can be formulated as follows.

**RQ1.** *What scalable Web data extraction method can we use to collect accurate variability data from the Web pages of a SC?* Currently, most of the variability data is presented in the semi-structured Web pages of a SC (*semi-structured data*). Each SC uses its specific Web objects (e.g., layouts and widgets) to visually represent this data in the page (*variations in presentation*). Moreover, Web SCs use a diversity of patterns to load options in the pages, handle different kinds of constraints, and control the configuration process (*variations in business-logic management*).

The data to be extracted from a page usually is not a *single slot* and individual data item, but a block of related data items should be extracted and reported as a *data record*. For instance, for each option in Figure 1, an option name, its price, and constraints should be collectively extracted.

**RQ1** addresses the problem of extracting *structured variability data* by static analysis of the source code of a Web

<sup>1</sup><http://getfirebug.com/>

<sup>2</sup>Actually, building a complete feature model typically also requires consulting other sources such as documentation, expert knowledge, etc.

page. We use the *vde* patterns to specify the variability data to be extracted from Web pages. We also implemented a Web Crawler to seek, find, and extract variability data from a page given *vde* patterns, and then to transform the extracted data into structured variability data.

**RQ2.** *How to semi-automatically produce and then extract the dynamic variability content?* Web SCs are highly interactive applications and as they are executing, new content may be automatically added to the page, and existing content may be removed or changed. This research question addresses the runtime behaviour of Web SCs. We should answer sub-questions like “*how to simulate the users’ configuration actions to automatically generate dynamic content?*” and “*how to deduce variability data from the dynamically generated content?*”. We are developing a Web Crawler to deal with dynamic behaviour of Web SCs.

## III. RESEARCH METHOD

**Task 1.** *State of the art.* To answer the research questions we first need to study existing methods and approaches currently used in the context of Web data extraction and reverse engineering of Web applications. Our survey shows that none of existing approaches addresses the problem of extracting feature models from Web applications.

**Task 2.** *Design a scalable Web data extraction approach adapted for SCs.* This task answers **RQ1** and aims to provide an efficient approach to extract configuration-specific data from Web SCs. Our proposed approach is based on the notion of *vde* patterns by which the user marks data of interest to be extracted from the pages of a SC. We implemented a Web Wrapper to support the process of extracting data using the *vde* patterns.

**Task 3.** *Develop a Web Crawler.* While **Task 2** targets the static aspect of Web SCs, **Task 3** focuses on the dynamic aspect and answers **RQ2**. A Web Crawler automatically explores the configuration space (e.g., navigates through the configuration steps) or configures options in a given region of the page. The exploration and configuration actions usually add new data to the page. The Wrapper then extracts this newly added data.

**Task 4.** *Evaluation.* We plan to evaluate our approach using three criteria. First, we should evaluate the *scalability* of our approach. The main metric to be examined for scalability in our work is the *expressiveness* of the *vde* patterns to make sure that they can deal with variations in presentation and implementation of configuration-specific entities in Web pages of different SCs. Second, we need to validate the *accuracy* of the extracted data. The data validation ensures that the extracted data is the *right* data (vs. *noisy* data). It also validates that the reverse-engineered models are *complete* models, and indicates whether some data is *missing*. Third, since our approach is supervised and semi-automatic, we should measure the users’ manual effort required to use our tool.

To evaluate the expressiveness of the *vde* patterns, we plan to apply them to a number of Web SCs chosen from different industries and extract their variability data. We will report on cases that the *vde* patterns fail to deal with. To measure the manual work, we count the total number of lines of code of all *vde* patterns written for each Web SC. Moreover, the time required to inspect the source code of the pages of a Web SC, to think, and then to write appropriate patterns to extract data from that SC is another criterion to assess the manual effort. To evaluate the accuracy of the extracted data and models, we will ask the users (the domain experts or developers of the chosen Web SCs, if possible) to manually build the variability model of the target Web SC. The model generated by our approach will be compared to the one generated by the experts to validate the accuracy of the extracted models.

#### IV. PROGRESS TO DATE

We first conducted a pilot study to understand how configuration-specific data is currently presented in the pages of Web SCs. We observed that a SC usually uses a set of *templates* to format and present data in the pages. We use templates in reverse order to extract configuration-specific data encoded in the code fragments generated using those templates.

**Task 1.** Answering the two research questions requires intersecting approaches coming from three fields of study: *reverse engineering Web applications*, *Web data extraction*, and *synthesizing feature models*.

Regarding reverse engineering Web application, we investigated approaches that were proposed to reverse engineer GUIs and Web pages [4], [5], [6]. The use of these approaches to reverse engineer feature models from Web SCs requires substantial changes to their core procedures because the algorithms they implement do not consider configuration aspects (e.g., configuration semantics of GUI elements) and specific properties of Web SCs.

We studied several approaches reported in the literature that address the problem of extracting record-level (vs. field-level) data from template-generated Web pages [7], [8], [9], [10]. These approaches are adopted in particular domains, and their scalability has not been evaluated to verify if they can be reused in other domains. Another issue we noticed in the surveyed approaches is that they assume that meaningful naming the extracted data is done as a post-processing step. This poses a serious data accuracy threat for Web SCs because not all data presented in the page is configuration-specific, and therefore, another effort is required to elicit the configuration-specific data from other noisy data.

Several authors have already addressed the (semi-automatic) reverse engineering of variability models from existing artefacts [11], [12], [13], [14]. Sources include user documentation, natural language requirements, formal requirements, product descriptions, dependencies, source code,

architecture, etc. To the best of our knowledge, none of existing reverse-engineering approaches tackles the extraction of feature models from SCs.

**Task 2.** To extract variability data presented in a Web page, we need a Web Wrapper to seek, find, and extract data of interest from the unstructured (or semi-structured) Web pages and then transform it into structured data. By static analysing the source code we can extract options, their descriptive information, and constraints textually explained in the page.

