AN ABSTRACT OF THE DISSERTATION OF

Iftekhar Ahmed for the degree of Doctor of Philosophy in Computer Science presented on May 10, 2018.

Title: Improving the Quality of Software Using Mutation Testing and Fault Prediction

Abstract approved:

_____

Carlos Jensen

Our confidence in software systems depends on our confidence in the exhaustiveness of our testing. As software systems get more complex, the task of exhaustive testing becomes more complex and even infeasible in some cases. In order to build less error prone systems, we therefore need to not only focus on quickly and efficiently identifying bugs through testing and verification of software, but also on identifying factors associated with bugs in order to prevent them from occurring in the first place. This thesis shows how mutation testing and fault prediction can be used to improve the quality of software.

In the first part of this thesis, we investigate the notion of testedness and how that is associated with widely-used measures of test suite quality. The first measure is statement coverage, the simplest and best-known code coverage measure. The second measure is mutation score, a supposedly more powerful, though expensive, measure. We evaluate these measures using the actual criteria of interest; if a program element is (by these measures) well tested at a given point in time, it should require fewer future bug-fixes than a "poorly tested" element. If not, then it seems likely that we are not effectively measuring testedness. We show that both statement coverage and mutation score have only a weak negative correlation with bug-fixes, mutation score having slightly stronger correlation between the two.

In the second part, we investigate the applicability of mutation analysis in real world complex software system. Despite four decades of research on mutation analysis technique, its use in large systems is still rare, in part due to computational requirements and high numbers of false positives. We present our experiences using mutation analysis on the Linux kernel's RCU (Read Copy Update) module, where we adapt existing techniques to constrain the complexity and computation requirements. We show that mutation analysis can be a useful tool, uncovering gaps in even well-tested module like RCU. This experiment has not only led to the identification of gaps and bugs in the RCU module but also has led to a discussion on identifying domain specific mutants in order to increase the applicability of mutation analysis in real world applications.

In the third part, we investigate fault prediction models. Although there has been much research on predicting failures, those predictions usually concentrate either on the technical, or the social side of software development. However, software development is not an isolated activity, it requires coordination between individuals and technology. Therefore, to attain the best possible predictive capability, we need to analyze the complex interactions between socio-technical factors. Using one such socio technical factor, merge conflict, we found significant improvement in fault prediction.

Improving the Quality of Software Using Mutation Testing and Fault Prediction

by
Iftekhar Ahmed

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 10, 2018
Commencement June 2018

Doctor of Philosophy dissertation of Iftekhar Ahmed presented on May 10, 2018.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Iftekhar Ahmed, Author

ACKNOWLEDGMENTS

The work described in this dissertation would not have been accomplished without the tremendous support of many individuals. I am truly thankful to them all.

First and foremost, this work would not have been possible without the guidance of my Ph.D. advisor, Carlos Jensen. Carlos accepted me to his group and believed in me when no one else including me would believe in me. He helped me regain my confidence and unquestionably played a pivotal role in mentoring my growth as a researcher. I would like to take this opportunity to express my sincere gratitude to him for being my advisor, my mentor.

I would also like to thank the other professors at Oregon State University that have provided special guidance and mentorship during my time here. Alex Groce, Anita Sarma, and Margaret M. Burnett taught me a great deal about testing, empirical analysis and human aspect of software engineering, and provided me with the advice on how to become a better researcher. I would also like to thank Paul E. McKenney for the guidance and advice he has given me during the time I have known him and also for introducing me to the wonderful world of Linux Kernel and opening up the immense research opportunity that is there.

I would also like to thank the members of my research group and other students I have been fortunate enough to work with at Oregon State: Rahul Gopinath, Amin Alipour and Caius Brindescu. I will miss our memorable conversations over coffee and all the experiences we shared.

I am also deeply grateful to my family. In particular, I am grateful to my brother, as I followed his footsteps to peruse Computer Science to begin with. I am indebted to my parents for their encouragement in pursuing my curiosities from a young age, without which this Ph.D. may not have happened.

Finally, and most importantly, I would like to thank my wife, Umme Ayda Mannan, for always being there as an unwavering foundation for me. Ayda, this Ph.D. was possible only because you were there.

Concern for man and his fate must always form the chief interest of all technical endeavors. Never forget this in the midst of your diagrams and equations.

Albert Einstein

CONTRIBUTION OF AUTHORS

Table 1: Author Contributions

Task	Conception	Data Collection	Empirical Analysis	Final Draft
Chapter 2	IA, AG, CJ	IA	IA	IA, AG, CJ
Chapter 3	IA, AG, CJ	IA	IA	IA, AG, CJ, PM
Chapter 4	IA	IA, CB, UM	IA, CB	IA, AS, CJ

The following authors contributed to the manuscript: Iftekhar Ahmed (**IA**), Rahul Gopinath (**RG**), Caius Brindescu (**CB**), Umme Ayda Mannan (**UM**), Paul E. McKenney (**PM**), Anita Sarma (**AS**), Alex Groce (**AG**) and Carlos Jensen (**CJ**). Their individual contributions to various papers are summarized in Table 1.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

Chapter 1 INTRODUCTION

Software continues to become a more and more pervasive part of our lives, making quality of these software systems important. Unfortunately evaluating quality, especially with regards to the presence of bugs, becomes more difficult as the complexity of the software increases. Failure to meet quality requirements can carry significant costs, especially when these lead to software failures [29]. Thus, checking software quality may consume a large amount of time and resources not only during development stages, but also after the software is released.

The best way to ensure software meets quality requirements is to engage in extensive and thorough software testing. The goal of testing is to discover faults in the System under Test (SUT) by executing test cases that create the conditions that will lead to failure, and provide the instrumentation to detect such a failure. In addition to common manual testing approaches, such as Unit Tests [108], there are a growing number of popular automated testing techniques such as symbolic execution (SE) [123], random testing [172], and Search-Based Software Testing (SBST) [17].

The best technique for evaluating test suites and test coverage is currently mutation analysis which introduces simple syntactic changes to the program and measures the ability of test-suite to distinguish the semantic difference caused by these changes. Mutation analysis subsumes condition coverage criteria [143, 167] detects more bugs [132], produces mutations that have some characteristics of real-world bugs [9,10], and tracks the fault detection capability of test-suites. These factors make mutation analysis an attractive tool for researchers investigating test-suite quality.

Despite its potential, mutation analysis is not often used by practitioners on large and complex programs. One of the primary reasons mutation analysis is avoided is the large number of mutants that can be generated, which grows exponentially with program complexity. Exhaustive mutation analysis generates large numbers of mutants, but mutant sampling [176, 209, 215] and mutant execution optimizations [109, 177, 216]

can help to mitigate the problem. Offutt et al. [166] categorized these efforts into do fewer, do faster, and do smarter approaches. Unlike model checking, mutation analysis doesn't require any kind of modeling of the environment and associated data structures, which makes it more easily applicable to complex systems.

Though testing is effective in ensuring software quality, as software systems get more complex, the task of exhaustive testing becomes more complex and even infeasible in some cases. In order to build less error prone systems, we therefore need to not only focus on quickly and efficiently identifying bugs through testing and verification of software, but also on identifying factors associated with bugs that could be used in fault prediction techniques to potentially help us focus quality-assurance efforts on the most defect-prone parts of the code.

Software fault prediction strives to improve software quality and testing efficiency by constructing predictive classification models from different types of attributes. The focus is to predict software faults and where they may be located, for instance which modules are likely to be particularly fault-prone. In order to achieve this, researchers have explored many different techniques that can generate a prediction system from known training examples. Researchers have historically hypothesized that properties of the code, measured using code metrics, could usefully predict fault-proneness. Code metrics could measure size (larger files may be more fault-prone), or complexity (more complicated files may be more fault-prone). The value of code metrics in fault prediction has been well-explored [131, 150, 220]. Studies [26, 157, 182, 207] also suggest that software development process (e.g., developer count, code ownership, developer experience, change frequency) is also important indicators of fault proneness.

Overall, the primary focus so far has been on investigating technical factors but more recently researchers have started investigating the effectiveness of using socio-technical factors in fault prediction [25] focusing on the fact that software development

is not a purely technical process but a socio-technical process and investigating socio-technical factors can be used as important indicators of fault proneness.

1.1 Background

In the following section we discuss mutation analysis, the problems associated with applying mutation analysis and the state of the art on how to mitigate these issues. Then we discuss about fault prediction techniques and the current state of art on using process and socio-technical metrics for fault detection.

1.1.1 Mutation Analysis

The idea of mutation analysis was first proposed by Lipton [137]. Mutation analysis seeks to evaluate test suites by embedding known defects into the SUT. These defects are called mutants, and are produced by simple syntactic rules, e.g., changing a relational operator from ">" to ">=". These rules are called mutation operators. The term mutagen is synonymous with both mutation-operator and mutation transformer in literature. If only one single mutation separates the mutant from the original program, it is called a first order mutant (FOM). A higher order mutant (HOM) is separated from the original by multiple mutations.

1.1.1.1 Similarity of mutants to real faults

Researchers have been trying to verify whether mutation analysis are similar to real faults. Daran et al. [48] found that the errors caused by 85% of mutants are similar to the errors caused by real faults. Unfortunately, this study has very small sample size both in terms of faults and SUT which makes the findings prone to statistical noise.

Andrews et al. [9] compared the ease of detection for both real and hand seeded faults where ease of detection is calculated as the percentage of test cases that killed each mutant. Their study concluded that the ease of detection of mutants was similar to real faults. However, their results rely on real faults from a single program, which limits the scope of inference. Namin et al. [160] found that caution is required when using mutation analysis as a proxy for real faults. They found that programming language,

kind of mutagens used, and even test-suite size has an impact on the relation between mutations and real faults. They found that there is only a weak correlation between real faults and mutations.

Just et al. [110] investigated the relation between mutation score and test case effectiveness using 357 bugs from 5 open source applications. They found that mutation score increased along with test-suite effectiveness for 75% of the cases while for coverage, the increase happened only for 46% of the cases. They also found that coverage increase results in mutation coverage increase, but the reverse does not seem to happen. Gopinath et al. [77] found that real faults are usually three to four tokens long and there is a mean syntactic difference between real faults and first order mutants, which means that it is not self-evident mutation analysis will help find real faults.

1.1.1.2 Limitations of mutation analysis

Mutation analysis suffers from a number of limitations, generating a large number of mutants for even small programs is one of them. We discuss about the limitations of mutation analysis in the following subsections.

1.1.1.2.1 Computational Cost of Mutation Analysis

The computational cost of mutation analysis is very high due to the large number of mutants that need to be analyzed. Mutants are generated exhaustively for each program element to which mutation operators can be applied. The total number of generated mutants increase with the number of applied mutation operators and the size of the program. The number of mutants eventually affects the running time of mutation analysis.

The runtime T_{total} of mutation analysis can be expressed as:

$$\boldsymbol{T_{total}} = \sum_{\boldsymbol{m \in M}} \boldsymbol{t_{seed,m}} + \sum_{\boldsymbol{m \in M}} \boldsymbol{t_{cmpl,m}} + \sum_{\boldsymbol{m \in M}} \sum_{\boldsymbol{tc \in TC}} \boldsymbol{t_{test,m,tc}}$$

Where M is the set of mutants, TC is the set of test cases and $t_{seed,m}$, $t_{\mathrm{cmpl},m}$ and $t_{test,m,tc}$ are the times for seeding, compiling, and testing respectively, a combination of a mutant m and a test case tc. Generally, the compilation time and the testing time are the dominant components in the total time of traditional mutation analysis.

1.1.1.2.2 Equivalent mutants

One problem with mutation analysis is that a syntactic change to the code may leave the program semantically the same. This is known as the "equivalent mutant problem". Identification of equivalent mutants and verification whether there exist any test inputs that expose semantically equivalent mutants account for a large amount of time and effort. Equivalent mutants could easily skew the test suites' mutation score, and no automated method can completely remove this problem. Manual equivalence inspection [212] requires a significant investment of developer time to weed out equivalent mutants. Even then, the identification rate of equivalent mutants is about 8% [1].

As the problem of identifying equivalent mutants is fundamentally an undecidable problem [33], the literature is limited to approximation techniques. One approach is to use compiler optimization rules to detect equivalent mutants [20]. If the original program and a mutant are both compiled to the same object code, using semantics-preserving optimizations, then the mutant is equivalent to the original program. Even in the presence of undefined behavior (which could make an optimization semantically unsafe), no test could ever reveal differences, so there is effective equivalence. Mothra [164] uses this technique and can identify about 45% of equivalent mutants. Recently Papadakis et al. proposed Trivial Compiler Equivalence (TCE) which simply declares equivalencies only for those mutants where the compiled object code is identical to the compiled object code of the original program [178] and collects mutants into equivalence classes for execution. In our studies we use TCE for equivalent mutant identification.

Another approach identified by researchers is using constraint solving [164, 165] to detect equivalence. Researchers also showed that program slicing [89, 94] can help in detecting equivalent mutants. Although these techniques are powerful, they suffer from the inherent limitations of the constraint-based and slicing-based techniques: these techniques are often neither very scalable nor very easy for developers to apply. Many other techniques, such as dependence-based analysis [89] have been proposed by researchers.

1.1.1.2.3 Duplicate mutants

Another problem related to equivalent mutants is the "duplicate mutant" problem. This is when multiple mutants generate the same program, semantically speaking. Papadakis et al. proposed to use TCE for identifying duplicate mutants [178] as well. In our studies we use TCE for duplicate mutant identification.

1.1.1.2.4 Need for human oracle

Another reason for less adoption of mutation analysis is the lack of proof of mutation analysis' applicability and effectiveness for complex real-world projects. Mutants that are not identified by the test suite needs to be manually checked and this requires human oracle. Researchers have been trying to reduce the computational cost in mutational analysis and but none of these efforts try to prioritize among the surviving mutants that need human intervention. Because of these shortcomings, mutation analysis has been more widely adopted by academia than industry, and the technique and associated tools have mostly been evaluated using relatively simple programs and test suites.

1.1.1.3 Techniques for making mutation scalable

Although mutation testing is powerful, its high execution cost has always been a problem. This problem has been extensively researched. Researchers have come up with various techniques for mitigating this issue. Offutt et al. [166] categorized these efforts into "do fewer", "do faster", and "do smarter" approaches. The "do fewer" approach seeks to reduce the number of mutants tested without a significant loss in accuracy of the final result, while the "do faster" approach tries to make the process of

running each mutant as fast as possible, and “do smarter” approach tries to distribute the load over several machines, or save and reuse the invariant results of a previous run to make the current run faster.

1.1.1.3.1 Do Fewer

Jia et al. [102] classified the approaches that use fewer mutants into four categories: selective mutation, mutant sampling, higher order mutation and mutant clustering. Selective mutation technique proposed by Wong and Mathur [209] seeks to find a small set of mutation operators that generate a subset of all possible mutants without a significant loss of test effectiveness. It is useful for reducing the size of a mutation operator list. Mutants sampling is another technique which had significant success. Wong et al. [209] found that even sampling of only 10% mutants is effective in achieving an accuracy close to 80%. Patrick et al. [179] use static-analysis to identify mutations that are closest to the program. These are harder to kill, and contribute more to the mutation analysis. Selection of higher order mutants [104] is another successful technique for reducing the number of mutants without losing much efficiency. The best hybrid strategy of higher order mutant selection achieves about 50% reduction in effort, with 1.75% average loss in accuracy. Mutant clustering is an approach that selects a subset of mutants using a clustering algorithm. Each mutant in a cluster is likely to be killed by the same set of test cases.

1.1.1.3.2 Do Faster

Schema based mutation analysis is one of the “do faster” techniques where all mutations are encoded within the same source code, and the code is compiled once, resulting in savings by avoiding multiple compilation of non-mutating parts. The bytecode-based approaches such as in MuJava [139] also help in reducing the cost of computation and fall under the do faster approach. Just et al. [110] inserts triggering conditions into the source code, such that each mutant can be triggered separately. This eliminates the separate compilation overhead.

1.1.1.3.3 Do smarter

The do smarter approaches include weak mutation, parallelization of mutant execution and incremental mutation approaches. Weak mutation is an approximation technique that compares the internal states of the mutant and the original program immediately after execution of the mutated potion of the program. Split-stream execution is another "do smarter" approach. King [123] initially proposed split-stream execution which splits the execution stream of the original program to begin mutant execution at the point where the mutated statement appears.

1.1.2 Fault prediction

Fault prediction strives to improve software quality and testing efficiency by constructing predictive classification models from different types of attributes. The goal is to predict software defects and their location at various granularity levels such as at file level, method level etc.

1.1.2.1 Classification models for predicting faults

Several classification models have been evaluated by researchers. Liang et al. used K-nearest [134], Gyimothy et al. [86] used decision tree and neural network as their classifier. Brun et al. used SVM and decision tree [30]. Kim et al. [120] also used SVM as their classifier. Menzies et al. [150] empirically found that Naive Bayes with a log-filtering preprocessor on the numeric data outperformed those of rule or tree-based learning methods. However, Lessmann et al. compared 22 classification models over 10 public domain software development data sets from the NASA MDP repository [131] and found that predictive accuracy of most classifiers do not differ significantly. They also found that simple classifiers suffice to model the relationship between code attributes and software defect. Ensembles of classifiers are also used in prediction [200] and have been shown to significantly improve predictive performance.

1.1.2.2 Metrics used for fault prediction

Researchers have been trying to come up with better fault prediction models. Various metrics have been used as the independent variables in these models.

1.1.2.2.1 Process Metrics

Researchers have used change metrics or code churn as predictors of faults. The intuition being bugs are introduced by changes [46] and thus, the more changes are done to a part of the source code the more likely it will contain bugs. In [80], Generalized Linear Models were built based on several change metrics, e.g., number of changes or average age of the code. A study showed that relative change metrics from the Windows Server change history are better indicators for defect density than absolute values [156]. The fault and change history in combination with a (negative binomial) regression model achieved good performance in predicting not only the location, but also the number of bugs [171]. Furthermore, the more complex source code changes are (as measured by entropy), the more likely they are bug-prone [91]. Nagappan et al. found that the number of subsequent, consecutive changes (rather than the total number of changes) is a strong predictor for bugs [158]. Shihab et al. predicted surprise defects in files that are rarely affected by changes [196]. A study on changes in general showed that a substantial amount of changes are non-essential changes [118].

1.1.2.2.2 Code Metrics

Using code metrics for predicting bugs assumes that a more complex piece of code is harder to understand and to change, and therefore, it is likely to contain more bugs [46]. Basili et al. investigated the impact of the CK object-oriented metrics suite to software quality [21]. The usefulness of (static) code metrics to build prediction models was demonstrated using the NASA dataset [150]. In an extensive study conducted with the same dataset focusing on evaluating different machine learning algorithms [131] the conclusion was that the difference between those algorithms is mostly not statistically significant. However, this ceiling effect is reported to disappear when focusing only on

maximizing detection and minimizing false alarm rates [151]. The practicability of lines of code (LOC) to predict defects was demonstrated in [214]. An extensive empirical study with 38 different metrics and multivariate models to predict the fault-prone modules of the Apache web-server is presented in [217].

Liang et al. used a combination of Object-Oriented metrics, statistics for warnings of each tool (e.g., number of warnings reported for the same location by each tool, and number of warnings reported for the same file by each tool) as their feature set [134]. Gyimothy et al. showed that object oriented metrics are useful predictors of faults [86]. Brun et al. used the Daikon dynamic invariant detector to generate runtime properties and used them as the feature set [30]. Kim et al. used Change Metadata such as length of change log, changed LOC (added delta LOC + deleted delta LOC) along with a combination of Object-oriented metrics as their feature set [120] showing that they had the highest level of accuracy compared to Gyimothy et al. and Brun et al.

1.1.2.2.3 Technical networks

Technical networks have been used in previous work to build prediction models for failures. Zimmermann et al. [217, 218] constructed networks from dependency information for binaries and subsystems in Windows Server 2003. This study used social network analysis on dependency information to build prediction models for post-release failures. Their results indicated that models built on social network metrics were better indicators of future failures than models based on standard source code metrics. Their approach leveraged SNA metrics to capture both local and global effects of network connectivity on defect-proneness.

1.1.2.2.4 Social networks

Prior work has also shown that software artifact properties are directly influenced by social network properties of teams, such as the email interactions, and their contribution history of developers. In earlier work, Bird et al. [24] constructed email social networks from open source project mailing lists and found that social network analysis measures

were highly correlated with development activity. Pinzger et al. [181] used contribution history to construct the networks of binaries and the developers that contributed to them. They found that measures such as degree centrality, closeness centrality, and Bonacich power in contribution networks also had very good predictive power in determining failure-prone binaries. Meneeley et al. [148] created networks that consisted solely of developers where edges between developers were based on collaboration on common files. They used social network analysis to assign values of metrics such as betweenness, degree, and closeness to developers. They found that a model using these metrics explained 60% of the variance of failures during the testing phase, but only 2.6% of the post-release failures.