Our analysis of the client-side source code of Web SCs shows that Web objects representing variability data are usually generated from a number of templates. We think of a template as an HTML code fragment that defines the structure and layout of data to be visually presented in a Web page. In a template, text elements and tag attributes are data slots filled by data items when generating the page. Each Web page consists of a number of *template instances*, which are syntactically identical fragments except for variations in values for data slots as well as minor changes to the structure. We take advantage of templates used in generating Web pages in reverse order to extract the required data. Our main proposal is the notion of *vde* pattern expressed in an HTML-like language to define templates. A *vde* pattern specifies (1) which configuration-specific data items from (2) which code fragments of the Web page will be extracted. For the former, the pattern marks text elements and attributes carrying the content of interest, and for the latter, it defines the structure of code fragments (in fact, template instances) that may contain the marked data. The Wrapper takes as input specification of a *vde* pattern and a Web page, seeks and finds code fragments in the page that *structurally* match the given pattern, and extracts as output data items from those code fragments corresponding to the marked data in the pattern.

Figure 2 presents two code fragments encoding the “Audi A1” (lines 1-4) and “Audi A3” (lines 5-8) options shown in Figure 1, and the *vde* pattern defined to extract data. In the pattern specification, *data-att-mar-key* (line 2) and *data-tex-mar-option-name* (line 4) are data marking elements denoting data items to be extracted from the matching code fragments. *data-att-mar-key*=“@key” tells the Wrapper to extract the value of the *key* attribute from the `<div class=“radioButton”>` elements of the matching code fragments. *data-tex-mar-option-name* tells the Wrapper that from the code fragments matching this pattern extract the value of the immediate child text element of the `<span class=“label”>` elements as output.

*vde* patterns address Web data extraction challenges such as *multi-valued* data, *optional* data, and *distinctive* data items. We also use a specific function called *skip* to guide the Wrapper to ignore *noisy* elements when traversing the code fragments.

The Wrapper provides a GUI through which the user

```

Code Fragments
1 <div class="radioButton" tooltip="html" rbgroup="carlineGroup" key="a1">
2   <span class="icon jqRadioButton"></span>
3   <span class="label">Audi A1</span>
4 </div>
5 <div class="radioButton" tooltip="html" rbgroup="carlineGroup" key="a3">
6   <span class="icon jqRadioButton"></span>
7   <span class="label">Audi A3</span>
8 </div>

Pattern Specification
1 <pattern data-att-met-pattern-type="option" data-att-met-pattern-name="options">
2   <div class="radioButton" data-att-mar-key="@key">
3     <span class="icon jqRadioButton"></span>
4     <span class="label">data-text-mar-option-name</span>
5   </div>
6 </pattern>

```

Figure 2. an example *vde* pattern

interactively defines the required *vde* patterns. It uses a *source code pattern matching* algorithm to find code fragments structurally matching the given *vde* patterns, and then extracts data marked in the patterns from those code fragments. The output data is represented in an XML file in a predefined data model. Once all the data is extracted and represented in the XML file, missing data is added, and irrelevant data is removed, we can generate a feature model. Specifically, we rely on the *Text-based Variability Language (TVL)* to represent feature models [3].

**Task 3.** This task is currently in progress. At present, the Crawler is able to explore simple configuration spaces. It can linearly navigate from one configuration step to another. The exploration action may dynamically add new content to the current page, e.g., by activating a step its contained options are made available in the page. The Web Wrapper is responsible to elicit and extract configuration-specific data from this newly generated content.

Regrading configuration actions, the Crawler is able to simulate the selection of items from a list box and the click on clickable elements (e.g., radio buttons). When an option is given a new value and one or more constraints apply, the SC may follow three patterns. First, it loads new options to the page that are consistent with the configured options and removes the irrelevant options. For instance, the selection of an option in the *Model line* group in Figure 1 loads its relevant options to the *Body style* group. Second, configuring an option changes the configuration state (selected, deselected, deactivated, etc.) of other impacted options. For instance, the selection of “Privacy glass” implies the deselection of “Twin-pane UV and heat-insulating glass” in Figure 1. And third, the SC presents a window to the user and alerts her about a conflict or asks her to confirm a change before altering the impacted options. Therefore, this window contains configuration-specific content and must be analysed to extract the data. The Web Wrapper and Crawler collaborate together to deal with these cases.

**Task 4.** We have started evaluating the expressiveness of *vde* patterns. For each Web SC, we define and apply a minimum set of patterns to specify templates used to generate its Web pages, and see how many of those templates are covered.

## V. CONCLUSION

This PhD thesis aims at reverse engineering feature models from Web SCs. The first contribution of our work is

proposing the notion of *variability data extraction* pattern in an HTML-like language to mark configuration-specific data to be extracted from Web pages of a SC. We also proposed a source code pattern matching algorithm to find code fragments in the source code structurally matching a given pattern. The second contribution of this PhD study is dynamic analysis of a Web page to systematically produce dynamic content and then extract from that the configuration-specific data. Finally, the third expected contribution of this thesis is delivering a set of tools to reverse engineer Web SCs. After our empirical study of Web SCs [1], this work takes a second step towards systematically re-engineering Web SCs [2].

## REFERENCES

- [1] E. K. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans, “The anatomy of a sales configurator: An empirical study of 111 cases,” in *CAiSE’13*, 2013.
- [2] Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans, “Towards more reliable configurators: A re-engineering perspective,” in *PLEASE’12*, 2012.
- [3] A. Classen, Q. Boucher, and P. Heymans, “A text-based approach to feature modelling: Syntax and semantics of tvl,” *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1130–1143, 2011.
- [4] J. Vanderdonckt, L. Bouillon, and N. Souchon, “Flexible reverse engineering of web pages with *vauquista*,” in *WCRE’01*. IEEE, 2001, pp. 241–248.
- [5] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, “Gui-tar: an innovative tool for automated testing of gui-driven software,” *Automated Software Engineering*, pp. 1–41, 2013.
- [6] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, “Reverse engineering web applications: the WARE approach,” *J. Softw. Maint. Evol.*, vol. 16, no. 1-2, pp. 71–101, Jan. 2004.
- [7] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender, “Automatic web news extraction using tree edit distance,” ser. WWW ’04. ACM, 2004, pp. 502–511.
- [8] A. Arasu and H. Garcia-Molina, “Extracting structured data from web pages,” ser. SIGMOD. ACM, 2003, pp. 337–348.
- [9] V. Crescenzi, G. Mecca, and P. Merialdo, “Roadrunner: Towards automatic data extraction from large web sites,” ser. VLDB ’01, 2001, pp. 109–118.
- [10] B. Liu, R. Grossman, and Y. Zhai, “Mining data records in web pages,” ser. KDD ’03. ACM, 2003, pp. 601–606.
- [11] N. Weston, R. Chitchyan, and A. Rashid, “A framework for constructing semantically composable feature models from natural language requirements,” in *SPLC’09*. ACM, 2009, pp. 211–220.
- [12] I. John, “Capturing product line information from legacy user documentation,” in *SPLC*. Springer, 2006, pp. 127–159.
- [13] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, “Reverse Engineering Architectural Feature Models,” in *ECISA’11*. Springer, 2011, pp. 220–235.
- [14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “Reverse engineering feature models,” in *ICSE’11*. ACM, 2011, pp. 461–470.

# Author Index

Abbasi, Ebrahim Khalil.....	586	Dasgupta, Tathagata.....	320
Abdeen, Hani.....	80	Decker, Michael John.....	516
Adams, Bram.....	20	Delange, Julien.....	400
Alba, Enrique.....	404	Demeyer, Serge.....	384
Alhindawi, Nouh.....	300, 416	Di Penta, Massimiliano.....	210, 230, 280, 484
Anand, Kapil.....	90	Dietrich, Jens.....	160
Antoniol, Giuliano.....	428	Dit, Bogdan.....	320, 330, 408
Asadullah, Allahbaksh.....	372	Do, Hyunsook.....	1
Asadullah, Allahbaksh M. ....	524	Dragan, Natalia.....	300
Asaduzzaman, Muhammad.....	230, 484	Duchien, Laurence.....	388
Bacchelli, Alberto.....	408	Dumas, Marlon.....	528
Bandara, Wathsala.....	452	Ebert, Achim.....	492
Barthel, Henning.....	492	Ebert, Felipe.....	448
Barua, Rajeev.....	90	Egyed, Alexander.....	404
Basavaraju, M. ....	524	Elwazeer, Khaled.....	90
Bauer, Ken.....	504	Engström, Emelie.....	20
Bavota, Gabriele.....	210, 280	Fails, Jerry Alan.....	512
Bazelli, Blerina.....	460	Feldt, Robert.....	480
Bazrafshan, Saman.....	50	Ferre, Vincenzo.....	260
Beck, Fabian.....	560	Ferrer, Javier.....	404
Bernat, Andrew R. ....	250	Filkov, Vladimir.....	340
Bhat, Vasudev.....	372	Flora, Parminder.....	110
Binkley, Dave.....	408, 432	Fokaefs, Marios.....	444
Bos, Jeroen van den .....	520	Fontana, Francesca Arcelli.....	260, 396
Brada, Premek.....	500	Francis, Patrick.....	436
Brinkkemper, Sjaak.....	220	Fritz, Thomas.....	290
Bruegge, Bernd.....	464	Garcia, Alessandro.....	508
Burge, Janet.....	432	Garousi, Vahid.....	428
Canfora, Gerardo.....	210, 280	Gharehyazie, Mohammad.....	340
Carruth, Chandler.....	548	Gobert, Maxime.....	472
Castor, Fernando.....	448	González, Marco.....	400
Castrejón, Juan.....	496	Gotlieb, Arnaud.....	540
Chaikalis, Theodore.....	392	Grechanik, Mark.....	320
Chatzigeorgiou, Alexander.....	392	Guan, Haibing.....	100
Chaudron, Michel R.V. ....	140	Haiduc, Sonia.....	452
Chicano, Francisco.....	404	Hamou-Lhadj, Abdelwahab.....	536
Cleve, Anthony.....	472	Harder, Jan.....	30
Collard, Michael L. ....	300, 416, 516	Harris, Ian.....	432
Collet, Christine.....	496	Hassan, Ahmed E. ....	110, 270, 350
Compton, Lawrence B. ....	552	Hebig, Regina.....	432
Conradi, Reidar.....	420	Hegedus, Péter.....	578
Cruz, Ana Erika Camargo.....	468	Hesse, Tom-Michael.....	464
Cruzes, Daniela Soares.....	420	Hill, Emily.....	376, 408, 432, 512

# Author Index

Hindle, Abram.....	460	Maletic, Jonathan I. ....	300, 416, 516
Holy, Lukas.....	500	Mallet, Greg.....	512
Hou, Daqing.....	60	Mäntylä, Mika V. ....	20, 396
Hua, Jing.....	440	Mao, Xiaoguang.....	180
Huang, Shiqiu.....	100	Marcus, Andrian.....	452
Iida, Hajimu.....	468	Marijan, Dusica.....	540
Jain, Nikita.....	524	Marino, Alessandro.....	260, 396
Jarzabek, Stan.....	40	Martenka, Pawel.....	260
Jasper, Daniel.....	548	Martinez, Matias.....	388
Jezek, Kamil.....	500	Mascaro, Angelica.....	556
Jiang, Zhen Ming.....	110	Matichuk, Isaac.....	504
Karus, Siim.....	412	McCartin, Catherine.....	160
Keromytis, Angelos.....	90	McDonnell, Tyler.....	70
Keszocze, Oliver.....	432	Mehrfard, Hossein.....	130
Khan, Taimur.....	492	Mendonça, Manoel.....	508
Khomh, Foutse.....	20, 270, 350	Meng, Xiaozhu.....	250
Kim, Miryung.....	70	Meqdadi, Omar.....	416
Klimek, Manuel.....	548	Miller, Barton P. ....	250
Klint, Paul.....	120	Miller, William L. ....	552
Kolbah, Bojana.....	130	Minelli, Roberto.....	476
Köppe, Christian.....	220	Mo, Lingfeng.....	60
Koschke, Rainer.....	50	Mockus, Audris.....	350
Kotha, Aparna.....	90	Monperrus, Martin.....	388
Krishnan, Giri Panamoottil.....	360	Moonen, Leon.....	424
Kruchten, Philippe.....	400	Moreno, Laura.....	452
Kwon, Young-Woo.....	170, 582	Moritz, Evan.....	320, 330
Labiche, Yvan.....	130	Muddu, Basavaraju.....	372
Landman, Davy.....	120	Müller, Sebastian C. ....	290
Lanza, Michele.....	476	Nagappan, Meiyappan.....	110, 270
Larsson, Alf.....	536	Nasser, Mohamed.....	110
Lawrie, Dawn.....	408, 432	Nguyen, Hoan Anh.....	150, 456
Le, Tien-Duy B. ....	310, 364, 380	Nguyen, Hung Viet.....	150, 456
Lei, Yan.....	180	Nguyen, Tien N. ....	150, 456
Leotta, Maurizio.....	428	Nguyen, Tung Thanh.....	150, 456
Li, Xiang.....	240	Nord, Robert L. ....	400
Liggemeyer, Peter.....	492	Novais, Renato L. ....	508
Ligu, Elvis.....	392	Nunes, Camila.....	508
Linares-Vásquez, Mario.....	330	Oliveto, Rocco.....	210, 280, 408
Ling, Charles X. ....	240	Osman, Mohd Hafeez.....	140
Lo, David.....	200, 310, 364, 380	Oü, Plumbr.....	544
Lopez-Herrejon, Roberto E. ....	404	Oyetoyan, Tosin Daniel.....	420
Lozano, Rafael.....	496	Ozkaya, Ipek.....	400
Maes, Jérôme.....	472	Paech, Barbara.....	464