1.1.2.2.5 Combination of metrics

These metrics are rarely used in isolation but instead are often combined for building bug prediction models [16, 195]. Although a general consensus has not been achieved, several studies showed that change metrics potentially outperform code metrics [114, 125, 154]. Hall et al. noted that models using a combination of static code metrics, process metrics and source code text performs better [87]. Combining social and technical networks has recently become a subject of study. Socio-technical networks encode connections between people, connections between technical artifacts and connections between people and artifacts. Amrit et al. [7] first proposed use of socio-technical networks. Cataldo et al. [38, 39] looked at time to resolution for modification requests and found that developers take 32% less time to complete tasks when there is "congruence" in the socio-technical network.

1.2 Research Goals

Philosophically, at a higher-level this thesis has the following high-level goals:

1. Investigating whether statement coverage or mutation coverage is a better measurement of test suite quality using a novel evaluation criteria of future bug proneness of a program element.

2. Investigating if mutation analysis can be applied to the real world complex systems within reasonable time and resource constraints, and if doing so could uncover bugs in well-tested software. To this end, we propose to investigate how we can quickly and efficiently triage all the mutants that are generated.
3. Investigating the applicability of fault prediction models built using socio-technical factors.

1.3 Structure

This thesis is based on the following papers:

Chapter 2 presents our paper "Can Testedness be Effectively Measured?" which was published at FSE 2016. This paper answers the first high level research goal of comparing the effectiveness of two testedness measurement criteria (statement coverage and mutation score).

Chapter 3 presents our paper "Applying mutation analysis on kernel test suites an experience report" which was published at ICST Workshop on Mutation Analysis in 2017. This paper answers the second research goal of investigating the applicability of mutation analysis in real world complex software system adapting existing techniques to constrain the complexity and computation requirements.

Chapter 4 presents our paper "An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts" which was published at ESEM 2017. This paper investigates the third research goal pertaining to applicability of socio-technical factors for fault prediction model building.

Chapter 5 concludes this dissertation and presents our ideas on how to take our research forward.

1.4 Contributions

The contributions of this dissertation are:

1. Proposing a novel approach of future bug-proneness as a criterion of measuring and comparing effectiveness of various testedness measurements (Chapter 2).
2. Empirical evaluation of effectiveness of testedness measurements such as statement coverage and mutation score and identifying mutation score as a better criterion between the two (Chapter 2).
3. Investigating the applicability of mutation analysis in real world complex software system adapting existing techniques and showing its effectiveness (Chapter 3).
4. Investigating the applicability of socio-technical factors for building fault prediction models (Chapter 4).

Can Testedness be Effectively Measured?

Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, Carlos Jensen

2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 547-558)

Chapter 2 CAN TESTEDNESS BE EFFECTIVELY MEASURED?

2.1 Introduction

The quality of software artifacts is one of the key concerns for software practitioners and is typically measured through effective testing. While it is widely held that "you cannot test quality into a product," you can use testing to detect that the Software Under Test (SUT) has faults, and to estimate its likely overall quality. Moreover, while testing itself does not produce quality, it leads to the discovery of faults. When these faults are corrected, software quality improves.

Testing software poses questions. First, how much testing is needed? Has "enough" testing been done? Second, where should future test efforts be applied in a partially tested program? The typical approach to answering these questions is to measure the quality of the test suite, not the SUT. Numerous measures, primarily focused on code coverage [72,82,76] have been proposed, and numerous organizations set testing requirements in terms of coverage levels [185]. Both code coverage and mutation score measure the "testedness" of an SUT, using the dynamic results of executing a test suite[1]. However, it is not established that using such testing requirements, or suite quality measures in general, translates into an effective practice for producing better software.

While test driven development, in particular, has pushed testing to new prominence, practicing software programmers often balk at having to satisfy what they see as arbitrary coverage requirements. Some go so far as to suggest that "code coverage is a

[1] Most studies consider coverage as measuring the testedness of the entire SUT for a given suite, but it is also obviously reasonable to project this concept onto individual program elements and measure how well tested each is, and most practical applications of coverage assume this usage as well.

useless target measure"[2]. Some studies seem to support this conclusion, at least in part [100]. The utility of testing itself has even come under attack[3].

Our aim in this paper is to place the evaluation of test suites (and thus decision making in testing) on a firmer footing, using a measure that translates directly into practical terms. There is, of course, no end to studies measuring the effectiveness of test suite evaluation techniques. However, these studies tend to either cover only a few subject programs or faults, not use real faults, not measure what developers directly care about, or assume the validity of mutation testing, which is itself a relatively unproven method for evaluating a test suite. The methodology of such studies also often involves using tool-generated or randomly-chosen subset-based test suites to measure correlation. Such suites may not resemble real-world test suites, and thus be of little relevance to most developer practice.

We propose a simpler and more direct method of evaluation that eliminates some of the concerns mentioned above. It can be argued that a correlation between measure of suite quality and fault detection is meaningless if the faults detected are unimportant, trivial, or artificial. As a recent paper evaluating the ability of automated unit test generators to detect real faults phrased it "Just because a test suite... [is effective] does not necessarily mean that it will find the faults that matter" [193]. It is very hard to argue that a bug that has been discovered and fixed did not matter. However, fixing bugs is di cult and resource-intensive, requiring developers (and possibly testers) to devote time to implementing a correction (and, hopefully, validating it). In many cases, the bug was detected because it caused problems for users. There is evidence that problems in code, such as those identified by code smells, that do not have significant immediate consequences are often never corrected, even at the price of design degradation [3]. Bug-fixes usually indicate important defects as developers thought

[2] For examples, see http://martinfowler.com/bliki/ TestCoverage.html and http://blog.ploeh.dk/2015/11/16/ code-coverage-is-a-useless-target-measure/

[3] For examples, see https://pragprog.com/magazines/2012-10/its-not-about-the-unit-tests and http://www.rbcs-us.com/documents/Why-Most unit-Testing-is-Waste.pdf

these problems worth addressing. The most practical goal of testing therefore, is to prevent future bug-fixes, by detecting faults before release, avoiding impact on users, and usually lowering development cost.

We should then evaluate test suite quality measures by a simple process: does a higher measure of testedness for a program element (in our work, a statement, block, method or class) correlate with a smaller number of future bug-fixes? By avoiding whole-project measures and focusing on individual program elements, we avoid the confounding effects of test suite size [100]. While a large test suite can produce higher coverage and detect more faults, even if coverage and fault detection are not themselves directly related, it cannot (at least in any way we can imagine) cause statements that are covered to have fewer faults than those that are not covered, unless coverage itself is meaningful. Similarly, using individual program entities as the basis of analysis mitigates some of the possible effects of, e.g., test suites with good assertions also having better coverage. This can cause a test suite with high coverage to perform better than a test suite without high coverage, on average, but it cannot, so far as we can see, plausibly result in covered entities having fewer bug-fixes than entities not covered, unless coverage itself matters.

The core argument for our analysis is as follows: if a particular program element is fully tested to conform to its specification, then that program element should have no bug-fixes applied (until the element ceases to exist as a result of a non bug-fix modification, e.g. adding new functionality potentially invalidating the old specification). Similarly, a program element that is not tested at all should on average have a higher chance of seeing future bug-fixes applied for the simple reason that a fault had no chance of being caught through testing. This, on its own, should result in a strong negative correlation between testedness and future bug-fixes, if our fundamental assumption about the utility of testing is correct and our measure of testedness is effective.

In general, a well-tested program element should require fewer bug fixes than a less well-tested program element.

In this paper we assume (and later, based on empirical data, argue) that testing itself is beneficial. We therefore primarily aim to evaluate the measurement of test suite quality/testedness. We focus on two important, widely studied, measures of suite quality. First, statement coverage is the simplest, most widely used, and easiest to understand coverage measure, and has some support as an effective measure of test suite quality in recent work [76]. Second, mutation score is not only commonly advocated as the best method for evaluating suite quality but is essential to most other studies of coverage method effectiveness [82, 110].

We evaluate these test suite evaluation methods empirically using a large representative set of real world programs, real world test suites, and bug-fixes, and find that while there is a small (but statistically significant) negative correlation between our testedness measures and future bug-fixes for program elements, the effect is so small as to be practically insignificant. There is very little useful continuous relationship between measures of testedness and actual tendency to not have bugs detected and fixed, and while it is reasonable to bet that a more-tested element will have fewer faults, the size of the effect is very small.

However, we do find that there is a consistent and practically (as well as statistically) significant difference in the mean number of bug-fixes for code, if we apply selected binary measures of "well-testedness" based on coverage or mutation score. For example, program elements with at least a 75% mutation score see, on average, only about half as many future big-fixes (normalized[4]) as program elements with a lower mutation score.

[4] By normalized bug-fixes, we mean bug-fixes per statement/line for elements larger than a single statement or line; unless we indicate otherwise, we always normalize bug-fixes when required.

One intuitively appealing explanation for the low correlation of testedness to bug-fixes is that, even if "poorly" tested, unimportant pieces of code are likely to see few future bug fixes. If very few users execute a program element, or if its effects have very limited impact, then the bug will likely not be fixed (even if reported). However, the problem of varying program element importance is unlikely to be the root cause for the lack of a useful continuous correlation for suite evaluation measures. If it were, we would expect the effects of importance to also prevent binary testedness criteria based on mere coverage from predicting future bug-fixes (since no one will bother to test program elements that are unlikely to ever exhibit important bugs). However, like program elements with < 75% mutation score, program elements that are not covered are also likely to see nearly twice as many future bug-fixes[5].

Nonetheless, perhaps a suite quality measure should reflect the importance of program elements. However, forcing developers to annotate code by its importance is impractical; we need a static measure of importance. One approach is to say that complex elements are more likely to be important, since developing complex code with many operators and conditionals, but low importance, is an unwise use of development resources. In this case, in addition to its other advantages, mutation testing may help take importance of code into account, in that complex program elements produce more mutants than simple elements (e.g., a simple logging statement will seldom perform any calculations, and so often only produce a single statement-deletion mutant). We therefore also measure whether the number of mutants (as a measure of code complexity) predicts the number of bug-fixes applied to a program element, and whether the number of mutants predicts the mutation score for an element. Both effects are significant but small. Surprisingly, more complex code sees slightly fewer bug-fixes than simple code. As might be expected if complexity is associated with importance, more complex code is also slightly more tested, according to mutation score. Both effects are too weak to be of much practical value, however.

[5] We only demonstrate this result for statements and methods; there were too few classes that were not covered by any tests in our data to show a significant relationship.

Our findings with respect to correlation of testedness measures and bug-fixes are potentially troubling for the research community. Software testing researchers often use a difference of a few percentage points in mutation score or a coverage measure as a means to assert that one test generation or selection technique is superior to another. However, our data shows that relying on a few percentage points is dangerous, as such small differences may not indicate real impact in terms of defects that are worth fixing. On the other hand, our data seems to support the types of "arbitrary" adequacy criteria often imposed by managers or governments (if not the precise values used). Indeed, our data suggests that while a continuous ranking of testedness for program elements is not currently possible, using various empirically validated "strata" of testedness (not covered, covered but with poor mutation score, covered with high mutation score) may provide a simple, practical way to direct testing efforts.

The contributions of this paper are:

- A novel approach to examining the utility of test suite quality measures that is based on direct practical consequences of testing.

- Analysis of relationships between bug-fixes, test suite quality (testedness) measures, and code complexity and importance metrics for 49 sampled projects from Github and Apache.

- Evidence that there is small negative correlation between the number of mutants (normalized) and the number of future bug-fixes to a program element, indicating that complexity alone does not predict importance well; in fact, more complex program elements seem to be changed less often than simple ones. However, this may partly be due to the fact that more complex elements are also somewhat more well tested (in terms of mutation score).

- Evidence that the negative correlation between testedness (by our measures) and number of future normalized bug-fixes is statistically significant, but far too small to have much practical impact.

- Evidence that well-chosen adequacy criteria (e.g.,is the mutation score above 75%) can be used to predict future normalized bug-fixes in a practical way (leading to differences of a factor of two in expected future bugs), and can serve to distinguish untested, poorly tested, and well-tested elements of an SUT.

2.2 Related work

Ours is not the first study to attempt to evaluate measures of testedness [82]. Researchers have often attempted to prove that mutation score is well correlated with real world faults. DeMillo et al. [55] empirically investigated the representativeness of mutations as proxies for real faults. They examined the 296 errors in TEX and found that 21% were simple faults, while the rest were complex errors. Daran et al. [48] investigated the representativeness of mutations empirically using error traces. They studied the 12 real faults found in a program developed by a student, and 24 first-order mutants. They found that 85% of the mutants were similar to real faults.

Another important study by Andrews et al. [9], investigated the ease of detecting a fault (both real faults and hand seeded faults), and compared it to the ease of detecting faults introduced by mutation operators. The ease was calculated as the percentage of test cases that killed each mutant. Their conclusion was that the ease of detection of mutants was similar to that of real faults. However, they based this conclusion on the result from a single program, which makes it unconvincing. Further, their entire test set was eight C programs, which makes the statistical inference drawn liable to type I errors. We also note that the programs and seeded faults were originally from Hutchins et al. [97] who chose programs that were subject to certain specifications of understandability, and the seeded faults were selected such that they were neither too easy nor too difficult to detect. In fact, the study eliminated 168 faults for being either too easy or too hard to detect, ending up with just 130 faults. This is clearly not an unbiased selection and cannot really tell us anything about the ease of detection of hand seeded faults in general (because the criteria of selection itself is confounding). A follow up study [10] using a large number of test suites from a single program, space.c, found that the mutation detection ratio and fault detection ratio are related linearly, with

similar results for other coverage criteria (0.83 to 0.9). Linear regression on the mutation kill ratio and fault detection ratio showed a high correlation (0.9).

The problems with some of these studies were highlighted in the work of Namin et al. [160] who used the same set of C programs as Andrews et al. [9], but combined them with analysis of four more Java classes from the JDK. This study used a different set of mutation operators and fault seeding by student programmers for the Java programs. Their analysis concluded that we have to be careful when using mutation analysis as a stand-in for real faults. The programming language, the kind of mutation operators used, and even the test suite size all have an impact on the relation between mutations introduced by mutation analysis and real faults. In fact, using a different mutation operator set, they found that there is only a weak correlation between real faults and mutations. However, their study was constrained by the paucity of real faults, which were only available for a single C program (also used in Andrews et al. [9]). Thus, they were unable to judge the ease of detection of real faults in Java programs. Moreover, the students who seeded the faults had knowledge of mutation analysis which may have biased the seeded faults (thus resulting in high correlation between seeded faults and mutants). Finally, the manually seeded faults in C programs, originally introduced by Hutchins et al. [97], were again confounded by a selection criteria which eliminated the majority of faults as being either too easy or too hard to detect.

Just et al. [110], using 357 real faults from 5 projects, showed that 1) adding more fault-detecting tests to a test suite led to the mutation score increasing more often (73%) than either branch (50%) or statement coverage (30%) and 2) mutation score was more positively correlated with fault detection than either of the other measures. Multiple studies provide evidence that mutation analysis subsumes different coverage measures [34,143,167], and it is on this basis that mutation score is often regarded as the “gold standard” for test suite quality measures.

One metric that is commonly used to measure the adequacy of testing is code coverage, that is, a measure of the set of program elements or code paths that are executed by a set of tests. A large body of work considers the relationship between coverage criteria and fault detection. Mockus et al. [152] found that increased coverage leads to a reduction in the number of post-release defects but increases the amount of test effort. Gligoric et al. [72,73] used the same statistical approach as our paper, measuring both Kendall τ and R^2 to examine correlations, for realistically non-adequate suites. Gligoric et al. found that branch coverage does the best job, overall, of predicting the best suite for a given SUT, but that acyclic intra-procedural path coverage is highly competitive and may better address the issue of ties, which is important in their research/method comparison context. Inozemtseva et al. [100] investigated the relationship of various coverage measures and mutation score for different random subsets of test suites. They found that when test suite size is controlled for, only low to moderate correlation is present between coverage and effectiveness. This conclusion held for all the coverage measures used. Frankl and Weiss [69] performed a comparison of branch coverage and def-use coverage, showing that def-use is more effective than branch coverage for fault detection and there is stronger correlation to fault detection for def-use than branch coverage. Namin and Andrews [159] showed that fault detection ratio (non-linearly) correlated well with block coverage, decision coverage, and two different data-flow criteria. Their research suggested that test suite size was a significant factor in the model. Wei et al. [206] examined branch coverage as a quality measure for suites for 14 Eiffel classes, showing that for randomly generated suites, branch coverage behavior was consistent across many runs, while fault detection varied widely. In their experiments, early in random testing, when branch coverage rose rapidly, current branch coverage had high correlation to fault detection, but branch coverage eventually saturated while fault detection continued to increase; the correlation at this point became very weak.

Offut et al. [167] showed that mutation coverage subsumes many other coverage criteria, including the basic six proposed by Myers [155]. Gupta et al. [85] compared

the effectiveness and efficiency of block coverage, branch coverage, and condition coverage, with mutation kill of adequate test suites as their evaluation metric. They found that branch coverage adequacy was more effective (killed more mutants) than block coverage in all cases, and condition coverage was better than branch coverage for methods having composite conditional statements. The reverse, however, was true when considering the efficiency of suites (average number of test cases required to detect a fault). Li et al. [132] compared four different criteria (mutation, edge pair, all uses, and prime path), and showed that mutation adequate testing was able to detect the most hand seeded faults (85%), while other criteria performed similarly to each other (in the range of 65% detection). Similarly, mutation coverage required the fewest test cases to satisfy the adequacy criteria, while prime path coverage required the most. Therefore, while there are no compellingly large-scale studies of many SUTs selected in a non-biased way to support the effective-ness of mutation testing, it is at least highly plausible as a better standard than other criteria.

Cai et al. [35] investigated correlations between coverage criteria under different testing pro les: whole test set, functional test, random test, normal test, and exceptional test. They investigated block coverage, decision coverage, C-use and P-use criteria. Curiously, they found that the relationship between block coverage and mutant kills was not always positive. Block coverage and mutant kills had a correlation of R^2 = 0.781 when considering the whole test suite, but as low as 0.045 for normal testing and as high as 0.944 for exceptional testing. The correlation between decision coverage and mutation kills was higher than statement coverage, for the whole test suite (0.832), ranging from normal test (0.368) to exceptional test (0.952). Frankl et al. [70] compared the effectiveness of mutation testing with all-uses coverage, and found that at the highest coverage levels, mutation testing was more effective. Kakarla et al. [113] and Inozemtseva et al. [99] demonstrated a linear relationship between mutation detection ratio and coverage for individual programs. Inozemtseva's study used machine learning techniques to come up with a regression relation and found that effectiveness is dependent on the number of methods in a test suite, with a correlation coefficient in the

range $0.81 \leq r \leq 0.93$. The study also found a moderate to high correlation, in the range $0.61 \leq \tau \leq 0.81$, between effectiveness and block coverage when test suite size was ignored, which reduced when test suite size was accounted for. Kakarla found that statement coverage was correlated to mutation coverage in the range of $0.73 \leq r \leq 0.99$ and $0.57 \leq \tau \leq 0.94$. Gopinath et al. [76] found that statement, out of branch, and path coverages, best correlated with mutation score, and hence may best predict defect density, in a study that compared suites and mutation scores across projects, rather than using multiple suites for the same project.

The study by Tengeri et al. [201] provided a simple (essentially non-statistical) assessment of how statement coverage, mutation score, and reducibility predicted project defect densities for four open source projects, using a limited set of mutation operators.

None of these studies, to our knowledge, adopted the method used in this paper, where rather than investigate faults and their detection, we look at whether being "well tested" has predictive power with respect to future defects[6]. Most also consider a smaller, less representative (at least of open source projects) set of programs, and the majority are based on programs chosen opportunistically, rather than by our more principled sampling approach. The programs used are often small but well-studied benchmarks such as the Siemens/SIR [197] suite, partly for purposes of comparison to earlier papers, and partly due to the lack of easily available realistic projects with test suites and defects, before the advent of very large open source repositories. Unfortunately, as noted by Arcuri and Briand, not at least attempting to randomize selection of programs to study can greatly reduce the generalizability of results [15].

[6] It is remotely possible that Tengeri et al. [201] are using a similar method, but this is not clear from their description, and the reasoning behind our approach is not elaborated in their work.

2.3 Methodology

Our goal was to evaluate various approaches to assessing the testedness of a program or program element using test suite quality measures.

2.3.1 Collecting the Subjects

For our empirical evaluation, we tried to ensure that the programs chosen offered a reasonably unbiased representation of modern software. We also tried to reduce the number of variables that can contribute to random noise during evaluation. With these goals in mind, we chose a sample[7] of Java projects from Github [71] and the Apache Software Foundation [11]. All projects selected used the popular maven [12] build system. We randomly selected 1,800 projects. From these, we eliminated aggregate projects that were di cult to analyze, leaving 1,321 projects, of which only 796 had test suites. Out of these, 326 remained after eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations). Next, the projects that did not pass their own test suites were eliminated as mutation analysis requires a passing test suite. Finally, we eliminated projects our AST walker could not handle. This resulted in 49 projects selected. The distribution of project size vs. test suite size, and the corresponding mutation score is given in Figure 2.1.

---

[7] Github allows us to access only a subset of projects using their search API. We believe that the results returned by Github search would not be dependent on their test suites, and hence should not confound our results.