# Author Index

Paige, Richard F. ....	480	Silva, Fabio Q.B. ....	556
Panichella, Sebastiano.....	210, 280	Slankas, John.....	432
Peng, Xin.....	40	Smithson, Matthew.....	90
Perez, Javier.....	384	Snajberk, Jaroslav.....	500
Pérez, Javier.....	572	Soetens, Quinten David.....	384
Petersen, Kai.....	20	Šor, Vladimir.....	544
Pinzger, Martin.....	368	Sözer, Hasan.....	532
Pollock, Lori.....	376	Srirama, Satish Narayana.....	544
Poshyvanyk, Denys.....	320, 330	Stroulia, Eleni.....	444, 460, 504
Posnett, Daryl.....	340	Sun, Chengnian.....	200
Poulding, Simon.....	480	Syer, Mark D. ....	110
Preining, Norbert.....	468	Taba, Seyyed Ehsan Salamati.....	270
Pruijt, Leo.....	220	Thung, Ferdian.....	380
Qi, Yuhua.....	180	Tian, Yuan.....	200
Qi, Zhengwei.....	100	Tilevich, Eli.....	170
Qian, Wenyi.....	40	Toroi, Tanja.....	11
Rajlich, Václav.....	440	Treier, Tarvo.....	544
Raninen, Anu.....	11	Tsantalis, Nikolaos.....	360
Ray, Baishakhi.....	70	Väätäinen, Lauri.....	11
Reed, Karl.....	432	van der Putten, Peter.....	140
Ricca, Filippo.....	428	van der Storm, Tijs.....	520
Risi, Michele.....	190	Vargas-Solar, Genoveva.....	496
Rocha, Fabio.....	504	Venkataramani, Rahul.....	372
Roehm, Tobias.....	464	Vijay-Shanker, K. ....	376
Roldan-Vega, Manuel.....	512	Vinju, Jurgen.....	120
Romano, Daniele.....	368	Walter, Bartosz.....	260
Rose, Louis M. ....	480	Wan, Zhanyong.....	548
Roy, Chanchal K. ....	230, 484, 488	Wang, Huaimin.....	240
Ruhe, Guenther.....	428	Wang, Shaowei.....	364
Saar, Tõnis.....	528	Wang, Tao.....	240
Saha, Ripon K. ....	488	Wang, Yin.....	100
Sahraoui, Houari.....	80	Weber, Jens.....	472
Sangwan, Raghvinder S. ....	400	Williams, Laurie.....	436
Santos, Andre L.M. ....	556	Williams, William R. ....	250
Scanniello, Giuseppe.....	190	Woodmansee, Bruce L. ....	552
Schneider, Kevin A. ....	230, 484, 488	Wright, Hyrum K. ....	548
Schwartz, Amanda.....	1	Xiang, Chengcheng.....	100
Semenenko, Nataliia.....	528	Xin, Rui.....	100
Sen, Sagar.....	540	Xing, Zhenchang.....	40
Shah, Syed Muhammad Ali.....	160	Yamashita, Aiko.....	424, 566
Shata, Osama.....	80	Ygeionomakis, Nikolaos.....	392
Shepherd, David.....	376	Yin, Gang.....	240
Siebra, Claurton.....	556	Yüksel, Ulas.....	532

# Author Index

Zanoni, Marco.....	396	Zhi, Junji.....	428
Zhang, Feng.....	350	Zou, Peng.....	240
Zhao, Wenyun.....	40	Zou, Ying.....	270, 350
Zheng, Yudi.....	100		

**T&C Board Vice President**

Paul R. Croll

*Computer Sciences Corporation***IEEE Computer Society Staff**Evan Butterfield, *Director of Products and Services*Lynne Harris, *CMP, Senior Manager, Conference Support Services*Alicia Stickley, *Senior Manager, Publishing Operations*Silvia Ceballos, *Manager, Conference Publishing Services*Patrick Kellenberger, *Supervisor, Conference Publishing Services***IEEE Computer Society Publications**

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at <http://www.computer.org/portal/site/store/index.jsp> for a list of products.

**IEEE Computer Society *Conference Publishing Services* (CPS)**

The IEEE Computer Society produces conference publications for more than 300 acclaimed international conferences each year in a variety of formats, including books, CD-ROMs, USB Drives, and on-line publications. For information about the IEEE Computer Society's *Conference Publishing Services* (CPS), please e-mail: [cps@computer.org](mailto:cps@computer.org) or telephone +1-714-821-8380. Fax +1-714-761-1784. Additional information about *Conference Publishing Services* (CPS) can be accessed from our web site at: <http://www.computer.org/cps>

*Revised: 18 January 2012*



**CPS Online** is our innovative online collaborative conference publishing system designed to speed the delivery of price quotations and provide conferences with real-time access to all of a project's publication materials during production, including the final papers. The **CPS Online** workspace gives a conference the opportunity to upload files through any Web browser, check status and scheduling on their project, make changes to the Table of Contents and Front Matter, approve editorial changes and proofs, and communicate with their CPS editor through discussion forums, chat tools, commenting tools and e-mail.

The following is the URL link to the **CPS Online** Publishing Inquiry Form:

<http://www.computer.org/portal/web/cscps/quote>

# *Proceedings*

---

## **2013 IEEE International Conference on Software Maintenance**





# *Proceedings*

---

## **2013 IEEE International Conference on Software Maintenance**

22–28 September 2013  
Eindhoven, The Netherlands



Los Alamitos, California  
Washington • Tokyo



All rights reserved.

*Copyright and Reprint Permissions:* Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

*The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.*

IEEE Computer Society Order Number E4981  
BMS Part Number CFP13079-ART  
ISBN 978-0-7685-4981-1

*Additional copies may be ordered from:*

IEEE Computer Society  
Customer Service Center  
10662 Los Vaqueros Circle  
P.O. Box 3014  
Los Alamitos, CA 90720-1314  
Tel: + 1 800 272 6657  
Fax: + 1 714 821 4641  
<http://computer.org/cspress>  
[csbooks@computer.org](mailto:csbooks@computer.org)

IEEE Service Center  
445 Hoes Lane  
P.O. Box 1331  
Piscataway, NJ 08855-1331  
Tel: + 1 732 981 0060  
Fax: + 1 732 981 9667  
[http://shop.ieee.org/store/  
customer-service@ieee.org](http://shop.ieee.org/store/customer-service@ieee.org)

IEEE Computer Society  
Asia/Pacific Office  
Watanabe Bldg., 1-4-2  
Minami-Aoyama  
Minato-ku, Tokyo 107-0062  
JAPAN  
Tel: + 81 3 3408 3118  
Fax: + 81 3 3408 3553  
[tokyo.ofc@computer.org](mailto:tokyo.ofc@computer.org)

*Individual paper REPRINTS may be ordered at: <[reprints@computer.org](mailto:reprints@computer.org)>*

Editorial production by Lisa O'Conner



**IEEE Computer Society  
Conference Publishing Services (CPS)**

<http://www.computer.org/cps>

## Foreword

Welcome to ICSM 2013, the 29th IEEE International Conference on Software Maintenance in Eindhoven, The Netherlands. Eindhoven, recently called by Forbes “the most innovative city in the world”, is located in the province of North Brabant in the south of the Netherlands. The city counts 213,809 inhabitants (as of January 1st, 2010), which makes it one of the five largest cities of The Netherlands. Eindhoven is well known for its modern art, design and technology, as illustrated by this year’s ICSM logo.

ICSM 2013 as a whole is the result of the tremendous effort of the 20 people composing its Organising Committee, who took care of everything from finding a location to making sure the food is delicious. For their effort, we thank them very much for having made our stay in Eindhoven such an enjoyable moment! Organising a conference is often done in addition to all the other duties of a professor with no prospect of rewards but the acknowledgment of the attendees and of the community as a whole for a job well-done. It also requires solving dozens of small but time-consuming problems while balancing the books and attending to the attendees’ needs. This year, attendees and presenters were particularly pampered. Thank you!

The ICSM technical program is the result of the amazing work of the 67 dedicated members of its Program Committee (PC), chaired by its two Program Co-chairs. The PC is relatively well balanced, both from a gender perspective (14 females out of 67 members, i.e., 21%) and from a geographical perspective (40% from Europe, 40% from North-America, 15% from Asia, and the remaining 5% from elsewhere). The PC members received help from many additional reviewers, thus further enforcing the mission of ICSM: it is not only a place to share knowledge but also a place to learn new skills. We would like to thank all reviewers warmly and commend them for the timeliness, thoroughness, and remarkable quality of their reviews.

From the initial 195 submitted abstracts, the PC collectively reviewed 163 high-quality papers, writing more than 400 reviews, 600 comments, and 100 revisions. Finally, it accepted 36 papers, corresponding to an acceptance rate of 22%. Of these 36 papers, six have been invited for a special issue of the Springer journal of Empirical Software Engineering. One of these papers received the best paper award during the conference banquet.

We have put together an exciting program that spans many aspects of software maintenance, including the usual suspects: reverse engineering, code cloning, defect management, and runtime analysis but also “emerging” topics such as programming interfaces, software ownership, and the context of software development. These proceedings contain the 36 full research papers spread across eight 3-paper sessions and six 2-paper sessions.

The conference program also features two keynotes by leaders in the field of software engineering: Jeff Magee of Imperial College London (UK), who presents an intrinsic definition in software architecture evolution, and Michael Feathers from DepthFirst, who reflects upon the useful life of software. In addition to the technical sessions and the keynotes, the program includes two special sessions: one celebrating the most influential paper of 2003 and another discussing opportunities for open access to research data, tools, and findings. It also includes an Industry Track with 9 papers (out of 18 submissions) presenting industrial results in the field of software maintenance and evolution, an Early Research

Achievements Track with 29 papers containing novel, emerging research results (out of 70 submissions), a Tool Demo Track showcasing 12 tools (out of 22 submissions), and a Doctoral Symposium during which 6 (ex-) Ph.D. students presented their work before the entire ICSM audience and 9 other Ph.D. students presented in front of a panel of top researchers in the field (out of 21 submissions).

Co-located with ICSM 2013 are the 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), the 1st IEEE Working Conference on Software Visualization (VISSOFT), the 15th IEEE International Symposium on Web Systems Evolution (WSE), and the 1st International Workshop on Communicating Business Process and Software Models (CPSM).

This program and the conference would not have been possible without the financial, logistic, and material support of our sponsors: the IEEE Computer Society, the Technical Council on Software Engineering, and, of course, the IEEE. It was also supported by many other organisations; in alphabetical order: ASML, Eindhoven University of Technology, Fontys University of Applied Sciences, Google, Grammatech, IBM, JACQUARD, and The Netherlands Organisation for Scientific Research (NWO). We warmly thank our sponsors, supporters and the Steering Committee for making ICSM possible.

All-in-all, you will certainly discover in these proceedings new techniques, technologies, and topics of interest and, above all, food for thought for the coming years of research.

Enjoy the reading!