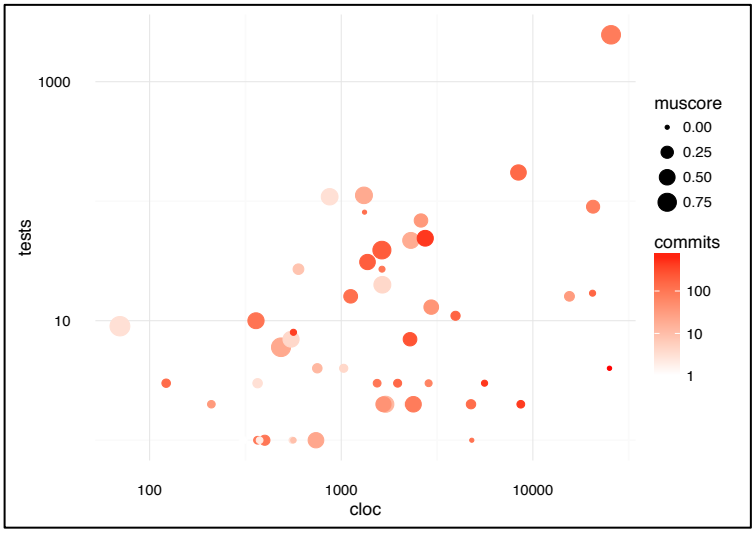

Figure 2.1: CLOC vs. tests for our projects.

2.3.2 Mutant Generation

In the next phase of our analysis, we used PIT [42] for our mutation analysis. PIT has been used in multiple previous studies [52, 76, 100, 194]. We extended PIT to provide the full matrix of test failures over mutants and tests. Mutants can basically be divided into three groups based on their runtime behavior: not covered, killed, and live mutants. We used this basic categorization in our analysis.

2.3.3 Tracking Program Elements

We started our investigation from an arbitrarily determined recent, but not too recent, point in time deemed the "epoch" - December 1, 2014. This was done to provide a point from which testedness (mutation score and statement coverage) could be calculated, and with respect to which bug-fixes could be considered to be "in the future". For the source code and test suite at epoch, we computed mutation score and statement coverage for each statement, block, method, and class in each project.

In order to determine when a program element (statement, block, method, or class) was changed, and track its history, we used the GumTree Differencing Algorithm [63]. For each element of interest, we considered it changed if the corresponding AST node

was changed, or had any children that were added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allowed us to accurately track the history of the program elements.

Using AST differencing gives us three advantages over simple line-based differencing. The first is that the algorithm ignores any whitespace changes. Second, we are able to track a node even if its position in the file changes (e.g. because lines have been added or deleted before our node of interest). Third, we are able to track nodes across refactorings, as long as the node stays in the same file. For example, we can track a node that has been moved because of an extract method refactoring.

When considering which statements to track, we used the version of the source code at epoch to determine which AST node resided at that particular line. We filtered only the commits that touch the le of interest. We then tracked that AST node forward in time, taking note of the commits that changed that particular node. For Java, it is possible for multiple statements to be in the same line (for example, a local variable declaration statement inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

2.3.4 Classifying Commits

In order to answer our research questions, we needed to categorize the code commits. For each program element, we computed the number of commits that touched that element starting from the epoch. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) Other - commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key problems are that it is not always trivial to determine which category a commit falls under, and that larger projects see a huge amount of activity. Manual classification of all commits was therefore not an option, and we decided to use machine learning techniques for this purpose, rather than limit the statistical power of

our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large bias into our results).

2.3.4.1 Manual Classification of Fix-inducing Changes

In order to build a classifier for bug-fixing commits, we randomly sampled commits and manually labeled fix-inducing commits. Some keywords indicating bug-fixes were Fix, Bug, and Resolves, along with their derivatives. We should mention that not all bug-fixing commit messages include the words bug or fix; indeed, commit messages are written by the initial contributor of a patch, and there are few guidelines as to their contents. A similar observation was made by Bird et al. [23], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. Improvements were manually identified based on the following keywords: Cleanup, Optimize, and Simplify or their derivatives. Commits were placed into the Other category if they had the keywords Add or Introduce. The number of lines modified was also compared with the lines added. Those commits with more lines added than modified were considered more likely to be associated with new features and were placed in the Other category. Anything that did not fit into this pattern was also marked as Other. We manually classified a set of 1,500 commits.

2.3.4.2 Training the Commit Classifier

We used the set of manually classified commits as the training data for the machine learning classifiers. Two evaluators worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the Other category.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) for automatically classifying the commits, using the scikit [180] platform. We applied the classifiers to the training data with 12-fold cross-validation. Our goal was to achieve high precision and recall, so we used the F1-score to measure and compare the

performance of the models. The F1-score considers precision and recall by taking their harmonic mean. The NB classifier outperformed the SVM. Tian et al. [202] suggested that for keyword based classification the F1 score is usually around 0.55 which also happened in our case. We used the classification identified by the NB classifier to classify 11,566 commits. Table 2.1 has the quality indicator characteristics of the NB classifier. While our classifier is far from perfect, it is comparable to "good" classifiers for this purpose in the literature (over a larger set of projects), and we believe it is likely that any biases do not have confounding interactions with the goals of our project. That is, while we may only analyze about 43% of bug-fixes, it would be surprising if the missed bug-fixes relate in some systematic way to the dynamic testedness measures of program elements, given that the classifier only sees code commits. Since our analysis only relies on relative counts of bug-fixes for elements, so long as we do not systematically undercount bug-fixes for only some elements, our results should be valid.

Table 2.1:Naive Bayes classifier details

	Precision	**Recall**	**F1 score**	**Support**
Bug-fix	0.63	0.43	0.51	75.00
Other	0.74	0.86	0.80	140.00

The bug-fixes associated with each program element in our analysis are based on the classifier results in a simple way. For each element, we count commits that affect that element that are classified as Bug-fix up to the first commit that is classified as Other. This is because once an element has had a change that is not a bug-fix, it is often no longer valid to assume tests at the epoch apply to that element, or that it even still exists with the same functionality. However, so long as only bug-fixes are applied, we assume the tests still apply to the program element, so all bug-fixes count as missed by the tests at epoch. Note that our classifier for Other commits is highly effective.

2.4 Analysis

We analyze the impact of testedness on program element bug-fixes using two measurements: mutation score and statement coverage. For mutation score, we analyze the score of each statement, block, method, and class, in increasing lexical scope. Since statement coverage is already at the statement level, we investigate the statement coverage of each block, method, and class in increasing lexical scope.

2.4.1 Correlation Results

We answer this question in increasing scope from statement, smallest block, method, and then class. In each scope, we evaluate how the degree of adequacy in both mutation score and statement coverage affects the total number of bug-fix commits.

2.4.2 Mutation Score (μ)

The correlation between number of bug-fixes per statement and mutation score is given in Table 2.2. For statements and methods, there is a statistically significant small negative linear correlation between number of bug-fixes per statement and mutation score. A similar effect is observed with Kendall τb correlation, where a small but statistically significant negative correlation is observed for statements, blocks, and methods but not for classes, where the correlation was, surprisingly, a very weak, but significant, positive correlation. The plot of mutation score vs. normalized bug-fixes for statements is given in Figure 2.2, for blocks in Figure 2.3, for methods in Figure 2.4, and for classes in Figure 2.5.

Table 2.2: Correlation between total number of bug-fixes per line and mutation score

	(a) R^2				**(b) Kendall τ_b**	
	Mean	**Low**	**High**	**p**	**Mean**	**P**
Statements	-0.12	-0.13	-0.11	0.00	-0.13	0.00
Blocks	-0.14	-0.15	-0.12	0.00	-0.19	0.00
Methods	-0.16	-0.18	-0.14	0.00	-0.14	0.00
Classes	-0.13	-0.18	-0.08	0.00	-0.10	0.00

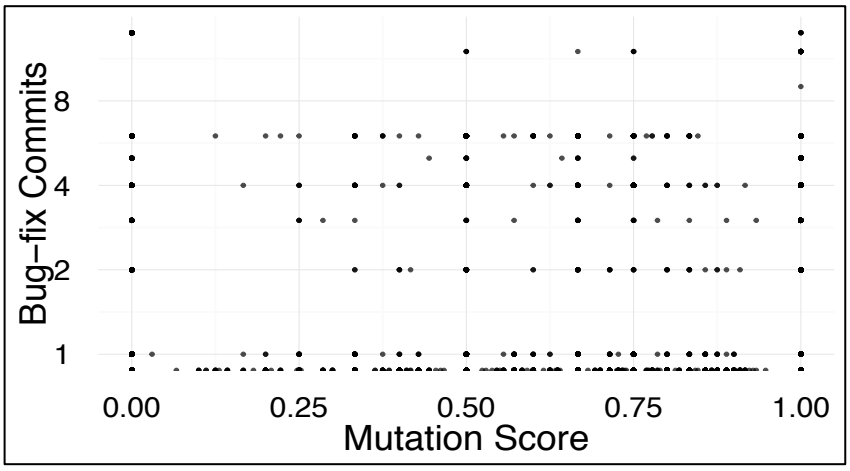

Figure 2.2: Mutation score vs. bug-fix commits for covered lines

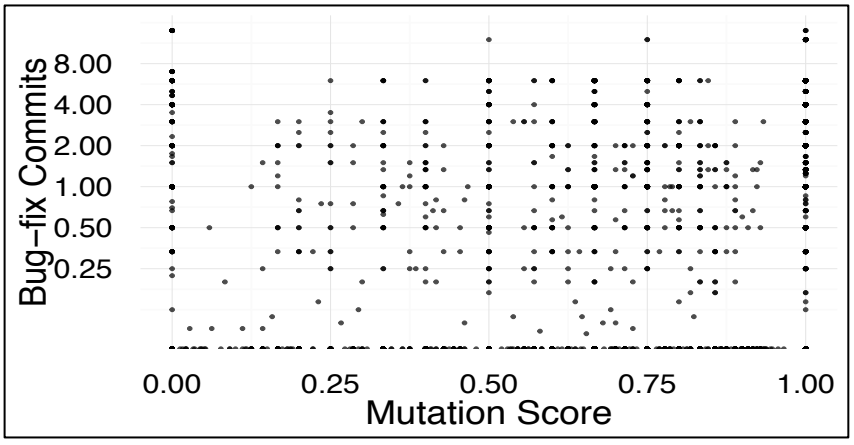

Figure 2.3: Mutation score vs. bug-fix commits for covered blocks

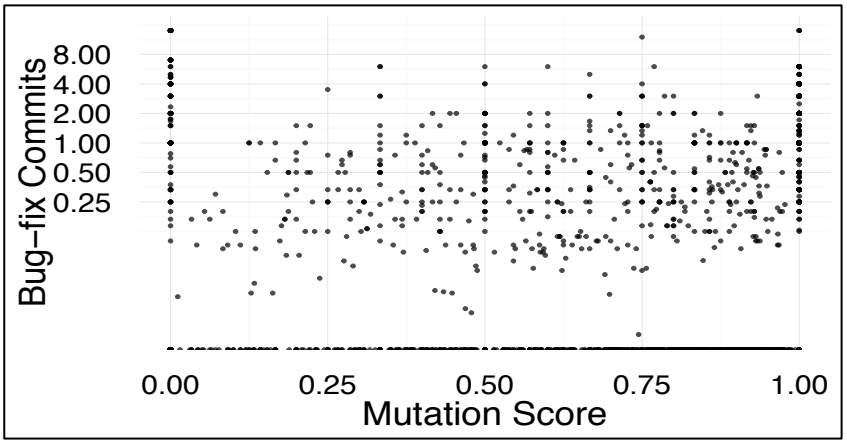

Figure 2.4: Mutation score vs. bug-fix commits for covered methods

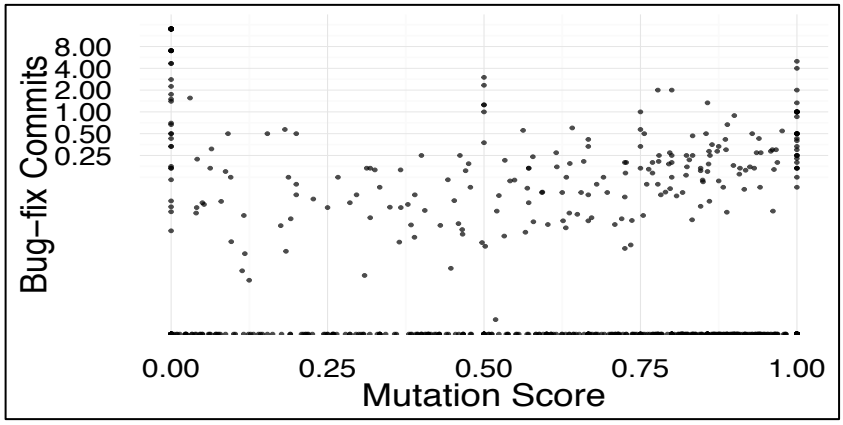

Figure 2.5: Mutation score vs. bug-fix commits for covered classes

2.4.3 Statement Coverage (λ)

The correlation between number of bug-fixes per statement and statement coverage is given in Table 2.3.

Table 2.3: Correlation between total number of bug-fixes per line and statement coverage

	(a) R^2				**(b) Kendall τ_b**	
	Mean	**Low**	**High**	**p**	**Mean**	**p**
Statements	-0.11	-0.12	-0.10	0.00	-0.13	0.00
Blocks	-0.13	-0.14	-0.12	0.00	-0.21	0.00
Methods	-0.14	-0.16	-0.12	0.00	-0.13	0.00
Classes	0.09	0.04	0.13	0.00	-0.04	0.00

For statements, blocks, methods, and classes, there was a small but statistically significant negative linear correlation between coverage and bug-fixing commits. A similar small but statistically significant negative correlation is also observed using Kendall τ_b.

These correlations (for mutant score and for statement coverage) are much lower than those seen in recent studies showing good correlation between coverage metrics and mutation scores [73, 76] (these studies are measuring a different property, but in some sense aiming for similarly strong correlations). These correlations are not so small as

to be completely devoid of value, but they do make the use of these measures dubious when comparing program elements or test suites with only small testedness differences. Unfortunately, this is a common practice in the evaluation of software testing experiments. Worse still, these results might be thought to suggest that testedness cannot be effectively measured, leaving the practicing tester without useful guidance.

2.4.4 Binary Testedness: Is It Covered?

However, using testedness as a continuous valuation, where we expect slightly more tested program elements to have fewer bug-fixes, is not the only way to make use of testedness. Instead of trying to separate very similarly tested elements, we could simply draw a line between tested and not-tested program elements. For example, common sense suggests that if testing is useful at all, then code that is not covered should probably have more bug-fixes than code that has at least some test covering it. This rationale is the intuition behind ideas like "getting to 80% code coverage," though it does not justify any particular target value. Code that isn't executed in tests is surely less tested than code that executes in even very poor tests (since even very badly designed tests with a weak oracle may catch crashes, uncaught exceptions, and infinite loops, for example).

We compared the mean number of bug-fixes for covered vs. uncovered program elements using a t-test. The results are shown in Table 2.4. By covered element we mean a program element which has at least a single statement exercised by some test case. While this is a reasonable binary distinction up the method level, a class with only a single statement covered may not be much more tested than a class that does not have any statements covered. This may account for the difference seen for classes in Table 2.4. We also note that there is insufficient data for statistical significance in classes (most classes are covered by at least some test).

Table 2.4: Difference in bug-fixes between covered and uncovered program elements

	Covered	**Uncovered**	**p**
Statements	0.68	1.20	0.00
Blocks	0.42	0.83	0.00
Methods	0.40	0.87	0.00
Classes	0.45	0.32	0.00

2.4.5 Binary Testedness: Mutation Score and Coverage Thresholds

While measuring testedness based on mutation score or statement coverage as a continuous value of limited value, we can do much better than just drawing a meaningful dividing line between covered and not-covered program elements.

We can instead evaluate whether the mean number of bug-fixes differs significantly when the tests reach a given adequacy threshold. Table 2.5,2.6 and Table 2.7,2.8 tabulate the mean number of normalized bug-fix commits per line for both above and below the thresholds μ= {0.25, 0.5, 0.75, 1.0} and λ= {0.25, 0.5, 0.75, 1.0}. We find that there is a statistically and practically significant difference between the mean number of bug-fixes for both measures at all thresholds selected (though with classes perfect statement coverage strangely becomes a predictor of more faults). Note that for individual statements, all thresholds based on statement coverage are equivalent (coverage is always 0 or 1).

Table 2.5: Mutation score thresholds

	(a) 0.25			(b) 0.5		
	$\mu \geq 0.25$	$\mu < 0.25$	p	$\mu \geq 0.50$	$\mu < 0.50$	p
Statements	0.60	1.20	0.00	0.60	1.19	0.00
Blocks	0.39	0.83	0.00	0.39	0.79	0.00
Methods	0.32	0.87	0.00	0.33	0.85	0.00
Classes	0.11	0.55	0.00	0.12	0.51	0.00

Table 2.6: Mutation score thresholds

	(c) 0.75			(d) 1.0		
	$\mu \geq 0.75$	$\mu < 0.75$	p	$\mu \geq 1$	$\mu < 1$	p
Statements	0.58	1.16	0.00	0.58	1.14	0.00
Blocks	0.39	0.71	0.00	0.39	0.67	0.00
Methods	0.34	0.81	0.00	0.41	0.75	0.00
Classes	0.13	0.46	0.00	0.20	0.40	0.00

Table 2.7:Statement coverage score thresholds

	(a) 0.25			(b) 0.5		
	$\mu \geq 0.25$	$\mu < 0.25$	p	$\mu \geq 0.50$	$\mu < 0.50$	p
Statements	0.68	1.20	0.00	0.68	1.20	0.00
Blocks	0.42	0.83	0.00	0.42	0.83	0.00
Methods	0.40	0.87	0.00	0.41	0.86	0.00
Classes	0.48	0.31	0.04	0.51	0.30	0.01

Table 2.8: Statement coverage score thresholds

	(c) 0.75			(d) 1.0		
	$\mu \geq 0.75$	$\mu < 0.75$	p	$\mu \geq 1$	$\mu < 1$	p
Statements	0.68	1.20	0.00	0.68	1.20	0.00
Blocks	0.42	0.82	0.00	0.42	0.82	0.00
Methods	0.42	0.84	0.00	0.46	0.80	0.00
Classes	0.59	0.28	0.00	0.90	0.24	0.00

Table 2.9 and 2.10 shows mutant threshold results if we first remove all program entities that are not covered. This has little impact on the ability of thresholds to predict bug-fixes.

Table 2.9: Mutation score thresholds with uncovered program elements filtered out

	(a) 0.25			(b) 0.5		
	$\mu \geq 0.25$	$\mu < 0.25$	p	$\mu \geq 0.50$	$\mu < 0.50$	p
Statements	0.60	1.16	0.00	0.60	1.11	0.00
Blocks	0.39	0.72	0.00	0.39	0.64	0.00
Methods	0.32	0.90	0.00	0.33	0.71	0.00
Classes	0.11	1.66	0.00	0.12	1.13	0.00

Table 2.10: Mutation score thresholds with uncovered program elements filtered out

	(c) 0.75			(d) 1.0		
	$\mu \geq 0.75$	$\mu < 0.75$	p	$\mu \geq 1$	$\mu < 1$	p
Statements	0.58	0.95	0.00	0.58	0.89	0.00
Blocks	0.39	0.50	0.00	0.39	0.47	0.00
Methods	0.34	0.53	0.00	0.41	0.39	0.00
Classes	0.13	0.75	0.00	0.20	0.50	0.00

2.4.6 Complexity and Change

We also compare the number of mutants, normalized by the size of the program element (dividing by the number of lines), to the number of post-epoch bug-fixes for that element.

Statements: Comparing the number of bug-fixes to the number of mutants per statement, we find that the 95% confidence interval is given by {-0.004697, 0.013204} at $p > 0.01$.

Methods: Comparing the number of bug-fixes to the number of mutants per method, we find that the 95% confidence interval is given by {-0.087117, -0.048715} at $p < 0.01$.

Classes: Comparing the number of bug-fixes to the number of mutants per class, we find that the 95% confidence interval is given by {-0.096285, -0.000863} at $p > 0.01$.

Summary: Most of the results are statistically significant. We also observe that there is a weak correlation between the number of mutants (normalized) and the number of bug-fixes. More "complex" code as measured by number of mutations has slightly fewer bug-fixes, but the correlation is even weaker than between testedness measures and bug-fixes. However, the difference in correlation is not very large, so another way to interpret this is that as a continuous measure, simple number of mutants, normalized, is only slightly worse as a predictor of bug-fixes than "testedness". However, unlike testedness measures, the number of mutants does not provide a useful binary predictor

for bug-fixes. Binary splits based on a threshold using the mean number of normalized mutants (2.79) do not produce significantly different populations. Setting a threshold of 5 or more normalized mutants does produce significant differences (p-value < 0.0001), but the means are very similar, e.g., 1.1 bug-fixes for less complex statements vs. 0.95 bug-fixes for statements with more mutants.

2.4.7 Complexity and Testedness

Statements: Comparing the normalized number of mutants to the mutation score per statement, we find that the 95% confidence interval is given by {0.008016, 0.025912} at $p < 0.01$.