Alexander Serebrenik

**General Chair**

*Eindhoven University of Technology*

*The Netherlands*

Tom Mens

**Program Co-chair**

*University of Mons*

*Belgium*

Yann-Gaël Guéhéneuc

**Program Co-chair**

*Polytechnique Montréal*

*Canada*

# Organising Committee

## General Chair

Alexander Serebrenik, *Eindhoven University of Technology, The Netherlands*

## Program Co-Chairs

Tom Mens, *University of Mons, Belgium*

Yann-Gaël Guéhéneuc, *Polytechnique de Montréal, Canada*

## ERA Program Co-Chairs

Romain Robbes, *University of Chile, Chile*

Bram Adams, *Polytechnique de Montréal, Canada*

## Industry Track Chair

Joost Visser, *Software Improvement Group, The Netherlands*

## Financial Chair

Chanchal K. Roy, *University of Saskatchewan, Canada*

## Doctoral Symposium Co-Chairs

Lori Pollock, *University of Delaware, USA*

Tibor Gyimothy, *University of Szeged, Hungary*

## Tools Track Co-chairs

Mark van den Brand, *Eindhoven University of Technology, The Netherlands*

Anthony Cleve, *University of Namur, Belgium*

## Publicity Co-chairs

Natalia Dragan, *University of Akron, USA*

Veronica Bauer, *Technische Universität München, Germany*

## Local Arrangement Co-Chairs

Martijn Klabbers, *Eindhoven University of Technology, The Netherlands*

Margje Mommers, *Eindhoven University of Technology, The Netherlands*

Sacha Claessens, *Eindhoven University of Technology, The Netherlands*

## Proceedings Chair

Bogdan Vasilescu, *Eindhoven University of Technology, The Netherlands*

## Web Chair

Minhaz F. Zibran, *University of Saskatchewan, Canada*

**Social Media Chair**

Felienne Hermans, *Delft University of Technology, The Netherlands*

**Student Volunteers Chair**

Yanja Dajsuren, *Eindhoven University of Technology, The Netherlands*

# Technical Program Committee

## Program Committee for Research Track

Alexander Serebrenik, *Eindhoven University of Technology, The Netherlands (Chair)*

Yann-Gaël Guéhéneuc, *Polytechnique de Montréal, Canada (Co-chair)*

Tom Mens, *University of Mons, Belgium (Co-chair)*

Paul Anderson, *Grammatech, USA*

Nicolas Anquetil, *INRIA Lille and Université Lille 1, France*

Doo-Hwan Bae, *KAIST, Korea*

Ayse Basar Bener, *Ryerson University, Canada*

Christian Bird, *Microsoft Research, USA*

Goetz Botterweck, *Lero / University of Limerick, Ireland*

Mark van den Brand, *Eindhoven University of Technology, The Netherlands*

Andrea Capiluppi, *Brunel University, UK*

Mariano Ceccato, *Fondazione Bruno Kessler, Italy*

Alexander Chatzigeorgiou, *University of Macedonia, Greece*

Jane Cleland-Huang, *DePaul University, USA*

Anthony Cleve, *University of Namur, Belgium*

Michael Collard, *University of Akron, USA*

James Cordy, *Queen's University, Canada*

Massimiliano Di Penta, *University of Sannio, Italy*

Danny Dig, *University of Illinois at Urbana-Champaign, USA*

Laurence Duchien, *INRIA Lille, France*

Jean-Rémy Falleri, *Université Bordeaux 1, France*

Rudolf Ferenc, *University of Szeged, Hungary*

Malcom Gethers, *University of Maryland, USA*

Yossi Gil, *Technion - Israel Institute of Technology, Israel*

Michael Godfrey, *University of Waterloo, Canada*

Mark Grechanik, *University of Illinois at Chicago, USA*

Rajiv Gupta, *University of California Riverside, USA*

Mark Harman, *University College London, UK*

Pedro Rangel Henriques, *Universidade do Minho, Portugal*

Felienne Hermans, *Delft University of Technology, The Netherlands*

Abram Hindle, *University of Alberta, Canada*

Zhang Hongyu, *Tsinghua University, China*

Daqing Hou, *Clarkson University, USA*

Marianne Huchard, *Université Montpellier 2, France*

Katsuro Inoue, *Osaka University, Japan*

Huzefa Kagdi, *Wichita State University, USA*

Foutse Khomh, *Queen's University, Canada*

Jens Knodel, *Fraunhofer Institute for Experimental Software Engineering, Germany*

Dawn Lawrie, *Loyola University Maryland, USA*

Leen Lambers, *Hasso-Plattner-Institut, Universität Potsdam, Germany*

Byungjeong Lee, *University of Seoul, Korea*

Zheng Li, *Beijing University of Chemical Technology, China*  
David Lo, *Singapore Management University, Singapore*  
Jonathan I. Maletic, *Kent State University, USA*  
Andrian Marcus, *Wayne State University, USA*  
Cristina Marinescu, *Politehnica University of Timisoara, Romania*  
Hong Mei, *Peking University, China*  
Leon Moonen, *Simula Research Laboratory, Norway*  
Tien Nguyen, *Iowa State University, USA*  
Oscar Nierstrasz, *University of Bern, Switzerland*  
Rocco Oliveto, *University of Molise, Italy*  
Ekaterina Pek, *University of Koblenz, Germany*  
Martin Pinzger, *University of Klagenfurt, Austria*  
Lori Pollock, *University of Delaware, USA*  
Steven Reiss, *Brown University, USA*  
Brian Robinson, *ABB Corporate Research, USA*  
Gregorio Robles, *Universidad Rey Juan Carlos, Spain*  
Chanchal Roy, *University of Saskatchewan, Canada*  
Giuseppe Scanniello, *University of Basilicata, Italy*  
Carolyn Seaman, *University of Maryland, USA*  
Janet Siegmund, *Otto-von-Guericke-Universität Magdeburg, Germany*  
Kobayashi Takashi, *Tokyo Institute of Technology, Japan*  
Ewan Tempero, *University of Auckland, New Zealand*  
Suresh Thummalapenta, *IBM Research, India*  
Marco Tulio Valente, *Universidade Federal de Minas Gerais, Brazil*  
Ragnhild Van Der Straeten, *VUB, Belgium*  
Antonio Vallecillo, *University of Malaga, Spain*  
Michel Wermelinger, *The Open University, UK*  
Lu Zhang, *Peking University, China*  
Lingming Zhang, *University of Texas at Austin, USA*  
Thomas Zimmermann, *Microsoft Research, USA*