Methods: Comparing the normalized number of mutants to the mutation score per method, we find that the 95% confidence interval is given by {0.005755, 0.044311} at $p > 0.01$.

Classes: Comparing the normalized number of mutants to the mutation score per class, we find that the 95% confidence interval is given by {-0.049426, 0.046223} at $p > 0.01$.

Summary: We found that at the statement level (only) there is a statistically significant but very weak correlation between the number of mutants (normalized) and the mutation score. More complex statements are (very slightly) more well-tested.

2.5 Discussion

This paper presents a novel approach to determining if what we call testedness measures actually help predict how many defects that escaped testing will be found (and fixed) in parts of a program. Our empirical results have some potentially important consequences for testing research and practice.

2.5.1 The Danger of Relying on Small Testedness Differences

First, there is only a weak correlation between either statement coverage or mutation score and future bug-fixes. This indirectly suggests that research efforts using coverage or mutants to evaluate test suite selection, generation, or reduction algorithms may draw unwarranted conclusions from small, significant differences in these measures. In particular, it may suggest that using mutation to evaluate testing experiments can potentially fail to reflect the ability of systems to detect the types of faults that are detected by practitioners and worth correcting in real-life. Given that the literature supporting the value of code coverage as a predictor of fault detection mostly relies on the ability of mutation testing to reflect real fault detection, and that mutation testing's effectiveness is validated by only a small number of studies, none of which present overwhelming evidence over a large number of programs, we strongly suggest that testing experiments, whenever possible, should rely on the use of some real faults in addition to coverage or mutation-score based evaluations. In some contexts, where detecting all possible faults is the goal (e.g., safety critical systems) and the oracle for correctness is known to be extremely good, mutation-based analyses may be justified, but even in those cases data based on real faults would be ideal.

2.5.2 Practical Application of Thresholds

On the other hand, our results show that numerous simple percentage thresholds for statement coverage and mutation score can, in a statistically significant way, predict the number of bug-fixes (with mean differences between populations of about 2x). This suggests a simple method for prioritizing testing targets in a program. The entities with the highest bug-fix counts were, unsurprisingly, those not even covered by any tests. As a first priority, covering uncovered program elements is likely to be the most rewarding way to improve testedness, since these elements can be expected to have the most potential undetected bugs that will be revealed in the near future. Surviving mutants of entities with low mutation scores can then be used to guide further testing. One obvious question is, which threshold should be used, since many thresholds seem

effective? Our data shows that it really does not matter much - the significance and even average bug-fixes are not radically different for different thresholds. The simplest answer is to start with low thresholds, keep improving testing until there are no remaining interesting elements below the current threshold, then move on to a higher threshold. Setting a particular threshold for project-level testing is not supported by our data, however, as there is no clearly "best" dividing line, only a number of ways to define "less tested" and "more tested" elements, most of which equate to more bug-fixes for less tested elements.

2.5.3 Complexity, Bug-Fixes, and Testedness

There does not seem to be any very strong or interesting relationship between complexity (as measured by number of mutants) and bug-fixes, or between complexity and testedness. More complex code is (very slightly) less fixed, perhaps because it is (very slightly) more tested. The main take-away from the complexity analysis is that the number of mutants is almost as good a predictor of lack of bug-fixes as testedness, if used as a simple correlation, but it does not support useful binary distinctions in likely bug-fixes.

2.5.4 Testing is Likely Effective

One final point to note is that our data provides fairly strong support for the idea that testing is effective in forcing quality improvements in code. Our measures of testedness are, essentially, based purely on the dynamic properties of a test suite, not on static properties of program elements (the number of mutants for an entity depends on static properties, but all statements with any mutants can achieve or fail to achieve a score of any particular threshold). This means that, without using the static properties of code, the degree to which code is exercised in a test suite can often be used to predict which of two entities will turn out to require more bug-fixes. As far as we can determine, there are only a few potential causes for this ability to use the dynamics of a test suite to predict bug-fixes:

- Some unknown property not related to code quality results in both a tendency to write tests that cover code and in fewer bug-fixes for that code.
- A known property results in both a tendency to write tests that cover code and in fewer bug-fixes for that code: namely, good developers write tests for their already more correct code. Testing itself is more a sign of good code than a cause of good code.
- Tests covering code often detect bugs, and developers fix the bugs, so the code has fewer bugs to fix.

The first possibility is, in our opinion, unlikely - it is difficult to imagine such an unknown factor. Some obvious candidate factors do not really bear up on examination. For example, perhaps code with many bug-fixes is new code, and so has not yet had tests written for it. If the act of writing tests for the new code makes it less buggy, however, then testing is in fact effective. Moreover, the predictive power of mutation score being over a threshold is present even if we restrict our domain to entities that have at least one covering test. New code might be expected to be completely untested, removing most truly new (no tests) code from this population.

The second possibility is more plausible, and may well be true to some extent. The third possibility seems most plausible, and we believe is likely to be the main cause of the observed effects. However, even if we assume that the second explanation is the primary cause for the relationships we observed, notice the peculiar consequences of this claim: developers who believe testing is worthwhile, and devote more time to it, are “wrong” in that testing itself is useless, but on the whole produce statistically better code than those who do not value testing. This may not be an appealing argument to those dubious about testing's value.

2.6 Threats to validity

While we have taken care to ensure that our results are unbiased and have tried to eliminate the effects of random noise, we cannot guarantee that our results are valid. In particular, our results are subject to the following threats.

Threats Due to Sampling Bias: To ensure representativeness of our samples, we opted to use search results from the Github repository of Java projects that use the Maven build system. We picked all projects that we could retrieve given the Github API, and selected from these only based on necessary constraints (e.g., the project must build, and tests at epoch must pass). However, our sample of programs could be biased by skew in the projects returned by Github. Github's selection mechanisms favoring projects based on some unknown criteria may be another source of error. We also handpicked some projects from Apache, such as commons-lang. As our samples only come from Github and Apache, this may be a source of bias, and our findings may be limited to open source programs. However, we believe that the large number of projects more than adequately addresses this concern.

Bias Due to Tool Used: For our study, we relied on PIT. We have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, ensuring also that the mutation operators were non-redundant (and have checked for redundancy in past work using PIT).

Secondly, we used the Gumtree algorithm discussed earlier for tracking program elements across commits. However, the algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Bias Due to Mutant Distribution: There is still a possibility that the kind of mutants produced may be skewed, which may impact our analysis.

Bias Due to Equivalent Mutants: In this study we did not apply a systematic method for the detection of equivalent mutants and also did not remove equivalent mutants. This might have impacted the mutation score of some projects where a large portion of the mutants were equivalent and were not killed.

Bias Due to Commit Classification: Our determination of commits as bug-fixes or not and of commits that "end the history" of a program element both depend on a learned classifier. While our results do not require those results to be anywhere near perfect, it may be that some unknown bias in the failures unduly influences our results, or gives rise to the weakness of observed correlations.

Bias Due to Lack of High Coverage: Some researchers have found that a strong relationship between coverage and effectiveness does not show up until very high coverage levels are achieved [68, 70, 97]. Since the coverage for most projects rarely reached very high values, it is possible that we missed the existence of such a dependent strong relationship.

2.7 Conclusion

This paper uses a novel method to evaluate the effectiveness of test suite quality measurements, which, we suggest essentially aim to capture the "testedness" of a program or program elements. Much of previous research attempting to evaluate such measures operates by a procedure that, at a suitably high level of abstraction, can be described as first collecting a large set of tuples of the form (testedness measure for suite, # faults found by suite), then applying some kind of statistical analysis. Details vary, in that suites may all be for one SUT, or for multiple SUTs (though seldom for more than 5-10 SUTs), and in most cases "actual faults" are either hand-seeded or "faults" produced by mutation testing (which is assumed to measure real fault detection on a largely recently established and still limited empirical basis [110]). These studies have produced a variety of results, sometimes almost contradictory [82]. Is coverage useful? Is mutation score (more) useful?

We propose a different approach. Measuring fault detection for a suite can be extremely labor-intensive; worse, depending on the definition of faults, we may give too much credit for detecting faults that are of little interest to most developers. Instead, our evaluation chooses a point in time, collects testedness measures (statement coverage and mutation score) for a passing test suite from that date, and then examines

whether these measures predict actual future bug-fixes for program elements. If “well tested” elements of a program require no less effort to correct, then either we are not measuring testedness effectively, or testing itself is not effective.

We assume that testing is effective. Under this assumption, we show that there is the expected negative correlation between testedness and number of future bug-fixes. How-ever, this correlation is so weak that it makes using it to compare testedness values in the continuous fashion, where slightly more tested code is assumed to be slightly better, or slightly higher scoring test suites are assumed to be better than slightly lower scoring test suite, a dubious enterprise. This suggests that the evaluation method in many software testing publications may be of questionable value. On the other hand, when we use testedness measures to split program elements into simple “more tested” and “less tested” groups, the population differences are typically significant and the mean bug-fixes are sufficiently different (usually about a factor of 2x) to provide practical guidance in testing.

So, is (statement) coverage useful? Is mutation score relevant? Is mutation score (more) useful? The answers, we believe, may be that it depends on what you expect to achieve using these methods. Testing is an inherently noisy and idiosyncratic process, and whether a suite detects a fault depends on a large number of complex variables. It would, given this complexity of process, be very surprising if any simple dynamic measure computable without human e ort for any test suite produced strong correlations like those often shown between code coverage and mutation score (0.6-0.9). The correlations between these measures are often high because both result from regular, even-handed, automated analysis of the dynamics of a test suite. Actual faults are apparently (unsurprisingly) produced and detected by a much more complex and irregular process. However, when used to draw the line between less tested and more tested program elements, testedness measures can provide a simple automated way to prioritize testing effort, and recognize when all the elements of an SUT have passed beyond a high threshold of testedness, and are thus likely to have fewer future faults. In short, while we cannot (at present) measure testedness as precisely as we (software

engineering researchers) would like, we can measure testedness in such a way as to provide some practical assistance to the humble working tester. Our data is available for inspection and further analysis at http://eecs.osuosl.org/rahul/fse2016/.

Applying Mutation Analysis On Kernel Test Suites An Experience Report

Iftekhar Ahmed, Carlos Jensen, Alex Groce, Paul E. McKenney

2017 IEEE International Conference on Software Testing, Verification and Validation Workshop on Mutation Analysis (pp. 110-115)

Chapter 3 APPLYING MUTATION ANALYSIS ON KERNEL TEST SUITES AN EXPERIENCE REPORT

3.1 Introduction

Quality is important for software systems. Unfortunately evaluating quality, especially the presence of bugs, becomes more difficult as the complexity of that software increases. The best way to ensure software meets quality requirements is to engage in extensive and thorough testing. The goal of testing is to discover faults in the System Under Test (SUT) by executing tests that create conditions that lead to failure, and detect these.

Two important "problems" with testing are knowing when you have performed sufficient testing and determining whether your testing is biased. A well-respected technique for evaluating test suites and test coverage is mutation analysis. Unlike code coverage and other benchmarking techniques, mutation analysis addresses the oracle problem as well as determining the degree to which SUT behaviors are explored. By inserting random but realistic bugs [9, 48, 110] in the SUT, known as mutants, we can determine a tests ability to uncover faults, not just its ability to explore behavior. The ratio of mutants found over all mutants, is used as the test suite's effectiveness (mutation score). Mutation analysis thus identifies gaps in the test suites and subsumes almost every other test adequacy criteria [18, 70, 169, 198].

One of the primary reasons mutation analysis is rarely used on large and complex programs is that mutation analysis generates large numbers of mutants, which must be analyzed, though mutant sampling [176, 209, 215] and mutant execution optimizations [109, 177, 216] can help to mitigate the problem. Another reason is the lack of proof of mutation analysis' applicability to complex real world projects. Thus, mutation analysis has been more widely adopted by academia than industry, and the technique mostly evaluated using relatively simple programs and test suites.

The Linux kernel is one of today's more complex software systems, evolving so rapidly that maintaining quality assurance is hard [64, 174]. Applying techniques such as code analysis and model checking on the kernel and its modules is difficult because of their size and the complexity of the code. Source code analysis generates a large number of false positives and warnings [106] which need to be screened by domain experts, which is expensive in terms of both time and resources.

Although mutation analysis can be applied to the kernel, it is not trivial to do so. First, one has to generate an enormous set of mutants. Second, one needs to compile and run each mutated version of the code and put it through the test harness. As with many complex systems, execution is probabilistic, meaning that the amount of time needed to "kill" a mutant (detecting it with the test suite) cannot be determined a priori. Not finding a fault at time j could be due to inefficient tests, or insufficient run time. Therefore, to avoid false positives, a test suite would need infinite run-time on each mutant. This is clearly not feasible, and a probabilistic approach must be adopted.

This paper describes our experience using mutation testing on the Linux-kernel's RCU [183]. Our goal was to determine whether: (1) mutation testing RCU is feasible, and (2) whether it can uncover bugs in RCU. Locating bugs in RCU is hard because RCU is well tested and heavily used: About one in 2,000 lines of kernel code uses RCU [146], and it has been a favorite target for model checkers [4, 81, 135, 138]. If mutation testing can locate new bugs, these are likely deep, and the technique can be useful for locating bugs in other complex software.

3.2 Background

3.2.1 Limitations of Mutation Analysis

It is infeasible to exhaustively test a test suite, as this would mean running it on all possible programs. Even running it on all mutation operators applied to all statements of a program is non-trivial. This only gets worse as the size and complexity of a code-base grows, leading to a combinatorial explosion [105]. Mutation testing is even

costlier for concurrent code because it must be tested against thread schedules and memory reordering.

One way to tackle complexity is reducing the mutants used. Kintis et al. [124] defined the notion of disjoint mutants, i.e., a set of mutants that subsumes all the others. Ammann et al. showed that minimal mutants [6] reduce the count. Kaminski et al. [115, 116] and Just et al. [111] used fault-based predicate testing to reduce redundancy of relational and logical operators. Sampling [176] and searching higher order mutants [90, 103, 129] also helps.

Mutation can leave semantics unchanged. These "equivalent mutants" skew results, and full detection is undecidable [33]. Manual equivalence inspection [212] requires significant effort and provides an identification rate of only 8% [1]. Automated equivalence heuristics are thus attractive, and useful solutions have been proposed [164]. One approach is to use the compiler to detect equivalent mutants [20]. If the original program and a mutant compile to the same object code, no test could reveal a difference. Mothra [164] showed that this identifies about 45% of equivalent mutants. Recently Papadakis et al. proposed Trivial Compiler Equivalence (TCE) which groups mutants with identical object code into equivalence classes [178], addressing the "duplicate mutant" problem. We used TCE to identify both equivalent and duplicate mutants.

3.2.2 Read Copy Update (RCU)

The RCU module of the Linux kernel is a synchronization mechanism that allows lightweight readers [144]. RCU read-side critical section entry/exit overhead can be exactly zero [144], excellent for read-mostly workloads [84, 144, 147]. However, RCU updaters cannot exclude readers, and must take care to avoid disrupting readers. Updaters typically maintain versions of the part of the structure being updated, reclaiming old versions only when safe.

RCU use in the Linux kernel has gone from 0 in 2002 to over 6,500 calls in 2013 [146]. RCU pervades the kernel, with one in every 2,000 lines using an RCU primitive [146]. Given the complexity of the code, and its importance, researchers have applied model checking techniques to RCU [4, 81, 126, 138], and the RCU test harness (rcutorture) is very well developed.

The bulk of RCU is in 4 files (srcu.c, tiny.c, update.c and tree.c). Together, these only total to 5,542 lines of code (LOC), with the largest being 3,771 LOC. The RCU is therefore not the largest program examined using mutation analysis (Apache Commons Math, with 202,000 lines of code, was analyzed by Gopinath et al. [78]), but it has the highest complexity, as Apache commons is a large but shallow set of library calls.

RCU's primary test system, rcutorture, is an automated stress-testing mechanism composed of 1,800 lines of code. rcutorture can simulate 12 different RCU scheduling variations and test on 16 hardware configurations. These configurations are specified using parameters such as CONFIG_NR_CPUS, CONFIG_HOTPLUG_CPU, CONFIG_SMP, etc. rcutorture uses Qemu to load kernels built using these parameters and monitors their performance for a user specified period. The test periodically outputs status messages via printk(), which can be examined via the dmesg command. Qemu uses KVM, essentially running a virtualizer (Qemu) on top of another virtual machine, a practice referred to as nested virtualization [161]. Interest in rcutorture has grown, with the number of contributors growing from 5 to 9 between 2006 and 2014.

Time dependent and stochastic testing systems such as rcutorture are common for critical systems code [173]. The longer you run rcutorture, the higher the chances of finding a bug, if present. This means that non-trivial mutants need to be run for very long periods of time. Because RCU is used on large clusters and has been extensively tested, most remaining bugs are likely to be in difficult-to-reach parts of the code.

3.3 Methodology

3.3.1 Mutation Generation

We used the tool developed by Andrews et al. [9] to generate mutants. We decided to use this tool as it was evaluated on a set of eight well-known subject programs, part of a Siemens suite [9]. The tool is also simple in design and implementation; a 350 LOC Prolog program and a shell script. This tool generates mutants from a source file, treating each line of code in sequence and applying four classes of "mutation operators". Every valid application of a mutation operator to a line of code results in a mutant being generated in a separate file. The four classes of mutation operators are given in Table 3.1.

Table 3.1: Mutation operators

Mutation operators Name	Description
rep_const	Replace integer constant C by 0, 1, -1, ((C) +1), or ((C)-1)
rep_op	Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class
Negate	Negate the decision in an "if" or "while" statement
del_stmt	Delete a statement

The first three classes are considered "sufficient" mutation operators (i.e., a set S of operators such that test suites that kill mutants in S tends to kill mutants formed by a broader set) [168]. The fourth operator handles pointer-manipulation and field-assignment statements that are not vulnerable to any of the sufficient mutation operators [8]. Table 3.2 contains some sample mutants from RCU and Table 3.3 contains the details of mutants for each mutation operator category.

Table 3.2: Mutation examples from RCU

Name	Original Version	Mutated Version
rep_const	if (rnp->qsmask ==1)	if(rnp->qsmask !=1)
rep_op	for (i = 0; i >= RCU_NEXT_SIZE; i++)	for (i = 0; i== RCU_NEXT_SIZE; i++)
Negate	if (rcu_batch_empty(b))	if(!(rcu_batch_empty(b)))
del_stmt	struct rcu_head *head;	

After applying the mutation generator to each of RCU's files (less than 5 minutes for all files), the next step was to compile the 3,169 resulting mutated versions of RCU. For scalability reasons, we did this and all stress testing on virtual machines built on the ESXi 5.5 platform [62].

After compilation, we had to test each of the mutants. Running this testing serially would take excessive amounts of time. The kernel cannot run as a thread, so we could not use threads to parallelize the testing. The logical step was therefore to use virtual machines. We used 4 virtual machines running in parallel, each of which had 2x 2.7GHz CPUs (x86_64 architecture), with 2 threads per CPU, and 4 GB memory. We used with RCU in Linux kernel version 3.18.5.

3.3.2 Reducing The Test Space

We had to reduce the number of mutants as much as possible and as early as possible. We trivially discarded the 354 (11.1%) which failed to build (mutation tools sometimes produce code which is syntactically nonsensical, e.g. changing parameters to a function call). Next we compared each mutants object code against that of the original code (to identify equivalent mutants) and to that of every other mutant (to identify duplicate mutants).

3.3.3 Running rcutorture On Mutants

The next step was to run the mutated RCU's to determine if rcutorture would flag them. Because execution and detection of faults is probabilistic, we allocated relatively short timeouts (2 minutes). We hypothesized that most faults would be trivially detected, while a handful of faults require very long runtimes. Our goal was to narrow the set as quickly as possible to then allocate more time and resources to the hard mutants.

Each virtual machine was assigned to handle one specific mutant. rcutorture uses Qemu to load different versions of the kernel, built using permutations of a set of parameters On each virtual machine, 14 parallel processes were set up to compile 14 different kernel images using these parameters. This helped us to cut the setup time down by 1/14. Next, a single sequential process would load the images on Qemu and monitor the thread for 2 minutes. We used a single process because all Qemu processes were killed after 2 minutes, which would kill all instances of Qemu. If we had run 14 Qemu instances in parallel, all would be killed when the first finished.

3.4 Analysis

Once the 2 minutes were up we parsed the logs generated by rcutorture for strings like "Assertion failure", "Badness", "WARNING:", "BUG", "!!!," etc. These are coded into the Linux kernel and rcutorture to indicate a failure. We treated mutants triggering such warnings as killed, and mutants that did not generate any warnings as surviving. The only exception was when a mutant caused the kernel to fail to execute.

While we expected to run the surviving mutants with longer and longer test durations, the list of surviving mutants was so small that manual inspection could be performed, which suggests that given a good testing framework like rcutorture, inspecting and checking surviving mutants (and determining true survivors) may be less onerous than expected.

We compiled the list of mutants and sent them to a human “oracle” (a maintainer of RCU and co-author of this paper) for inspection. The oracle examined each surviving mutant to determine if there was a test that would eventually catch the mutant, or whether there was a gap in the test harness. When deficiencies were identified, new tests were built, and the RCU was tested to determine if the gap was masking a bug.

3.5 Result

3.5.1 Mutant Attrition

Table 3.3: Mutants in mutation operator category

File	del_stmt	negate	rep_const	rep_op
srcu.c	116	17	72	45
tiny.c	86	12	47	37
update.c	126	25	131	61
tree.c	858	173	732	631
Total	1,186	227	982	774

Figure 3.1 shows the percentage of mutants that survived the build process by file. We see that invalid mutants were relatively evenly distributed across the 4 files.

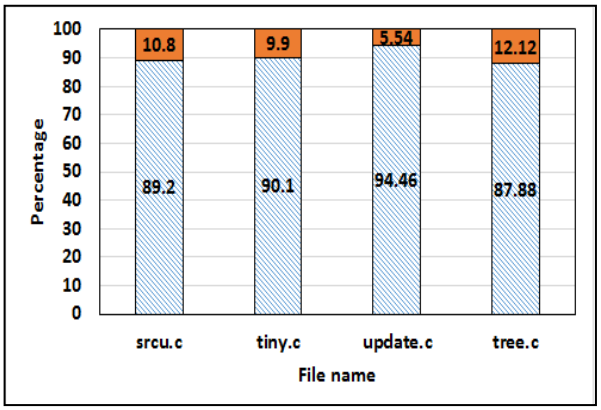

Figure 3.1: % of mutants failing/surviving build process (Fail: top of bars)

Applying the TCE, we found that about 70% of buildable mutants were unique (Figure 3.2). Surprisingly we found a disproportionate number of equivalent in update.c. Of the 2,815 total buildable mutants, 2,150 were unique.

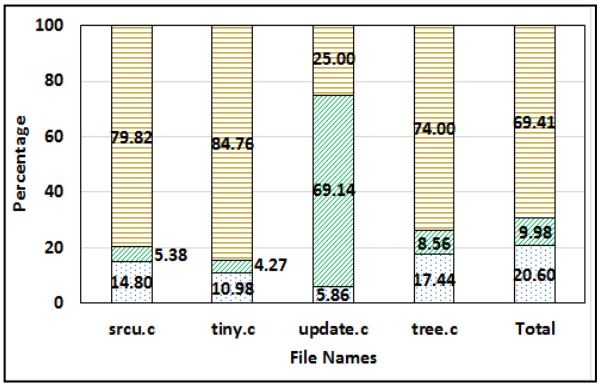

Figure 3.2: % of equivalent, duplicate and unique mutants in build surviving mutants (Top: unique, middle: equivalent, bottom: duplicate)

Next we ran our 2-minute test runs of rcutorture on all unique mutants. We found that only 380 mutants (17.7% of unique buildable mutants, 12.0% of generated mutants), survived (not identified as bugs by the test harness). Figure 3.3 shows the attrition of mutants for each process stage. These were passed on to our human oracle for manual inspection. After manual inspection, our oracle identified 3 weaknesses in rcutorture. Of the 3 failures, 2 were determined to conceal bugs in RCU itself (see Section V).

3.5.2 Time investment

Generating mutants was trivial, and took on the order of ~150 seconds. It took ~30 minutes to compile each mutated version of the Linux kernel on the machine we used. This process can be parallelized (up to one kernel build per machine/VM). We used the diff command to identify duplicates, which took ~1 second to calculate each diff.

$$N + \sum_{i=1}^{4} n_i^2 \qquad (1)$$

Equation (1) calculates the number of diffs performed, where N is the number of mutants and n_i is the number of mutants in each file (each mutant has to be compared to the gold standard, then to each other mutant).

Each of the 2-minute rcutorture test runs had to go through a setup phase, generating a set of scripts and building an image of the kernel with a specific configuration to load

in Qemu. This one-time setup took ~30 minutes, which preceded each of the 2 minute runs. These images can be reused for longer runs.

Finally, there is the human time investment. Estimating this is harder, since the analysis was performed on a catch-as-catch-can basis as new results arrived. A good approximation is 5 minutes per-mutant, but with very large variance. Some mutants were automatically understood as of no interest, while others required much more effort to analyze (but these also included the most beneficial, the ones resulting in patches). A good estimate for overall human effort is 25 hours.

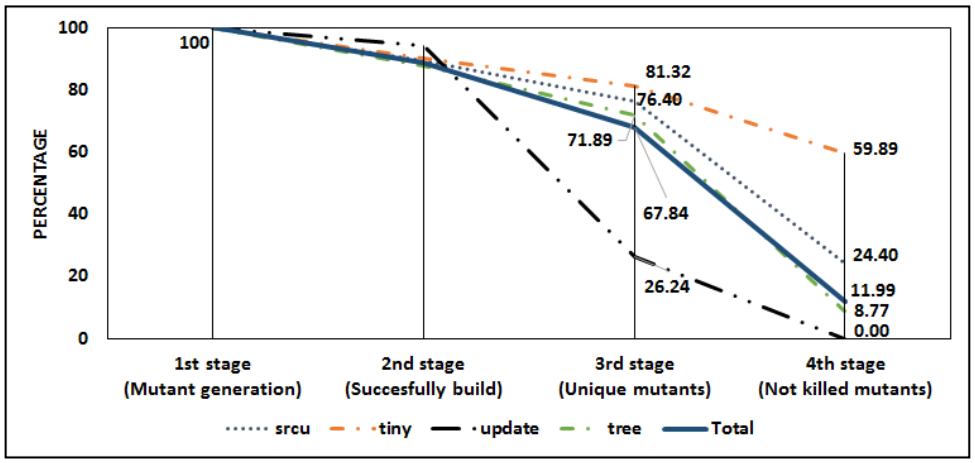

Figure 3.3: Percentage of mutant surviving after every stage of processing

3.6 Resulting Patches To RCU

In this section we list the patches that resulted from our application of mutation analysis on RCU along with a brief description. All patches can be accessed using the provided footnotes.

3.6.1 Patch 1: rcutorture: Test SRCU cleanup code path.

Details: An rcutorture memory leak of the dynamically allocated ->per_cpu_ref per-CPU variables was identified via our mutation analysis. This commit adds a second form of srcu (called srcud) that dynamically allocates and frees the associated per-CPU variables. This commit also adds a cleanup() member to rcu_torture_ops that is invoked

at the end of the test, after ->cb_barriers(). After the patch, the SRCU-P torture-test configuration selects scrud instead of srcu, with SRCU-N continuing to use srcu, thereby testing both static and dynamic srcu_struct structures[8].

3.6.2 Patch 2: rcutorture: Test both RCU-sched and RCU-bh for Tiny RCU

Tiny RCU provides both RCU-sched and RCU-bh configurations, but only RCU-sched was tested by the rcutorture previously. This gap was identified via mutation analysis on tiny.c. This commit changed the TINY02 configuration to test RCU-bh, with TINY01 continuing to test RCU-sched[9].

3.6.3 Patch 3: rcu: Correctly handle non-empty Tiny RCU callback list with none ready

This fixes an RCU bug. This bug is most likely to occur if there is a new callback between the time rcu_sched_qs() or rcu_bh_qs() is called before __rcu_process_callbacks() is invoked. This bug was detected by the addition of RCU-bh to rcutorture[10].

3.6.4 Patch 4: rcu: Don't redundantly disable irqs in rcu_irq {enter,exit}()

This replaces a local_irq_save() and local_irq_restore() pair with a lockdep assertion which removes the corresponding overhead from the interrupt entry/exit fast paths. This change was introduced because mutation testing showed that removing rcu_irq_enter()'s call to local_irq_restore() had no effect, indicating interrupts were always disabled[11].

---

[8]http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=ca1d51ed9809a99d71c23a343b3acd3fd4ad8cbe

[9]http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=f13bad9042dcf9b60b48a0137951b614a2ee24b

[10]http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=6e91f8cb138625be96070b778d9ba71ce520ea7e

[11]http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=7c9906ca5e582a773fff696975e312cef58a7386

3.6.5 Patch 5: rcu: Make rcu_gp_init() bool rather than int

Mutation testing showed that the return value from rcu_gp_init() is always used as a boolean, so this commit makes it a Boolean[12].

3.7 Discussion

Following the above process, we were able to narrow 3,169 mutants to only 380 potentially interesting mutants with little or no human intervention, using modest compute resources (3,499 hours of runtime on a normal machine, a load which is very parallelizable). While 380 may seem like a large number, it is very likely that this could be further reduced by giving rcutorture more run-time to try to kill these mutants. We look at our process as a kind of mutation analysis pre-processing, where we, as quickly as possible, with maximum automation, narrow the field of mutants to the set of interesting mutants.

We found that code that calculates heuristics and error-recovery timeouts can be surprisingly robust to mutations, and adding tests that kill these mutants would lead to more false positives under heavy load or other extreme conditions. Similarly, mutants that cause small degradations in throughput or real-time response may prove difficult to kill. Finally, test suites for algorithms with some degree of redundancy may find it difficult to kill mutants that disable only a subset of the redundant code paths. For example, RCU has a number of quiescent states, including the context switch, the idle loop, usermode execution, and offline CPUs. A mutant that disables detection of any one type of quiescent state will likely survive testing because one of the other types of quiescent states will likely be encountered sooner rather than later.

Bugs found using rcutorture are often non-deterministic. Some may occur only after extremely long runtimes (~1,000 hours). To obtain perfect confidence, rcutorture needs to run for a very long time, which is impractical. Instead, the approach we advocate is

[12]http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=45fed3e7cfb4001c80cd4bd25249d194a52bfed3

to narrow the field of candidates so that either enough machine resources are available, or a human oracle can reasonably inspect and evaluate each case. Our goal is to determine how the set of mutants is further narrowed by longer and longer runtime windows. Because rcutorture and kernel testing is a non-deterministic process, it is likely the case that a set of short runs is more efficient for killing mutants than longer runs. We will investigate this in our future work.

Given the complexity of RCU, one could expect to see most mutants fail during compilation. However, only 11% of generated mutants failed to build. Most of these failing mutants came as a result of mutating function or other parameters in a way that causes a conflict, which the compiler will catch. This is an indication that the mutation framework is doing a reasonably good job of only creating plausible mutants rather than randomly changing tokens in the code. For a simpler application, we'd expect to see an even lower failure rate.

One might hope that a perfect test suite would kill all mutants, but that is unlikely. First is the issue of equivalent mutants. Though we tried to factor most of these out, some cannot be caught by any automated method. For instance, it is common in C to use an integer as a boolean, where 0=false and any non-zero value =true. Mutating one non-zero value to another non-zero value will result in an equivalent program which cannot be detected using diff, depending on how the value is used, and which cannot be killed by a valid test case (due to no semantic difference).

We found that about 10% of our mutants were equivalent, which is close to the findings of Papadakis et al. [178] when they looked at 18 programs. We found that about 20% of the mutants were duplicate mutants, which is also close to their findings. When we look at unique mutants in each file we see that tree.c has the highest percent of unique mutants (74%). This is the biggest file, with 101 functions. tree.c implements a large part of RCU's synchronization.

Any mutant affecting a portion of the program that is conditionally compiled out will "survive," as it is not present in the object code. This usually indicates that the test

suite needs to be expanded to include a configuration that compiles and tests the code affected by such a mutant. Similarly, a mutant affecting dead code will survive, but also indicates that the test suite's coverage needs to increase, for example, by including a greater variety of inputs, or, that the code should be removed. In the case where a greater variety of input is required, some sort of randomized testing (e.g., as provided by American Fuzzy Lop (AFL) [5]) can be useful. These last categories of mutants are normally the most productive in terms of improving the test suite. For example, the rcutorture tests for Tiny RCU failed to test callback handling. Fixing rcutorture to cover callback handling by applying patch 2 located a bug in callback handling which was later fixed by applying patch 3.

3.8 Threats to validity

We used the tool by Andrews et al. [9] to generate mutants. Using different mutation operators or tools could lead to different results. Our study looked at a program written in C, so additional studies on large projects in other programming languages would be needed to verify the same benefits there.

Other threats are due to the use of potentially faulty software. We used gcc to identify equivalent mutants, but the gcc compiler and diff utility may have defects. However, these systems are heavily tested and deployed, so it is unlikely that they would have such grave defects as to influence our results. We used nested virtualization and that might impact the performance of the guest kernels, but not rcutorture.

3.9 Conclusion and Future work

The main contribution of the paper is an investigation of how to apply mutation analysis on a complex software system, as well as demonstrating the value of doing so, even on well-tested systems. While mutation testing can generate a lot of random results, this randomness can be quickly and efficiently triaged, and a human oracle can concentrate on a small number of interesting cases. We found that mutation analysis can uncover interesting instances of weak testing, even in a robust system like

rcutorture. While a fairly large number of mutants were left alive after our initial run, subsequent runs should further reduce the surviving mutants.

An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts

Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Anita Sarma, Carlos Jensen

2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 58-67)

Chapter 4 An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts

4.1 Introduction

Modern software systems are becoming more and more complex and requires a large development team to develop and maintain. Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management, branching, and merging. This enables large teams to work together efficiently. But it has been shown that this process is sometimes halted when isolated private development lines are synchronized and the developer runs into merge conflicts. Conflicts distract the developers as they have to interrupt their workflow to resolve them. Developers have to reason about the conflicting changes and find an acceptable merging solution. This process of conflict resolution can itself introduce bugs. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions [49, 162] which might degrade the quality of the merged code.

Researchers have looked at many ways of preventing merge conflicts, and make developer's lives easier when they do occur. Researchers have proposed workspace awareness tools [22, 47, 92, 188, 192] that help prevent merge conflicts by making the developers aware of each other's changes. Also, new merge techniques [13, 14, 117] have been proposed that would reduce the number of merge conflicts. However, little research has been devoted to the causes of merge conflicts. Are there any endemic issues that arise from the design itself? We are interested in knowing whether the design of the codebase has an effect on the merge conflicts and what is its impact on the overall quality.

Just like merge conflicts, bad design can inflict pain on developers. Bad design makes maintenance and future changes difficult and error prone. Code smells, an indication of bad design, imply that the structure of the code is badly organized. This can lead to

developers stepping on each other's toes as they make their changes. This, in turn, can lead to merge conflicts.

If there are "fundamental flaws" in the design itself, as the project grows, and the codebase grows in size and complexity, understanding and working around these "rough spots" becomes more challenging. Thus, the chances of creating a conflict increases because of the need to generate workarounds. This means that as projects grow, merge conflicts should be more likely to occur, especially around the smelly parts of the code. We aim to examine whether there is a correlation between the two, to examine whether such a link is credible.

In order to evaluate the design we look at the code smells [130]. We investigate if there is a connection between entities that contain code smells, the code smells they contain, and the merge conflicts that surround the smelly entities.

It is important to note that not all smells are created equal. Some might be more associated with a merge conflict than others. For example, a class is considered a God Class if it contains an oversized part of the entire functionality of the final product. Therefore, any changes have a high likelihood of involving changes in the God Class. When multiple developers are working, they all have a high likelihood of touching the God class. This can easily lead to merge conflicts down the road. If the changes involved are not trivial then the task of merging them will be not trivial as well.

In this paper, we investigate the following questions:

RQ1: Do program elements that are involved in merge conflicts contain more code smells?

RQ2: Which code smells are more associated with merge conflicts?

RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

To answer these questions, we investigated 143 projects. Across them, we had 36,122 merge commits, out of which 6,979 were conflicting. We identified 7,467 code smells instances across our whole corpus. We found that merge conflicts involved more "smelly" program elements than merges that did not conflict. Our results also show that not all code smells are created equal. Some are more likely to cause problems than others. When we looked at the difficulty of merge conflicts, we found that some of the smells are more likely to be involved in semantic merge conflicts than others. Finally, we found that code smells have a negative impact on code quality.

4.2 Related Work

4.2.1 Code smells and their impact

Various measures of software quality have been proposed. Boehm et al. [27], and Gorton et al. [79], to mention a few, have explored measures including completeness, usability, testability, maintainability, reliability, efficiency etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality [140, 141], though smells are often focused on future maintainability issues. The concept of code smells was first introduced by Fowler [67]. Code smells are symptoms of poor design and implementation choices [67] in code base which eventually affect the maintainability of a software system [128]. Studies also showed that there is an association between code smells and bugs [133, 170] and code maintainability problems [67]. Code smells also leads to design debt. Zazworka et al. [213] found that the God Class smell is related to technical debt. Ahmed et al. [3] found how software gets worse over time in terms of design degradation. They analyzed 220 open source projects in their study and confirmed that ignoring the smells leads to "software decay".

Researchers have proposed many different approaches for detecting code smell, such as metric based [53, 54, 130, 133, 140] and meta-model based [153]. Researchers used different techniques for identifying code smells. Fontana et al. [65] used machine

learning techniques for detecting code smells. Researchers also used both static analysis [53, 54, 133] and techniques that rely on the evaluation of successive versions of a software system [112, 130, 175].

4.2.2 Work related to code smells and bugs

Researchers have also considered the relationship between the presence of code smells and bug appearance in the code base. Khomh et al. [119] showed that classes affected by design problems ("code smells") are more likely to contain bugs in the future. Hall et al. [88] also found relationships between code smells and fault-proneness. According to their study some code smells indicate fault-proneness in the code base but the effect size is small (under 10%). Zazworka et al. [213] found that God Classes are fault-prone in some cases. Li et al. [133] also studied the relationship between code smells and the probability of faults in industrial systems, and found that the Shotgun Surgery smell was correlated with a higher probability of faults. To the best of our knowledge no work has tried to research on the relationship between code smells and how it impacts collaborative work flow, specifically merging individual works.

4.2.3 Merge conflicts

Several studies have been done on identification of conflicts and developers' awareness about potential conflicts. Awareness is frequently defined as an understanding of the activities of others to give a context for one's activities [60], which is a very important issue in Global Software Engineering (GSE) [187]. Researchers have looked at different techniques of avoiding merge conflicts by increasing the developer's awareness of the changes others made to the source code. Biehl et al. [22] proposed FastDash, which sends notifications about potential conflicts when two or more developers are modifying the same file. Another awareness tool called Syde by Hatori et al. [92] consider the source code changes at Abstract Syntax Tree (AST) level operations to detect conflicts by comparing tree operations. Da Silva et al. [47] introduced Lighthouse, which is another tool for increasing awareness among

developers about the conflict. Palantír by Sarma et al. [188] detects the changes made by other developers and show them in a graphical, non-intrusive manner. Servant et al. [192] also presented a tool and visualization that can be used to understand the impact of developers' changes to prevent indirect conflicts.

Guimaraes et al. [83] introduce WeCode which continuously merges uncommitted and committed changes in the IDE to detect merge conflicts as soon as possible. Brun et al. [31] used the similar approach in Crystal, to detect both direct and indirect conflicts. A software development model presented by Dewan et al. [58] aims to reduce conflicts by notifying developers who are working on the same file.

4.2.4 Work related to merge conflict resolution

Researchers have also studied different ways of managing the merge of developers' changes to efficiently resolve conflicts. This resolution could be either in an automated way or by preserving and presenting a useful context for the developer trying to resolve the conflict. A comprehensive survey of merge approaches was done by Mens [149]. Apel et al. [13, 14] presented a merging technique called semistructured merge. This considers the structure of the code which is being merged. Operation based merging by Lippe et al. [136] considers all the changes performed during development, in addition to the result, when merging.

Kasi and Sarma [117] present a technique of avoiding merge conflicts by scheduling tasks in a way that the probability of a conflict is minimized. SafeCommit by Wloka et al. [208] uses a static analysis approach to identify changes in a commit with no test failure. They proposed to use this approach when detecting indirect conflicts.

4.2.5 Conflict categorization

Researchers have come up with different ways of categorizing conflicts. Sarma et al. [188] grouped conflicts into two categories. One is direct conflicts, where the changes conflict directly. The other is indirect conflicts, where the files don't conflict directly, but integrating the changes cause build or test failures. Similarly, Brun et al. [31],

categorized conflicts as first level (textual) conflicts and second level (build and test failure) conflicts. Buckley et al. [32] proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level.

4.2.6 Tracking code changes and conflicts

Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. [36] proposed an algorithm that uses Levenstein edit distance to compute similarity of lines, matching "chunks" of changed code. Zimmerman et al. [219] proposed annotation graphs which works at the region level for tracking lines. Godfrey et al. [75] described "origin analysis", a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match before applying more expensive techniques to track code entities. Finally, Kim et al. [122] propose an algorithm, SZZ, for tracking the origin of lines across changes.

4.3 Methodology

Our goal was to identify the effect of design issues on merge conflicts and the quality of the resulting code (whether these changes are associated with bug fixes or other improvements.)

Here we discuss the various steps of collecting data: (1) selecting the sample of projects for the study, (2) identifying which merge commits lead to merge conflicts, (3) tracking the lines of code through different versions and merges to investigate how the code evolved and which lines were associated with conflicts, (4) identifying code smells at the time of the conflicting merge commit. Next, we determine the nature of the code updates (e.g. was the commit a result of a bug fix or a new feature etc.) taking place on those lines. In order to do this, we manually classify a subset of the commits

as bug-fix related or other. We train a machine learning classifier to classify the rest. Finally, we build a model to predict the total number of bug fixes that would occur on a conflicting line that also contained a code smell. The following subsections describe each of these steps in detail.