**Program Committee for Tool Demo Track**

Anthony Cleve, *University of Namur, Belgium (co-chair)*  
Mark van den Brand, *Eindhoven University of Technology, The Netherlands (co-chair)*  
Mathieu Acher, *INRIA Rennes, France*  
Amir Aryani, *RMIT University, Australia*  
Alberto Bacchelli, *University of Lugano, Switzerland*  
Marcus Denker, *INRIA Lille, France*  
Alessandro Garcia, *Pontifical Catholic University of Rio de Janeiro, Brazil*  
Malcom Gethers, *College of William and Mary, USA*  
Sonia Haiduc, *Wayne State University, USA*  
Jan Harder, *University of Bremen, Germany*  
Adrian Johnstone, *Royal Holloway University of London, UK*  
Yasutaka Kamei, *Kyushu University, Japan*

Foutse Khomh, *Queen's University, Canada*  
Jens Knodel, *Fraunhofer Institute for Experimental Software Engineering, Germany*  
Mircea Lungu, *University of Bern, Switzerland*  
Xin Peng, *Fudan University, China*  
Juergen Rilling, *Concordia University, Canada*  
Wenhua Wang, *Marin Software, USA*

#### **Program Committee for Doctoral Symposium**

Tibor Gyimóthy, *University of Szeged, Hungary (co-chair)*  
Lori Pollock, *University of Delaware, USA (co-chair)*  
Denys Poshyvanyk, *College of William and Mary, USA*  
Eleni Stroulia, *University of Alberta, Canada*  
Paolo Tonella, *Fondazione Bruno Kessler, Italy*  
Alexander Serebrenik, *Eindhoven University of Technology, The Netherlands*  
Mark van den Brand, *Eindhoven University of Technology, The Netherlands*  
László Vidács, *University of Szeged, Hungary*

#### **Program Committee for ERA Track**

Amir Aryani, *RMIT University, Australia*  
Gabriele Bavota, *University of Salerno, Italy*  
Árpád Beszédes, *University of Szeged, Hungary*  
Yingnong Dang, *Microsoft Research Asia, China*  
Fabio Q. B. Da Silva, *Federal University of Pernambuco, Brazil*  
Chen Fu, *Accenture Technology Labs, USA*  
Alessandro Garcia, *Pontifical Catholic University of Rio de Janeiro, Brazil*  
Nils Göde, *Universität Bremen, Germany*  
Sonia Haiduc, *Wayne State University, USA*  
Yoshiki Higo, *Osaka University, Japan*  
Emily Hill, *Montclair State University, USA*  
Sungwon Kang, *KAIST, South Korea*  
Sègla Kpodjedo, *Benchmark Consulting, Canada*  
Nicholas A. Kraft, *University of Alabama, USA*  
Jens Krinke, *University College London, UK*  
Alessandro Marchetto, *Fiat Research Center, Italy*  
Meiyappan Nagappan, *Queen's University, Canada*  
Takayuki Omori, *Ritsumeikan University, Japan*  
David Röthlisberger, *Universidad de Chile, Chile*  
Emad Shihab, *Rochester Institute of Technology, USA*  
Ladan Tahvildari, *University of Waterloo, Canada*  
Eli Tilevich, *Virginia Tech, USA*  
Porfirio Tramontana, *University of Naples Federico II, Italy*  
Nikolaos Tsantalis, *Concordia University, Canada*  
Burak Turhan, *University of Oulu, Finland*  
László Vidács, *University of Szeged, Hungary*

Neil Walkinshaw, *University of Leicester, UK*  
Anja Guzzi, *Delft University of Technology, The Netherlands*  
Christian Hammer, *Saarland University, Germany*  
Lile Hattori, *University of Lugano, Switzerland*  
Angela Lozano, *Université Catholique de Louvain, Belgium*

**Program Committee for Industry Track**

Joost Visser, *Software Improvement Group, The Netherlands (Chair)*  
Andreas Winter, *Carl von Ossietzky University, Germany*  
Mark van den Brand, *Eindhoven University of Technology, The Netherlands*  
Tiago Alves, *OutSystems, USA*  
Petra Heck, *Fontys University of Applied Sciences, The Netherlands*  
Merijn de Jonge, *Sorama, The Netherlands*  
Lodewijk Bergmans, *University of Twente, The Netherlands*  
Andy Zaidman, *Delft University of Technology, The Netherlands*  
Eric Bouwers, *Software Improvement Group, The Netherlands*  
Rudolf Ferenc, *University of Szeged, Hungary*  
Robert Nord, *Carnegie Mellon Software Engineering Institute, USA*  
Ariadi Nugroho, *Software Improvement Group, The Netherlands*  
Mark Grechanik, *University of Illinois at Chicago, USA*  
Grace Lewis, *Carnegie Mellon Software Engineering Institute, USA*  
Elmar Juergens, *Technische Universitaet Muenchen, Germany*  
Istvan Siket, *University of Szeged, Hungary*  
Carl Worms, *Credit Suisse, Switzerland*  
Serge Demeyer, *Universiteit Antwerpen, Belgium*  
James Terwilliger, *Microsoft Corporation, USA*  
Paul Anderson, *GammaTech Inc., USA*  
Michaela Greiler, *Microsoft Corporation, USA*  
Carola Lilienthal, *CI WPS GmbH/University of Hamburg, Germany*  
Thomas Dean, *Queens University, Canada*  
Tom Mens, *University of Mons, Belgium*