4.3.1 Project Selection Criteria

We wanted to make sure that our findings would be representative of the code developed in real world, thus we selected active, open source projects hosted in GitHub. We decided to use Java as the language of focus. This decision was influenced by 2 factors: First, Java is one of the most popular languages (according to the number of projects hosted on Github and the Tiobe index [203]). The second was the availability of code smell detection tools for Java, as compared to other programming languages. Further, for ease of building and analyzing the code, we select projects using the Maven [12] build system.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects our AST walker, implemented using the GumTree algorithm [63], could not handle. This left us with a total of 200 projects.

Next, we removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code. We also removed projects that had no merge conflicts. These selection criteria were used, since we are interested in the effect of design issues and merge conflicts in moderately large, collaborative projects. Our final data set contained 143 projects. Table 4.1 provides a summary of features and other descriptive information of the projects in our study.

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [50]. Table 4.2 has the summary of the domains of the projects.

Table 4.1: Project statistics

Dimension	**Max**	**Min**	**Average**	**Std. dev.**
Line count	542,571	751	75,795	105,280.1
Duration (Days)	6,386	42	1,674.54	1,112.11
# Developers	105	4	72.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

Table 4.2: Distribution of Projects by domain

Domain	**Percentage**
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

4.3.2 Code smell detection tool selection

We chose to use InFusion [98] to identify code smells because it has been found to identify the broadest set of smells [66]. Researchers have found that the metric-based approach identified by Marinescu [141] has the highest recall and precision (precision: 0.71, recall: 1.00) for finding most code smells [191]. InFusion uses this same principle and set of thresholds for identifying code smell, which was another reason for using InFusion. Researchers [3] have evaluated the smell detection performance of InFusion where they found it to have precision of 0.84, recall of 1.00 and an F-measure of 0.91.

4.3.3 Conflict Identification

Since Git does not record information about merge conflicts, we had to recreate each merge in the corpus in order to determine if a conflict had occurred. We used Git's default algorithm, the recursive merge strategy, as this is the most likely to used by the average Git project. From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. The average number of merge commits was 253. Out of all the merges, 6,979 (19.32%) were identified as leading to a conflict. The distribution of merge conflicts is shown in Figure 4.1. We see that projects experience an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the developer experience.

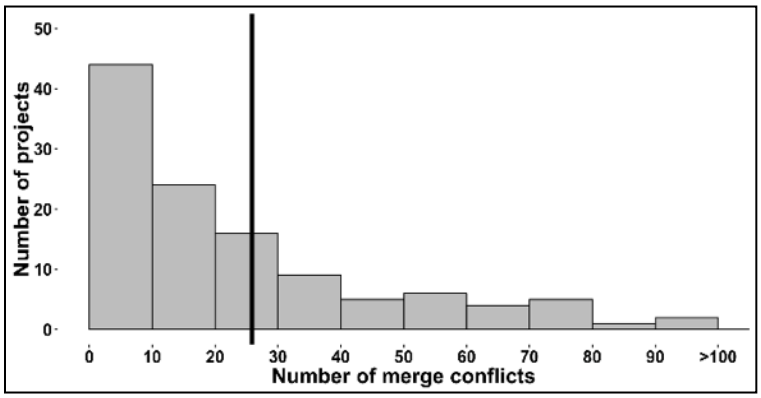

Figure 4.1: Distribution of merge conflicts. The vertical line represents the mean (25.86)

We then collected statistics regarding each file involved in a conflict. We tracked the size of the changes being merged, the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). To determine the AST difference, we used the Gumtree algorithm [63]. We also tracked the number of authors involved in the merge.

4.3.4 Conflict Type Classification

To answer our second research question, we needed to categorize the conflicts based on the type of changes (e.g., whitespace or comment added vs. variable name changed).

We identified two categories of conflicts. The first one being semantic conflicts which requires understanding the program logic of the changes in order to successfully resolve the conflict. The other type of conflict is non-semantic which easier and less risky to resolve since they do not affect the programs' functionality. We manually classify 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). Two of the authors coded 300 of these commits using qualitative thematic coding [63]. They achieved an interrater agreement of over 80% on 20% of the data: we obtained a Cohen's Kappa of 0.84. Having reached an agreement, one of the authors classified the remaining 306 commits. The codes and their definitions are given in Table 4.3.

Table 4.3: Conflict Categories

Category	Definition	Example
Semantic	Conflicts involving semantic changes	A refactoring and a bug fix involving the same lines.
Non-Semantic	Conflicting changes in formatting/comments	One of the branches contains only formatting changes (whitespace).

To train the classifier (to differentiate between semantic and non-semantic commits) we use a set of 24 features, including: the total size of the versions (LOC) involved in a conflict, the number of statements, methods and classes involved in the conflict. Details of the features are in the accompanying website [43]. We use the set of 606 (10%) commits as training data for a machine learning classifier. We used Adaptive Boost (AdaBoost) ensemble classifier that can only be used for binary classes. We categorized the 6,979 conflicting commits. We use 10-fold cross-validation to test the performance of our classifier. The precision of predicting the semantic conflicts is high at 0.75.

4.3.5 Measuring Code Smells and Tracking Lines

For each of the 6,979 conflicting commits we collected the code smells that were associated with a conflict. We needed to track them to measure the effect of having the smell and being involved in a conflict on the quality of the resulting code.

We use GumTree [63] for our analysis, as it allows us to track elements at an AST level. This way we can track only the elements that we are interested in (statements), and ignore other changes that do not actually change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ [122]. These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes (e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactorings, as long as the node stays within the same file. Using this technique, we can track a node even when it has been moved, for example, because of an extract method refactoring.

For each node (in the AST) involved in a conflict and having a smell, we identify all future commits that touched the file containing said node and tracked the AST node forward in time. For Java, it is possible for multiple statements to be expressed in the same line (e.g., a local variable declaration inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

4.3.6 Commit Classification

In order to answer our third research question, we needed to categorize the type of change for a code commit. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) Other — commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key

problems with this classification are: (1) it is not always trivial to determine which category a commit falls under, and (2) larger projects see a huge amount of activity. Manual classification of all commits was not an option, and we decided to use machine learning techniques for this purpose, rather than limiting the statistical power of our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large bias into our results).

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. The first two authors worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the Other category. Some keywords indicating bug-fixes or improvements were Fix, Bug, Resolves, Cleanup, Optimize, and Simplify, and their derivatives. Anything that did not fit into this pattern was marked as Other.

Not all bug-fixing commits include these keywords or direct reference to the issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. [23], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. This means that we are conservative in identifying commits as bug-fixes.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) by using the SciKit toolset [180]. We used 10% of the data to train the classifier. We applied the classifiers to the training data using a 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the SVM. Therefore, we used the NB classifier to classify our full corpus.

Table 4.4 has the quality indicator characteristics of the NB classifier. Tian et al. [202], suggest that for keyword-based classification the F1 score is usually around 0.55, which also occurs in our case. While our classifier is far from perfect, it is comparable

to “good” classifiers in the literature, and we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, so long as we do not systematically undercount bug-fixes for only some statements, our results should be valid.

Table 4.4: Naive Bayes classifier details

	Precision	**Recall**	**F1-measure**
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of (future) commits in which it appears, as long as those commits are identified as bug-fixes. We stop the tracking when we encounter a commit that is classified as Other. Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

4.3.7 Regression analysis

In order to answer our third research question that is related to the effect of code smells on quality of the resulting code, we needed to build a regression model to identify the impact of code smell on the number of bug-fixes that occur on lines of code that are associated with a code smell and a merge conflict. We use Generalized Linear Regression [41]. The dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

In order to build our model, we collect information about the smells and the conflicts. We use Understand [204] to count the number of references to, and from other files to the files that are involved in a conflict. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files can increase the chance of a change being required in other files, and therefore lead to

multiple developers making changes to these files, which can in turn lead to conflicting changes.

We also collect the following factors for each commit such as the difference between the two merged branches in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger "chunks" of changes should have a higher chance of causing a conflict. We also calculate the number of authors who made commits to the branches that were merged, since there is a higher likelihood of conflicts if multiple developers are involved.

We also determine the experience level of each developer by splitting them into two categories: core and non-core. To calculate the category for each developer, we split the development history into quarters. For each commit a developer is classified as core if he is in the top 20% of the developers in that quarter (calculated by the number of commits). Otherwise he is noncore. We use this process because, in open source projects, authors come and go. Also, an author can be classified as core and non-core in different quarters, depending on his contribution to the project.

After collecting these metrics, we checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model [41]. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

4.4 Results

4.4.1 RQ1: Do program elements that are involved in merge conflicts contain more code smells?

As a first step, we collect the total number of code smells for each of the 6,979 conflicting commits in our dataset. Table 4.5 contains the percentage of each smell and

the percentage of projects that have a particular smell. We find that external and internal duplication have a much higher instance than others when considering the percentage of smells in the dataset. However, about 50% of projects have Data Class and SAP Breakers smells.

Table 4.5: Percentage of code smell

Smell	% of smells in the full dataset	% of projects w/ smell
External Duplication	42.79	22.53
Internal Duplication	**34.05**	**23.80**
Feature Envy	4.04	28.42
Data Clumps	3.71	20.36
Intensive Coupling	3.50	14.30
Data Class	3.18	48.05
Blob Operation	2.58	30.05
Sibling Duplication	2.35	10.86
SAP Breakers	1.52	52.76
God Class	**0.89**	**19.10**
Schizophrenic Class	0.58	20.00
Message Chains	0.33	5.34
Tradition Breaker	0.17	6.33
Refused Parent Bequest	**0.19**	**5.25**
Shotgun Surgery	0.01	1.72
Distorted Hierarchy	**0.003**	**0.36**

We next compare the mean number of code smells associated with each merge commit, for cases when they conflict and for cases when they do not conflict. Note that a commit can involve multiple files, and a file can contain multiple smells. We calculate the total number of smells for each file. For example, a conflicting merge commit in the commandhelper project (with the SHA1 of a91faa) contains one conflicting file, and that conflicting files contains a total of 8 smells.

The mean number of smells in conflicting program elements is 6.54, whereas the mean for non-conflicting program elements is 1.92. The results are statistically significant (Mann-Whitney test, U=6.24e6, $p<4.77e-10$.); we use the non-parametric Mann-Whitney test since our population is not normally distributed. Therefore, we find

that program elements that are involved in merge conflicts are, on average, more smelly than entities that are not involved in a merge conflict.

4.4.2 RQ2: Which code smells are more associated with merge conflicts?

Next, we compare the occurrence of each individual smell across conflicting and non-conflicting commits. Since we are performing multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction, which gives us an adjusted p-value of 0.0031.

For 12 out of 16 total smells, we find significant differences (Mann-Whitney test, $\alpha<0.0031$) between the means of conflicting and non-conflicting commits. The conflicting commits have a higher incidence of smells. Table 4.6 presents the results for code smells where the difference was significant along with the p-values of individual comparisons.

Table 4.6: Mean number of Smells in Conflicts vs. Non-conflict Commits Calculated per Commit

Smell	Smells in conflicts	Smells in non conflict	p-value
God Class	1.23	0.25	0.0001
Data Clump	0.65	0.27	0.0001
Sibling Duplication	0.58	0.10	0.000001
Data Class	0.47	0.12	0.000001
Distorted Hierarchy	0.45	0.05	0.000001
Unnecessary Coupling	0.33	0.10	0.0001
Internal Duplication	0.24	0.08	0.000001
SAP Breaker	0.12	0.07	0.000001
Tradition Breaker	0.10	0.05	0.00007
Blob Operation	0.07	0.06	0.0001
Message Chain	0.04	0.03	0.00062
Shotgun Surgery	0.01	0.00769	0.00021

The following are the top 5 smells in terms of their (mean) numbers per conflict: God Class, Data Clump, Sibling Duplication, Data Class and Distorted Hierarchy. It is worth noting that the distribution of smells per conflict (Table 4.6) is different from Table

4.5. This is because in Table 4.6 we are looking only at the smells that affect the entities involved in merge conflicts, whereas Table 4.5 shows all the smells in the project. This discrepancy is an effect of the fact that merge conflicts exhibit a different smell pattern compared to the overall project.

Next, we perform two steps. First, we investigate the correlation between each smell and the merge conflicts to identify which of the above smells are more strongly associated with conflicts. Then, we categorize merge conflicts into semantic and non-semantic conflicts to further explore the associations of smells to these types of conflicts.

Code smells and conflicts: We perform a correlation analysis between the count of smells and merge conflicts to distill which of the smells from Table 4.6 are more closely associated with conflicts, and should be attended to. We use the Kendall correlation test because it is a non-parametric test and it is more accurate with a smaller sample size. As we perform the tests for each smell, we are splitting out data into smaller chunks. Therefore, the Kendall correlation test is more appropriate.

We find that, except for External Duplication, Schizophrenic Class, SAP Breaker and Data Class all smells are correlated with merge conflicts (Kendall correlation test, $\alpha<0.0031$). We report the statistically significant results in Table 4.7.

The three strongest correlation to conflicts are with the following smells: God Class, Internal Duplication and Distorted Hierarchy. These smells all relate to cases where object-oriented design principles of encapsulation and structuring is not well used, leading to problems with developers making conflicting parallel changes. We discuss these reasons further in Section 5.

Table 4.7: Correlation between conflict and smell count

Smell	Correlation	p-value
God class	0.18	<0.0001
Internal Duplication	0.17	<0.0001
Distorted Hierarchy	0.13	<0.0001
Refused Parent Bequest	0.10	<0.0001
Message Chain	0.10	<0.0001
Data clump	0.09	<0.0001
Feature Envy	0.09	<0.0001
Tradition Breaker	0.09	<0.0001
Blob Operation	0.08	<0.0001
Shotgun Surgery	0.07	<0.0001
Unnecessary Coupling	0.05	0.00007
Sibling Duplication	0.04	0.00021

Types of conflicts and their classification: Not all conflicts are the same, some involve changes to the actual code structure and require the developer to understand the logic behind the changes before they can be integrated (semantic conflicts), whereas others can be formatting or cosmetic changes (non-semantic). Semantic conflicts are inherently harder to resolve. Therefore, we investigate whether specific types of code smells are more likely to occur with semantic conflicts. We use the conflict classification methodology in Section III-D.

Recall, we manually labeled 606 conflicts to classify them into semantic or non-semantic, which we then use for the automated classification of 6,979 commits. We present the distribution between the manual and automatic classification in Table 4.8. The distributions of semantic and non-semantic conflicts in the automatically classified data match the distribution of our manual labeling (training data), which shows the efficacy of the automated classifier.

Table 4.8: Conflict types based on their frequency of occurrence

Category	# of Conflicts	% of total (classifier)	% of total (training)
Semantic	5,250	75.23%	76.12%
Non-Semantic	1,729	24.77%	23.88%

Semantic conflicts are more common (76.12% in the manually labeled data and 75.23% in the automated classified data), as compared to the non-semantic conflicts (23.88% in manually labeled and 24.77% in automated classified data).

Semantic conflicts and code smells: To understand if there is any correlation between semantic conflicts and the types of code smells we perform the Kendall correlation test for each smell in the presence of semantic merge conflicts (in our total dataset). We use the Kendall correlation test and found significant correlation ($\alpha<0.0031$) only for Internal Duplication and Blob Operation. Table 4.9 contains all correlations, where the cells marked with ** are significant.

Since the correlation for both Blob Operation and Internal Duplication are small, we perform an odds-ratio test to understand which of these smells are more likely to be involved in a Semantic merge conflict, as compared to entities that do not have these smells, but were involved in a conflict. Since we are performing two comparisons, we have to adjust the significance value to adjust for multiple hypothesis testing. Like in the previous sections, we performed a Bonferroni correction, which gives us significance value of $\alpha=0.0025$ to test at.

We performed an odds ratio test (Fisher's exact test) for the Blob Operations and find that they are 1.7 times more likely to be involved in a Semantic merge conflict (odds ratio: 1.77, p=0.0024). Blob Operations are methods that are very complex and have many responsibilities. Therefore, any change to the method will likely impact multiple lines, which may intersect with logical changes made by another developer to the same method. This explains the high likelihood of their involvement in Non-Semantic conflicts.

For Internal Duplication, we found that they are 1.55 times more likely to be involved in merge conflict (odds ratio: 1.55, p=0.0001.) We attribute this to the fact that, because of duplication, a change has to be repeated in multiple locations. This increases the chances of developers making overlapping changes.

Table 4.9: Smell Categories for Semantic Conflicts (significance level α=0.0031)

Smell	**Correlation**	**p-value**
Blob Operation **	0.05	0.0030
Internal Duplication **	0.07	0.0002
Message Chain	0.01	0.4970
Refused Parent Bequest	0.03	0.0492
SAP Breaker	-0.02	0.2652
Schizophrenic Class	-0.03	0.0832
Shotgun Surgery	-0.008	0.6597
Sibling Duplication	0.031	0.1029
Tradition Breaker	0.018	0.3291
Unnecessary Coupling	-0.01	0.5681
Data Class	-0.02	0.1524
Data Clumps	0.030	0.1103
Distorted Hierarchy	0.034	0.0670
Feature Envy	0.009	0.6206
God Class	0.050	0.0072

4.4.3 RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

We aim to model the effects of code smells on the bugginess of a line of code involved in a merge conflict. As defect prediction literature has already identified several factors (e.g., the size of the module under investigation [61], number of committers [207], centrality of files [37]) that affect bugginess, we include them in our model also. Table 4.10 lists the final set of factors that we use, which include metrics that are code-based (F1, F2), change-related (F5-F8), author-related (F3, F4), and code-smells. We compute whether a developer is core or non-core based on our methodology in Section III-G.

To answer our third research question, we build two Generalized Linear Models (GLM). The first contains the number of code smells as a factor, and the second does not. The first (Poisson regression) model is built with a log linking function as explained in Section III-G. After filtering the factors with VIF $\leq$ 5, we had a set of 8 factors out of 43 factors. All eight factors were statistically significant (see Table 4.10).

The predicted value is the total number of bug fixes occurring on a line of code that was involved in a merge conflict. Note that smell count was a significant factor in the model ($p<0.05$), with an estimate of 0.427.

The McFadden Adjusted R^2 [95] of this model is 0.47. We calculated McFadden's Adjusted R^2 as a quality indicator of the model because there is no direct equivalent of R^2 metric for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, adjusted R^2 values like McFadden's cannot be interpreted as one would interpret OLS R^2 values. McFadden's Adjusted R^2 values tend to be considerably lower than those of the R^2. Values of 0.2 to 0.4 represent an excellent fit [95]

Table 4.10: Poisson regression model predicting bug-fix occurrence on Lines of Code involved in a merge conflict

Factor#	**Factor**	**Estimate**	**p-value**
F1	In Deps	3.195	<0.0001
F2	Out Deps	-0.053	<0.0001
F3	Noncore author	-3.799	<0.0001
F4	No. Authors	0.129	<0.0001
F5	No. Classes	-0.373	<0.0001
F6	No. Methods	0.244	<0.0001
F7	AST diff	0.001	<0.0001
F8	LOC diff	0.00002571	<0.0001
F9	Number of Smells	0.427	<0.0001

To understand the impact that code smells have, we built the same model by removing the total number of smells as a factor. This decreased the adjusted-R^2 from 0.47 to 0.44. We can therefore conclude that code smells have a significant impact on the final quality of the code. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, had the code smells not mattered, removing it could have increased the adjusted-R^2 instead of reducing it.

4.5 Discussion

To the best of our knowledge, we are the first to investigate the association of code smells with that of merge conflicts, and their impact on the bugginess of the merged results (line of code). We find that program elements that are involved in merge conflicts contain, on average, 3 times more code smells than program elements that are not involved in a merge conflict.

Not all code smells are equally correlated to merge conflicts. 12 out of the 16 code smells that co-occur with conflicts are significant associated with merge conflicts. The top five code smells from this list are: God class, Message Chain, Internal Duplication, Distorted Hierarchy and Refused Parent Bequest. Interestingly, the only (significant) code smells associated with Semantic conflicts are Blob Operation and Internal Duplication.

All the above code smells arise when developers do not fully exploit the advantages of object-oriented design, leading to high coupling, duplication, or large containers. These factors lay the groundwork for parallel conflicting efforts, where developers step on each other's toes. For example, the Blob Operation is a large and complex method that grows over time becoming hard to maintain. In such a situation, multiple developers may need to make changes to the same method and, therefore, collide when merging. Similarly, Internal Duplication arises when code is duplicated, which bloats methods and makes it hard to ensure all clones evolve in the same way. In such a situation, developers might have to "touch" multiple parts of the method to ensure all clones are being updated, causing situations of parallel, conflicting edits.

It is interesting to observe that Semantic merge conflicts are associated with smells at the method level. For example, the Blob Operation and Internal Duplication smells are 1.77 times and 1.55 times, respectively, more likely to be present in a semantic conflict as compared to a non-semantic conflict. This indicates that bloated methods or duplicated code in methods increase the spread of the change a developer is likely to make, which in turn increases the likelihood of two or more changes conflicting during

a merge. Prior work has associated code duplication with negative consequences such as increased maintenance cost [127, 184] and faults [19, 107]. Our findings indicate that duplication also negatively impacts the collaborative workflow by making it difficult to merge changes.

It is worth noting that while smells, such as God Class have a significant correlation with overall merge conflicts, they do not have a significant correlation with semantic merge conflicts. We posit that a large container (class) with cohesive logical units (methods) can lead to multiple developers making parallel changes that are localized to specific areas (methods) and do not intersect. In these cases, when changes are merged conflicts can arise because of the movement of code or formatting changes (non-semantic conflicts). The same reasoning is also applicable for Distorted Hierarchy, Refused Parent Bequest and Message Chain. In contrast, as discussed earlier method-level smells seem are correlated with semantic conflicts.

To the best of our knowledge, ours is the first empirical study to investigate the effects of merge conflicts and code smells on the bugginess of code. We found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess (see Table 4.10). Including code smells as a factor increases the McFadden's adjusted R^2 value from 0.44 to 0.47. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, an increase in the value signifies that adding code smells as a factor was valuable. We find that factors such as incoming-dependencies and the number of code smells have the highest correlation estimate, indicating their importance to the model.

We find that some factors, such as non-core author, number of classes, and outward dependencies have a negative effect on bugginess. This is counter intuitive. We had assumed that changes from multiple non-core authors are more likely to be buggy. We believe that the following reasons lead to this surprising outcome. It might be the case that non-core contributors are more thorough and put more effort towards submitting code that is less bug prone. Or it might be the process via which newcomers'

contributions are accepted. For example, core developers might pay more attention to changes coming from non-core contributors. Further empirical studies on the differences in review processes for core vs. non-core developers will be interesting. We also found that the number of classes involved in a conflict has a negative correlation to its bugginess. This might be because changes that involve multiple classes are more likely to be refactoring or licensing changes, and therefore, less likely to introduce bugs.

Implications: Our findings have a number of implications for software practitioners, tool builders and researchers.

Code smells have been historically associated with maintenance issues, which are known to be a problem in the long term. However, developers are often unaware of code smells. Yamashita et al. found that a considerable portion (32%) of developers did not know about code smells [211]. Our findings shed a different light on the impact of code smells and on the importance of addressing them. Our results show that code smells are an immediate concern for day-to-day activity such as merging changes.

Merge conflicts delay the project by requiring an examination of the conflict, and disrupting the developers' workflow. Anecdotal evidence shows that developers hate resolving conflicts. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes etc.) or rush to commit their work in an effort to avoid having to resolve conflicts [49]. A developer may also choose to delay the incorporation of others' work, fearing that a conflict may be hard to resolve [49]. Such processes can have a detrimental effect on team productivity and morale. This situation can only become worse as the project evolves on two fronts. First, the number of code smells is likely to increase as the project ages [3]. Second, there is a likelihood of increase in merge conflicts as more developers start to contribute. Our results indicate that practitioners should pay more attention to code smells, as it will not only make the code quality better, but will also help them minimize the number of merge conflicts they need to resolve.

Practitioners, when investigating the root cause of a merge conflict can start by looking for smelly program elements in the code. Moreover, since changes that involve entities containing code smells are more likely to lead to semantic merge conflicts, integrators (or code reviewers) should pay particular attention to and attempt to remove code smells when reviewing commits. Practitioners should also pay attention to "good" software engineering processes when they deal with smelly program elements. For example, when changes are being made to smelly parts of the code base developers should merge more frequently and perform more thorough code reviews.

Our results show that code smells are a good predictor of merge conflict and the level of difficulty of that conflict. Therefore, tool builders can use the information of incidence of code smells to support distributed work – either in predicting likelihood of conflicts or their difficulty. Code smells can also be used as a factor to schedule tasks (e.g., program elements that have code smells should not be edited in parallel) or assign tasks (e.g., developers with higher experience should work on smelly program elements).

Our results have implications for researchers. Since code smells together with merge conflicts can predict bugginess, researchers can use this information in bug prediction models to increase their effectiveness. To the best of our knowledge, no merge conflict prediction tool exists. Our results show that code smells have a strong association with merge conflicts, therefore, researchers can use this information to predict impending merge conflicts. Our results also have implications in testing. For example, increasing the test coverage of smelly lines that were involved in a merge conflict can be used as an objective/fitness function in the field of search-based software engineering.

4.6 Threats to validity

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Bias due to sampling: Our samples have been from a single source - Github. This may be a source of bias, and our findings may be limited to open source programs from Github and not generalizable to commercial programs. However, the threat is minimal since we analyze a large number of projects spanning eight different domains.

Bias due to tools used: The smell detection tool we used uses static code analysis to identify smells and research shows that code smells that are "intrinsically historical" such as Divergent Change, Shotgun Surgery and Parallel Inheritance are difficult to detect by just exploiting static source code analysis [175]. So the number occurrence of such "intrinsically historical" smells should be different when historical information based smell detection technique is used.

Secondly, we used the Gumtree algorithm [63] for tracking program elements across commits. However, the algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Bias Due to using classifiers: We use machine learning to group conflicts into the two categories, and to determine whether a commit was a bug-fix. As with any classifier, we have some mislabeling. While our results do not require those results to be anywhere near perfect, this threat is low as our classifiers have good F1-measure and high precision.

Regarding the bug-fix classifier, our recall and precision measures are on par with past work [23]. Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

4.7 Conclusions

In this paper, we study the history of 143 open source projects, from which we extract 6,979 merge conflicts to see if there is any correlation between code smells and merge conflicts. We found that entities involved in merge conflicts contain almost 3 times more code smells than non-conflicting entities.

To have a better understanding of the effect of code smells on merge conflicts, we categorized conflicts into semantic conflicts – changes to the AST and hard to resolve – and non-semantic – changes that are cosmetic. We found two method-level code smells (Blob Operation and Internal Duplication) to be significantly correlated with semantic conflicts. More specifically, methods that contained the Blob Operation and Internal Duplication smells were more likely to be involved in a semantic merge conflict, by 1.77 times and 1.55 times respectively. We also found that code smells have a significant impact on the final quality of the code. Count of code smells was a significant factor when we modeled the bugginess of lines of code involved in a merge conflict.

Our results show that code smells, thought to be a maintenance issue and often neglected by practitioners, have an immediate impact in how distributed development is managed. Their presence is not only associated with difficult merge conflicts (semantic), but also with the likelihood of bugs getting introduced in the code base.

Chapter 5 CONCLUSION AND FUTURE WORK

This dissertation focused on how testing and fault prediction can be used to improve the quality of software by uncovering software errors in large, complex, real world systems.

In the first part of this thesis, we investigated the notions of testedness and illustrated how that is associated with widely-used measures of test suite quality measured using the actual criteria of interest "future bug-fixes". We show that both statement coverage and mutation score have only a weak negative correlation with future bug-fixes, mutation score having slightly stronger correlation between the two. In the second part, we investigate the applicability of mutation analysis in a real world complex software system (RCU). Here we adapt existing techniques to constrain the complexity and computation requirements associated with mutation analysis and show that mutation analysis can be a useful tool, uncovering gaps in even well-tested module like RCU. We also show that some of the problems identified by researchers are not that significant when you try to apply mutation analysis on real world software. In the third part, we investigate fault prediction models and showed that using a combination of socio-technical factors, merge conflict and code smells, we can build fault prediction models that can help to achieve significant improvement in fault prediction.

Although we have barely scratched the surface of using mutation analysis on real world complex software for bug-finding, we believe that the results presented in this dissertation speak clearly and significantly regarding the potential of using mutation analysis for this purpose using existing techniques and minimal human intervention. In tandem with creating new and more efficient testing techniques, we need to come up with better ways to integrate existing techniques like mutation testing into the actual development workflow of developers. One such way is to help developers in manual inspection of surviving mutants. As manual inspection step of mutation analysis is an expensive part in terms of required human time when the number of surviving mutant is high. One of the future research direction is to find ways of prioritizing the surviving

mutants so that developers can first inspect important mutants within their budgeted time.

On the fault prediction side, despite of decades of research on predicting failures usually concentrating either on the technical, or the social side of software development, researchers haven't focused much on using combination of socio-technical factors. One of the future research direction is to analyze the complex interactions between socio-technical factors for identifying new features which can be used in fault prediction models to improve the fault prediction accuracy.

Luck favors the prepared mind

Louis Pasteur

BIBLIOGRAPHY

[1] Acree Jr, & Troy, A. (1980). "On Mutation (No. GIT-ICS-80/12)". In PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia.

[2] Agrawal, H., DeMillo, R., Hathaway, R., Hsu, W., Hsu, W., Krauser, E., ... & Spafford, E. (1989). "Design of mutant operators for the C programming language.". In Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana.

[3] Ahmed, I., Mannan, U. A., Gopinath, R., & Jensen, C. "An empirical study of design degradation: How software projects get worse over time". In Empirical Software Engineering & Measurement (ESEM), 2015, pp.1-10.

[4] Alglave, J., Maranget, L., & Tautschnig, M. (2014). "Herding cats: Modelling, simulation, testing, and data mining for weak memory". In ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 36, No. 2, (pp. 7).

[5] American Fuzzy Lop (AFL): http://lcamtuf.coredump.cx/afl/

[6] Ammann, P., Delamaro, M. E., & Offutt, J. (2014,). "Establishing theoretical minimal sets of mutants". In Seventh International Conference on Software Testing, Verification and Validation, (pp. 21-30). IEEE.

[7] Amrit, C., Hillegersberg, J., & Kumar, K. (2004). "A social network perspective of conway's law". In Proceedings of the CSCW Workshop on Social Networks, Chicago, IL, USA.

[8] Andrews, J. H., & Zhang, Y. (2003). "General test result checking with log file analysis". In Transactions on Software Engineering, Vol. 29, No.7, (pp. 634-648). IEEE

[9] Andrews, J. H., Briand, L. C., & Labiche, Y. (2005). "Is mutation an appropriate tool for testing experiments?". In Proceedings of the 27th international conference on Software engineering (pp. 402-411). ACM.

[10] Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). "Using mutation analysis for assessing and comparing testing coverage criteria". In IEEE Transactions on Software Engineering, 32(8), 608-624.

[11] Apache Software Foundation. Apache commons. http://commons.apache.org/.

[12] Apache Software Foundation. Apache maven project. http://maven.apache.org

[13] Apel, S., Leßenich, O., & Lengauer, C. (2012). “Structured merge with auto-tuning: balancing precision and performance”. In International Conference on Automated Software Engineering, 2012, (pp. 120-129).

[14] Apel, S., Liebig, J., Brandl, B., Lengauer, C., & Kästner, C. (2011). “Semistructured merge: rethinking merge in revision control systems”. In 13th European conference on Foundations of software engineering, 2011, (pp. 190-200).

[15] Arcuri, A., & Briand, L. (2014). “A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering”. In Software Testing, Verification and Reliability, 24(3), 219-250.

[16] Arisholm, E., & Briand, L. C. (2006). “Predicting fault-prone components in a java legacy system”. In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (pp. 8-17). ACM.

[17] Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., & Vos, T. (2011). “Symbolic search-based testing”. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (pp. 53-62). IEEE Computer Society.

[18] Baker, R. J., & Habli, I. (2013). “An empirical evaluation of mutation testing for improving the test quality of safety-critical software”. In Transactions on Software Engineering, , Vol. 39, No.6, (pp. 787-805). IEEE.

[19] Bakota, T., Ferenc, R., & Gyimothy, T. (2007). “Clone smells in software evolution”.In IEEE International Conference on Software Maintenance, 2007, (pp. 24-33).

[20] Baldwin, D., & Sayward, F. (1979). “Heuristics for Determining Equivalence of Program Mutations”. In PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia..

[21] Basili, V. R., Briand, L. C., & Melo, W. L. (1996). “A validation of object-oriented design metrics as quality indicators”. In IEEE Transactions on software engineering, 22(10), 751-761.

[22] Biehl, J. T., Czerwinski, M., Smith, G., & Robertson, G. G. (2007). “FASTDash: a visual dashboard for fostering awareness in software teams”. In Human factors in computing systems, 2007, (pp. 1313-1322).

[23] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009). “Fair and balanced?: bias in bug-fix datasets”. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 121-130). ACM.

[24] Bird, C., Gourley, A., Devanbu, P., Gertz, M., & Swaminathan, A. (2006). “Mining email social networks”. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 137-143). ACM.

[25] Bird, C., Nagappan, N., Gall, H., Murphy, B., & Devanbu, P. (2009). “Putting it all together: Using socio-technical networks to predict failures”. In Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on (pp. 109-119). IEEE.

[26] Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). “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 (pp. 4-14). ACM.

[27] Boehm, B. W., Brown, J. R., & Lipow, M. (1976). “Quantitative evaluation of software quality”. In International conference on Software engineering, 1976 (pp. 592-605).

[28] Bradbury, J. S., Cordy, J. R., & Dingel, J. (2006). “Mutation operators for concurrent Java (J2SE 5.0)”. In Mutation Analysis, 2006. Second Workshop on (pp. 11-11). IEEE.

[29] Briand, L. C., Thomas, W. M., & Hetmanski, C. J. (1993). “Modeling and managing risk early in software development”. In Proceedings of the 15th international conference on Software Engineering, (pp. 55-65). IEEE Computer Society Press.

[30] Brun, Y., & Ernst, M. D. (2004). “Finding latent code errors via machine learning over program executions”. In Proceedings of the 26th International Conference on Software Engineering (pp. 480-490). IEEE Computer Society.

[31] Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2011). “Proactive detection of collaboration conflicts”. In European conference on Foundations of software engineering, 2011, (pp. 168-178).

[32] Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). "Towards a taxonomy of software change". In Journal of Software: Evolution and Process, 17(5), 309-332.

[33] Budd, T. A., & Angluin, D. (1982). "Two notions of correctness and their relation to testing". In Acta Informatica, Vol. 18, No.1, (pp. 31-45).

[34] Budd.T. A. (1980). "Mutation Analysis of Program Test Data". PhD thesis, Yale University, New Haven, CT, USA, 1980.

[35] Cai, X., & Lyu, M. R. (2005). "The effect of code coverage on fault detection under different testing profiles". In ACM SIGSOFT Software Engineering Notes, 30(4), 1-7.

[36] Canfora, G., Cerulo, L., & Di Penta, M. (2007). "Identifying Changed Source Code Lines from Version Repositories". In MSR (Vol. 7, p. 14).

[37] Cataldo, M., & Herbsleb, J. D. (2013). "Coordination breakdowns and their impact on development productivity and software failures". In IEEE Transactions on Software Engineering, 39(3), 343-360.

[38] Cataldo, M., Herbsleb, J. D., & Carley, K. M. (2008). "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 (pp. 2-11). ACM.

[39] Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., & Carley, K. M. (2006). "Identification of coordination requirements: implications for the Design of collaboration and awareness tools". In Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work (pp. 353-362). ACM.

[40] Chess, B., & West, J. (2007). "Secure programming with static analysis". In Pearson Education.

[41] Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2013). "Applied multiple regression/correlation analysis for the behavioral sciences". Routledge.

[42] Coles.H. Pit mutation testing. http://pitest.org/.

[43] Companion Website: https://goo.gl/ORpLkU

[44] Costa, C., Figueiredo, J. J., Ghiotto, G., & Murta, L. (2014). “Characterizing the Problem of Developers' Assignment for Merging Branches”. In International Journal of Software Engineering and Knowledge Engineering, 24(10), 1489-1508.

[45] Cunningham, W. (1993). “The WyCash portfolio management system”. In ACM SIGPLAN OOPS Messenger, 4(2), 29-30.

[46] D’Ambros, M., Lanza, M., & Robbes, R. (2012). “Evaluating defect prediction approaches: a benchmark and an extensive comparison”. In Empirical Software Engineering, 17(4-5), 531-577.

[47] Da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M., & Van Der Hoek, A. (2006). “Lighthouse: coordination through emerging design”. In Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange (pp. 11-15). ACM.

[48] Daran, M., & Thévenod-Fosse, P. (1996). “Software error analysis: A real case study involving real faults and mutations”. In ACM SIGSOFT Software Engineering Notes (Vol. 21, No. 3, pp. 158-171). ACM.

[49] De Souza, C. R., Redmiles, D., & Dourish, P. (2003). “Breaking the code, moving between private and public work in collaborative software development”. In Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work (pp. 105-114). ACM.

[50] De Souza, L. B. L., & de Almeida Maia, M. (2013). “Do software categories impact coupling metrics?”. In Mining Software Repositories (MSR), 2013, (pp. 217-220).

[51] Debroy, V., & Wong, W. E. (2009). “Insights on fault interference for programs with multiple bugs”. In Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on (pp. 165-174). IEEE.

[52] Delahaye, M., & Bousquet, L. (2015). “Selecting a software engineering tool: lessons learnt from mutation analysis”. In Software: Practice and Experience, 45(7), 875-891.

[53] Deligiannis, I., Shepperd, M., Roumeliotis, M., & Stamelos, I. (2003). “An empirical investigation of an object-oriented design heuristic for maintainability”. Journal of Systems and Software, 65(2), 127-139.

[54] Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., & Shepperd, M. (2004). "A controlled experiment investigation of an object-oriented design heuristic for maintainability". In Journal of Systems and Software, 72(2), 129-143.

[55] DeMillo, R. A., & Mathur, A. P. (1991). "On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software". In Thirteenth Minnowbrook Workshop on Software Engineering.

[56] DeMillo, R. A., Mathur, A. P., Wong, W. E., Frankl, P. G., & Weyuker, E. J. (1995). "Some critical remarks on a hierarchy of fault-detecting abilities of test methods [and reply]". In IEEE Transactions on Software Engineering, 21(10), 858-863.

[57] Denaro, G., & Pezzè, M. (2002). "An empirical evaluation of fault-proneness models". In Proceedings of the 24th International Conference on Software Engineering (pp. 241-251). ACM.

[58] Dewan, P., & Hegde, R. (2007). "Semi-synchronous conflict detection and resolution in asynchronous software development". In ECSCW 2007, 159-178.

[59] DiGiuseppe, N., & Jones, J. A. (2011). "Fault interaction and its repercussions". In 27th IEEE International Conference on Software Maintenance (pp. 3-12) (ICSM), 2011. IEEE.

[60] Dourish, P., & Bellotti, V. (1992). "Awareness and coordination in shared workspaces". In ACM conference on Computer-supported cooperative work (pp. 107-114). ACM.

[61] El Emam, K., Benlarbi, S., Goel, N., & Rai, S. N. (2001). "The confounding effect of class size on the validity of object-oriented metrics". In IEEE Transactions on Software Engineering, 27(7), 630-650.

[62] ESXI: http://searchvmware.techtarget.com/definition/VMware-ESXi

[63] Falleri, J. R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014). "Fine-grained and accurate source code differencing". In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (pp. 313-324). ACM.

[64] Feitelson, D. G. (2012). "Perpetual development: a model of the Linux kernel life cycle". In Journal of Systems and Software, 85(4), 859-875.

[65] Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). “Comparing and experimenting machine learning techniques for code smell detection”. In Empirical Software Engineering, 21(3), 1143-1191.

[66] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011). “An experience report on using code smells detection tools”. In IEEE Fourth International Conference on Software Testing, Verification and Validation Workshop (ICSTW), pp. 450-457.

[67] Fowler, M., & Beck, K. (1999). “Refactoring: improving the design of existing code”. Addison-Wesley Professional.

[68] Frankl, P. G., & Iakounenko, O. (1998). “Further empirical studies of test effectiveness”. In ACM SIGSOFT Software Engineering Notes, 23(6), 153-162.

[69] Frankl, P. G., & Weiss, S. N. (1993). “An experimental comparison of the effectiveness of branch testing and data flow testing”. In IEEE Transactions on Software Engineering, 19(8), 774-787.

[70] Frankl, P. G., Weiss, S. N., & Hu, C. (1997). “All-uses vs mutation testing: an experimental comparison of effectiveness”. In Journal of Systems and Software, 38(3), 235-253.

[71] GitHub Inc. Software repository. http://www.github.com.

[72] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., & Marinov, D. (2013). “Comparing non-adequate test suites using coverage criteria”. In Proceedings of the 2013 International Symposium on Software Testing and Analysis (pp. 302-313). ACM.

[73] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., & Marinov, D. (2015). “Guidelines for coverage-based comparisons of non-adequate test suites”. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(4), 22.

[74] Gligoric, M., Zhang, L., Pereira, C., & Pokam, G. (2013, July). “Selective mutation testing for concurrent code”. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, (pp. 224-234). ACM.

[75] Godfrey, M. W., & Zou, L. (2005). “Using origin analysis to detect merging and splitting of source code entities”. In IEEE Transactions on Software Engineering, 31(2), 166-181.

[76] Gopinath, R., Jensen, C., & Groce, A. (2014). "Code coverage for suite evaluation by developers". In Proceedings of the 36th International Conference on Software Engineering (pp. 72-82). ACM.

[77] Gopinath, R., Jensen, C., & Groce, A. (2014). "Mutations: How close are they to real faults?". In Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium, (pp. 189-200). IEEE.