## Sub-Reviewers

Hani Abdeen	Scott Grant	Floréal Morandat
Jongsun Ahn	Sonia Haiduc	Laura Moreno
Hwi Ahn	Ah-Rim Han	Marco Mori
Manar Alalfi	Shi Han	Dan Mosora
Nouh Alhindawi	Dan Hao	Csaba Nagy
Saleh Alnaeli	David Hartveld	Christian Newman
Farouq Alomari	Shinpei Hayashi	Hoan Nguyen
Muhammad Asaduzzaman	Regina Hebig	Tung Nguyen
Nabil Bachir Djarallah	Péter Hegedűs	Yitao Ni
Motahareh Bahrami Zanjani	Yoshiki Higo	Jens Nicolay
Gergő Balogh	Stephan Hildebrandt	Semih Okur
Eiji Barbosa	André Hora	Nuno Oliveira
Brian Bartman	Md. Kamal Hossen	Haidar Osman
Olga Baysal	Ji-Min Hwa	Jihun Park
Thomas Beyhl	Takashi Ishio	Taehyun Park
Xavier Blanc	Syed Islam	Andreas Pleuss
Dragan Bosnacki	Judit Jász	Taiana Pontes
Caius Brindescu	Niels Joncheere	Masud Rahman
Simon Butler	Howell Jordan	Daniele Romano
Bruno Cafeo	Changsup Keum	Dominik Rost
Gul Calikli	Jingyu Kim	Serguei Roubtsov
Andrea Caracciolo	Sergiy Kolesnikov	Avigit Saha
Fernando Castor	Oleksii Kononenko	Ronnie Santos
Andrei Chis	Jan Kurš	Theodoor Scholte
Elder Cirilo	Gergely Ladányi	Lajos Schrettner
Mihai Codoban	Jannik Laval	Sandro Schulze
Lauro Costa	Seonah Lee	Cory Schutz
Daniela Da Cruz	Jihyun Lee	Niko Schwarz
Yanja Dajsuren	Ge Li	Yeong-Seok Seo
Coen De Roover	Mario Linares Vásquez	Abdelhak Seriai
Michael Decker	Xuanzhe Liu	Donghwan Shin
Tezcan Dilshener	Alda Lopes Gancarski	István Siket
Steve Dodier-Lazaro	Mircea Lungu	Zéphyrin Soh
Johannes Dyck	Zhiyi Ma	Reinout Stevens
Anne Etien	Isela Macia	Yanchun Sun
Jee Eunkyoungh	Matías Martínez	Jeffrey Svajlenko
Eduardo Figueiredo	Mohammad Masudur Rahman	Cédric Teyton
Thomas Forster	Omar Meqdadi	Chouki Tibermacine
Matthieu Foucault	Sebastian Middeke	Zoltán Tóth
Lajos Jenő Fulop	Manishankar Mondal	Maria João Varanda Pereira
Gregor Gabrysiak	Martin Monperrus	Andrei Varanovich
Paul Geesaman	João Monteiro	Tom Verhoeff
Tamás Gergely	Lionel Montrieux	Thomas Vogel
Mattijs Ghijsen	Kevin Moore	Yan Wang

Wei Wang  
Xiaoyin Wang  
Sebastian Wätzoldt  
Balthasar Weitzel  
Erwann Wernli

Rongxin Wu  
Jifeng Xuan  
Suan Yong  
Norihito Yoshida  
Ying Zhang

Wei Zhang  
Haiyan Zhao  
Bo Zhou

# Keynotes

## **Intrinsic Definition in Software Architecture Evolution**

Jeff Magee, *Imperial College London, UK*

*Session Chair: Tom Mens*

Incremental change is an integral part of both the initial development and the subsequent evolution and maintenance of large complex software systems. The talk discusses both, the requirements for and the design of, an approach that captures this incremental change in the definition of software architecture. The predominate advantage in making the definition of evolution intrinsic to architecture description is in permitting a principled and manageable way of dealing with unplanned change and extension.

Intrinsic definition facilitates decentralized evolution in which software is extended and evolved by multiple independent developers. The objective is an approach which permits unplanned extensions to be deployed to end users with the same facility that plugin extensions are currently added to systems with planned extension points. The talk advocates a model-driven approach in which architecture definition is used to directly construct both initial implementations and extensions/modification to these implementations.

Existing approaches to dealing with the evolution of software architecture have focused on extending configuration management systems with architectural concepts. The intrinsic definition approach does not replace the need for version control provided by configuration management systems, rather it is intended as a way of capturing architectural change that can both co-exist with and exploit the use of industry standard configuration management systems.

An implementation of intrinsic evolution definition in Backbone is presented. Backbone is a development of the Darwin architectural description language (ADL) and it has both a textual and a UML2 based graphical representation. The talk uses Backbone and it's supporting toolset Evolve to illustrate basic concepts through simple examples and reports experience in applying the language and tools to larger examples.

### **Short Biography of Keynote Speaker, Jeff Magee**

Professor Jeff Magee is Principal of the Faculty of Engineering at Imperial College. He is a graduate in electrical engineering and holds MSc and PhD degrees in Computing Science from Imperial College London. He was appointed Head of the Department of Computing at Imperial in 2004 and Deputy Principal (Research) of the Faculty of Engineering in 2008.

His research is concerned with the software engineering of self-adaptive and distributed systems, including design methods, analysis techniques, languages and program support environments for these systems. His work on software architecture, that resulted in the Darwin architectural description language

(ADL) for distributed systems, was put to commercial use by Phillips in consumer television products as the Koala ADL. He has worked extensively with industry on joint research projects and as a consultant.

He is the author of over 100 refereed research publications in software engineering and has co-authored a book on concurrent programming entitled *Concurrency - State models and Java programs* which is now in its 2nd Edition and has been widely adopted world-wide. He was editor of the *Institute of Electrical Engineering's Proceedings on Software Engineering* and until 2007 was an associate editor of *Transactions on Software Engineering and Methodology*. He is the recipient of the 2005 ACM Special Interest Group on Software Engineering Outstanding Research Award for his work in Distributed Software Engineering. Most recently, he was awarded the SIGSOFT Retrospective Paper Award for his FSE4 paper on "Dynamic Structure in Software Architectures".

## **Reflecting Upon the Useful Life of Software**

Michael Feathers, *DepthFirst*

*Session Chair: Yann-Gaël Guéhéneuc*

Nearly every organization that develops software has maintenance challenges. Implicit in them is the assumption that it is valuable to keep existing software in service and that rewriting it would be cost prohibitive. There certainly is convincing evidence that both of these assumptions are true, but it's worth investigating whether we can finesse the problem of maintenance by treating software lifetime as a variable. In this keynote, Michael Feathers will explore this idea and its ramifications.

### **Short Biography of Keynote Speaker, Michael Feathers**

Michael Feathers is a consultant with DepthFirst, prior to that he was Member of the Technical Staff at Groupon. Prior to joining Groupon, Michael was the Chief Scientist of Obtiva, and a Senior Consultant with Object Mentor International. Over the years, Michael has spent a great deal of time helping teams after design over time in code bases.

Michael is also the author of the book *Working Effectively with Legacy Code* (Prentice Hall, 2004).