[78] Gopinath, R., MA Alipour, Ahmed, I., Jensen, C., & Groce, A. (2016). "On The Limits of Mutation Reduction Strategies". In Proceedings of the 38th International Conference on Software Engineering, (pp-511-522). ACM.

[79] Gorton, I., & Liu, A. (2002). "Software component quality assessment in practice: successes and practical impediments". In Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on (pp. 555-558). IEEE.

[80] Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). "Predicting fault incidence using software change history". In IEEE Transactions on software engineering, 26(7), 653-661.

[81] Groce, A., Ahmed, I., Jensen, C., & McKenney, P. E. (2015). "How Verified is My Code? Falsification-Driven Verification." In Proceedings of the 30th ACM/IEEE international conference on Automated software engineering (ASE).

[82] Groce, A., Alipour, M. A., & Gopinath, R. (2014). "Coverage and its discontents". In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (pp. 255-268). ACM.

[83] Guimarães, M. L., & Silva, A. R. (2012). " Improving early detection of software merge conflicts". In 34th International Conference on Software Engineering (ICSE), (pp. 342-352).

[84] Guniguntala, D., McKenney, P. E., Triplett, J., & Walpole, J. (2008). "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux".In IBM Systems Journal, Vol. 47, No. 2, (pp. 221-236).

[85] Gupta, A., & Jalote, P. (2008). "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing". In International Journal on Software Tools for Technology Transfer, 10(2), 145-160.

[86] Gyimothy, T., Ferenc, R., & Siket, I. (2005). “Empirical validation of object-oriented metrics on open source software for fault prediction”. In Software Engineering, IEEE Transactions on, 31(10), 897-910.

[87] Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). “A systematic literature review on fault prediction performance in software engineering”. In Software Engineering, IEEE Transactions on, 38(6), 1276-1304.

[88] Hall, T., Zhang, M., Bowes, D., & Sun, Y. (2014). “Some code smells have a significant but small effect on faults”. In ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4), 33.

[89] Harman, M., Hierons, R., & Danicic, S. (2001). “The relationship between program dependence and mutation analysis”. In Mutation testing for the new century, (pp. 5-13). Springer US.

[90] Harman, M., Jia, Y., Reales Mateo, P., & Polo, M. (2014). “Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation”. In Proceedings of the 29th international conference on Automated software engineering, (pp. 397-408). ACM.

[91] Hassan, A. E. (2009). “Predicting faults using the complexity of code changes”. In Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on (pp. 78-88). IEEE.

[92] Hattori, L., & Lanza, M. (2010). “Syde: A tool for collaborative software development”. In 32nd International Conference on Software Engineering-Volume 2 (pp. 235-238).

[93] Heckman, S., & Williams, L. (2008). “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 (pp. 41-50). ACM.

[94] Heckman, S., & Williams, L. (2011). “A systematic literature review of actionable alert identification techniques for automated static code analysis”. In Information and Software Technology, 53(4), 363-387.

[95] Hensher, D. A., & Stopher, P. R. (Eds.). (1979). “Behavioural travel modelling”. London: Croom Helm.

[96] Hovemeyer, D., & Pugh, W. (2004). “Finding bugs is easy”. In ACM Sigplan Notices, 39(12), 92-106.

[97] Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). “Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria”. In Proceedings of the 16th international conference on Software engineering (pp. 191-200). IEEE Computer Society Press.

[98] InFusion, http://www.intooitus.com/inFusion.html. (accessed at January 2014)

[99] Inozemtseva, L. M. M. (2012). “Predicting test suite effectiveness for java programs”. (Master's thesis, University of Waterloo).

[100] Inozemtseva, L., & Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering (pp. 435-445). ACM.

[101] Izurieta, C., & Bieman, J. M. (2007). How software designs decay: A pilot study of pattern evolution. Empirical Software Engineering and Measurement. (pp. 449-451).

[102] Jia, Y., & Harman, M. (2008). “MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language”. In Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference (pp. 94-98). IEEE.

[103] Jia, Y., & Harman, M. (2008). “Constructing subtle faults using higher order mutation testing”. In 8th International Working Conference on Source Code Analysis and Manipulation, (pp. 249-258). IEEE.

[104] Jia, Y., & Harman, M. (2009). “Higher order mutation testing”. In Information and Software Technology, 51(10), 1379-1393.

[105] Jia, Y., & Harman, M. (2011). “An analysis and survey of the development of mutation testing”. In Software Engineering Transactions, Vol. 37, No. 5, (pp. 649-678). IEEE.

[106] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R.(2013). “Why don't software developers use static analysis tools to find bugs?”. In Proceedings of the 2013 International Conference on Software Engineering (pp. 672-681). IEEE Press.

[107] Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). “Do code clones matter?”. In Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on (pp. 485-495). IEEE.

[108] JUnit Testing Framework. http://www.junit.org, 2014.

[109] Just, R., Ernst, M. D., & Fraser, G. (2014). “Efficient mutation analysis by propagating and partitioning infected execution states”. In Proceedings of the International Symposium on Software Testing and Analysis, (pp. 315-326). ACM.

[110] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., & Fraser, G. (2014). “Are mutants a valid substitute for real faults in software testing?”. In Proceedings of the 22nd International Symposium on Foundations of Software Engineering, (pp. 654-665). ACM.

[111] Just, R., Kapfhammer, G. M., & Schweiggert, F. (2012). "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis". In Software Reliability Engineering, (pp. 11-20). IEEE.

[112] Kagdi, H., Gethers, M., Poshyvanyk, D., & Collard, M. L. (2010). “Blending conceptual and evolutionary couplings to support change impact analysis in source code”. In Reverse Engineering (WCRE), 2010 17th Working Conference on (pp. 119-128). IEEE.

[113] Kakarla, S. (2010). “An analysis of parameters influencing test suite effectiveness”. In Doctoral dissertation.

[114] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K. I., Adams, B., & Hassan, A. E. (2010). “Revisiting common bug prediction findings using effort-aware models”. In Software Maintenance (ICSM), 2010 IEEE International Conference on (pp. 1-10). IEEE.

[115] Kaminski, G., Ammann, P., & Offutt, J. (2011). “Better predicate testing”. In Proceedings of the 6th International Workshop on Automation of Software Test, (pp. 57-63). ACM

[116] Kaminski, G., Ammann, P., & Offutt, J. (2013). “Improving logic-based testing”. In Journal of Systems and Software, Vol. 86, No. 8, (pp. 2002-2012).

[117] Kasi, B. K., & Sarma, A. (2013). “Cassandra: Proactive conflict minimization through optimized task scheduling”. In International Conference on Software Engineering (pp. 732-741). IEEE Press.

[118] Kawrykow, D., & Robillard, M. P. (2011). “Non-essential changes in version histories”. In Proceedings of the 33rd International Conference on Software Engineering (pp. 351-360). ACM.

[119] Khomh, F., Di Penta, M., Guéhéneuc, Y. G., & Antoniol, G. (2012). “An exploratory study of the impact of antipatterns on class change-and fault-proneness”. In Empirical Software Engineering, 17(3), 243-275.

[120] Kim, S., & Ernst, M. D. (2007). “Which warnings should I fix first?”. In Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 45-54). ACM

[121] Kim, S., Whitehead, E. J., & Zhang, Y. (2008). “Classifying software changes: Clean or buggy?”. Software Engineering, IEEE Transactions on, 34(2), 181-196.

[122] Kim, S., Zimmermann, T., Pan, K., & James Jr, E. (2006). “Automatic identification of bug-introducing changes”. In International Conference on Automated Software Engineering, ASE'06. (pp. 81-90).

[123] King, J. C. (1976). “Symbolic execution and program testing”. In Communications of the ACM, Vol.19, No. 7, (pp. 385-394).

[124] Kintis, M., Papadakis, M., & Malevris, N. (2010). “Evaluating mutation testing alternatives: A collateral experiment”. In Software Engineering Conference (APSEC), 2010 17th Asia Pacific (pp. 300-309). IEEE.

[125] Knab, P., Pinzger, M., & Bernstein, A. (2006). “Predicting defect densities in source code files with decision tree learners”. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 119-125). ACM.

[126] Kokologiannakis, M., & Sagonas, K. (2017). “Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU)”. In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (pp. 172-181). ACM.

[127] Koschke, R. (2007). “Survey of research on software clones”. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[128] Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). “Technical debt: From metaphor to theory and practice”. IEEE software, 29(6), 18-21.

[129] Langdon, W. B., Harman, M., & Jia, Y. (2010). "Efficient multi-objective higher order mutation testing with genetic programming". In Journal of systems and Software, Vol. 83, No. 12, (pp. 2416-2430).

[130] Lanza, M., & Marinescu, R. (2007). "Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems". In Springer Science & Business Media.

[131] Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). "Benchmarking classification models for software defect prediction: A proposed framework and novel findings". In IEEE Transactions on Software Engineering, 34(4), 485-496.

[132] Li, N., Praphamontripong, U., & Offutt, J. (2009). "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage". In Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on (pp. 220-229). IEEE.

[133] Li, W., & Shatnawi, R. (2007). "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". In Journal of systems and software, 80(7), 1120-1128.

[134] Liang, G., Wu, L., Wu, Q., Wang, Q., Xie, T., & Mei, H. (2010). "Automatic construction of an effective training set for prioritizing static analysis warnings". In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 93-102). ACM.

[135] Liang, L., McKenney, P. E., Kroening, D., & Melham, T. (2016). "Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel". In arXiv preprint arXiv:1610.03052.

[136] Lippe, E., & Van Oosterom, N. (1992). "Operation-based merging". In ACM SIGSOFT Software Engineering Notes (Vol. 17, No. 5, pp. 78-87). ACM.

[137] Lipton, R. (1971). "Fault diagnosis of computer programs". Student Report, Carnegie Mellon University.

[138] Lissy, A., Laurière, S., & Martineau, P. (2011). "Verifications around the Linux kernel". In Linux Symposium, (p. 37).

[139] Ma, Y. S., Offutt, J., & Kwon, Y. R. (2005). "MuJava: An automated class mutation system". In Software Testing, Verification and Reliability, 15(2), 97-133.

[140] Marinescu, R. (2001). “Detecting design flaws via metrics in object-oriented systems”. In International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, (pp. 173-182).

[141] Marinescu, R. (2004). “Detection strategies: Metrics-based rules for detecting design flaws”. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on (pp. 350-359). IEEE.

[142] Martin, R. C. (2003). “Agile software development: principles, patterns, and practices”. Prentice Hall PTR.

[143] Mathur, A. P., & Wong, W. E. (1994). “An empirical comparison of data flow and mutation-based test adequacy criteria”. In Software Testing, Verification and Reliability, 4(1), 9-31.

[144] McKenney, P. E. (2013). “Structured deferral: synchronization via procrastination”. In Communications of the ACM, Vol. 56, No. 7, (pp. 40-49).

[145] McKenney, P. E., & Slingwine, J. D. (1998). “Read-copy update: Using execution history to solve concurrency problems”. In Parallel and Distributed Computing and Systems, (pp. 509-518).

[146] McKenney, P. E., Boyd-Wickizer, S., & Walpole, J. (2013). “RCU usage in the Linux kernel: one decade later”.

[147] McKenney, P. E., Eggemann, D., & Randhawa, R. (2013). “Improving energy efficiency on asymmetric multiprocessing systems”.

[148] Meneely, A., Williams, L., Snipes, W., & Osborne, J. (2008). “Predicting failures with developer networks and social network analysis”. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (pp. 13-23). ACM.

[149] Mens, T. (2002). “A state-of-the-art survey on software merging”. In IEEE transactions on software engineering, 28(5), 449-462.

[150] Menzies, T., Greenwald, J., & Frank, A. (2007). “Data mining static code attributes to learn defect predictors”. In IEEE transactions on software engineering, 33(1), 2-13.

[151] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). “Defect prediction from static code features: current results, limitations, new approaches”. In Automated Software Engineering, 17(4), 375-407.

[152] Mockus, A., Nagappan, N., & Dinh-Trong, T. T. (2009). "Test coverage and post-verification defects: A multiple case study". In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on (pp. 291-301). IEEE.

[153] Moha, N., Rezgui, J., Guéhéneuc, Y. G., Valtchev, P., & El Boussaidi, G. (2008). "Using FCA to suggest refactorings to correct design defects". In Concept Lattices and Their Applications (pp. 269-275). Springer Berlin Heidelberg.

[154] Moser, R., Pedrycz, W., & Succi, G. (2008). "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 (pp. 181-190). ACM.

[155] Myers, G. J., Sandler, C., & Badgett, T. (2011). "The art of software testing". John Wiley & Sons.

[156] Nagappan, N., & Ball, T. (2005). "Use of relative code churn measures to predict system defect density". In Proceedings of the 27th international conference on Software engineering (pp. 284-292). ACM.

[157] Nagappan, N., & Ball, T. (2007). "Using software dependencies and churn metrics to predict field failures: An empirical case study". In Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on (pp. 364-373). IEEE.

[158] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). "Change bursts as defect predictors". In Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on (pp. 309-318). IEEE.

[159] Namin, A. S., & Andrews, J. H. (2009). "The influence of size and coverage on test suite effectiveness". In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 57-68). ACM.

[160] Namin, A. S., & Kakarla, S. (2011). "The use of mutation in testing experiments and its sensitivity to external threats". In Proceedings of the 2011 International Symposium on Software Testing and Analysis (pp. 342-352). ACM.

[161] Nested Virtualization: https://msdn.microsoft.com/en-us/virtualization/hyperv_on_windows/user_guide/nesting

[162] Nieminen, A. (2012). “Real-time collaborative resolving of merge conflicts”. In Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on (pp. 540-543). IEEE.

[163] Offutt, A. J. (1992). “Investigations of the software testing coupling effect”. In ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 1, No. 1, (pp. 5-20).

[164] Offutt, A. J., & Craft, W. M. (1994). “Using compiler optimization techniques to detect equivalent mutants”. In Software Testing, Verification and Reliability, Vol. 4, No. 3, (pp-131-154).

[165] Offutt, A. J., & Pan, J. (1996). “Detecting equivalent mutants and the feasible path problem”. In Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference, (pp. 224-236). IEEE.

[166] Offutt, A. J., & Untch, R. H. (2001). “Mutation 2000: Uniting the orthogonal”. In Mutation testing for the new century (pp. 34-44). Springer, Boston, MA.

[167] Offutt, A. J., & Voas, J. M. (1996). “Subsumption of condition coverage techniques by mutation testing”.

[168] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). “An experimental determination of sufficient mutant operators”. In Transactions on Software Engineering and Methodology, Vol. 5, No. 2, (pp. 99-118).

[169] Offutt, A. J., Pan, J., Tewary, K., & Zhang, T. (1996). “An experimental evaluation of data flow and mutation testing”. In Softw., Pract. Exper, Vol. 26, No. 2, (pp. 165-176).

[170] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009). “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 (pp. 390-400). IEEE Computer Society.

[171] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). “Predicting the location and number of faults in large software systems”. In IEEE Transactions on Software Engineering, 31(4), 340-355.

[172] Pacheco, C., & Ernst, M. D. (2005). “Eclat: Automatic generation and classification of test inputs”. Springer Berlin Heidelberg, (pp. 504-527).

[173] Pacheco, C., & Ernst, M. D. (2007). “Randoop: feedback-directed random testing for Java”. In Companion to the 22nd conference on Object-oriented programming systems and applications, (pp. 815-816). ACM.

[174] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., & Muller, G. (2011). “Faults in linux: ten years later”. In Computer Architecture News, Vol. 39, No. 1, (pp. 305-318). ACM.

[175] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2013). “Detecting bad smells in source code using change history information”. In Automated software engineering (ASE), (pp. 268-278).

[176] Papadakis, M., & Malevris, N. (2010). “An empirical evaluation of the first and second order mutation testing strategies”. In Third International Conference on Software Testing, Verification, and Validation Workshops, (pp. 90-99). IEEE.

[177] Papadakis, M., & Malevris, N. (2010). “Automatic mutation test case generation via dynamic symbolic execution”. In 21st international symposium on Software reliability engineering, (pp. 121-130). IEEE.

[178] Papadakis, M., Jia, Y., Harman, M., & Le Traon, Y. (2015). “Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique”. In 37th IEEE International Conference of Software Engineering, (pp. 936-946). IEEE.

[179] Patrick, M., Oriol, M., & Clark, J. A. (2012). “MESSI: Mutant evaluation by static semantic interpretation”. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on (pp. 711-719). IEEE.

[180] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). “Scikit-learn: Machine learning in Python”. In Journal of machine learning research, 12(Oct), 2825-2830.

[181] Pinzger, M., Nagappan, N., & Murphy, B. (2008). “Can developer-module networks predict failures?”. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (pp. 2-12). ACM.

[182] Rahman, F., & Devanbu, P. (2011). “Ownership, experience and defects: a fine-grained study of authorship”. In Proceedings of the 33rd International Conference on Software Engineering (pp. 491-500). ACM.

[183] Read-copy-Update (RCU): https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

[184] Roy, C. K., & Cordy, J. R. (2007). A survey on software clone detection research. Queen's School of Computing TR, 541(115), 64-68.

[185] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-1789B, RTCA, Inc., 1992.

[186] Rutar, N., Almazan, C. B., & Foster, J. S. (2004). "A comparison of bug finding tools for Java". In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on (pp. 245-256). IEEE.

[187] Sarma, A., & Van Der Hoek, A. (2006). "Towards awareness in the large". In Global Software Engineering, 2006. ICGSE'06. International Conference on (pp. 127-131).

[188] Sarma, A., Noroozi, Z., & Van Der Hoek, A. (2003). "Palantír: raising awareness among configuration management workspaces". In International Conference on Software Engineering, (pp. 444-454).

[189] Schuler, D., & Zeller, A. (2009). "Javalanche: efficient mutation testing for Java". In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 297-298). ACM.

[190] Schuler, D., & Zeller, A. (2010). (Un-) Covering Equivalent Mutants. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on (pp. 45-54). IEEE.

[191] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010). "Building empirical support for automated code smell detection". In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (p. 8). ACM.

[192] Servant, F., Jones, J. A., & Van Der Hoek, A. (2010). "CASI: preventing indirect conflicts through a live visualization". In Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering (pp. 39-46). ACM.

[193] Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., McMinn, P., & Arcuri, A. (2015). "Do automatically generated unit tests find real faults? an empirical study of

effectiveness and challenges". In Automated software engineering (ASE), 2015 30th IEEE/ACM international conference on (pp. 201-211). IEEE.

[194] Shi, A., Gyori, A., Gligoric, M., Zaytsev, A., & Marinov, D. (2014). "Balancing trade-offs in test-suite reduction". In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 246-256). ACM.

[195] Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., & Hassan, A. E. (2010). "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 (p. 4). ACM.

[196] Shihab, E., Mockus, A., Kamei, Y., Adams, B., & Hassan, A. E. (2011). "High-impact defects: a study of breakage and surprise defects". In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 300-310). ACM.

[197] SIR: Software-artifact Infrastructure Repository. Sir usage information, accessed at March 8, 2016. http://sir.unl.edu/portal/usage.php.

[198] Smith, B. H., & Williams, L. (2009). "On guiding the augmentation of an automated test suite via mutation analysis". In Empirical Software Engineering, Vol. 14, No. 3, (pp. 341-369).

[199] Spacco, J., Hovemeyer, D., & Pugh, W. (2006). "Tracking defect warnings across versions". In Proceedings of the 2006 international workshop on Mining software repositories (pp. 133-136). ACM.

[200] Sun, Z., Song, Q., & Zhu, X. (2012). "Using coding-based ensemble learning to improve software defect prediction". In IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42(6), 1806-1817.

[201] Tengeri, D., Vidács, L., Beszédes, Á., Jász, J., Balogh, G., Vancsics, B., & Gyimóthy, T. (2016). "Relating code coverage, mutation score and test suite reducibility to defect density". In Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on (pp. 174-179). IEEE.

[202] Tian, Y., Lawall, J., & Lo, D. (2012). “Identifying linux bug fixing patches”. In Proceedings of the 34th International Conference on Software Engineering (pp. 386-396). IEEE Press.

[203] Tiobe, http://tiobe.com/index.php/content/paperinfo/tpci/index.html

[204] Understand™ Static Code Analysis Tool. (2017).

[205] Wah, K. S. (2000). “A theoretical study of fault coupling”. In Software testing, verification and reliability, Vol. 10, No. 1, (pp. 3-45).

[206] Wei, Y., Meyer, B., & Oriol, M. (2012). “Is branch coverage a good measure of testing effectiveness?”. In Empirical Software Engineering and Verification (pp. 194-212). Springer, Berlin, Heidelberg.

[207] Weyuker, E. J., Ostrand, T. J., & Bell, R. M. (2008). “Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models”. In Empirical Software Engineering, 13(5), 539-559.

[208] Wloka, J., Ryder, B., Tip, F., & Ren, X. (2009). “Safe-commit analysis to facilitate team software development”. In Proceedings of the 31st International Conference on Software Engineering (pp. 507-517). IEEE Computer Society.

[209] Wong, W. E., & Mathur, A. P. (1995). “Reducing the cost of mutation testing: An empirical study”. In Journal of Systems and Software, 31(3), 185-196.

[210] Wong, W. E., Delamaro, M. E., Maldonado, J. C., & Mathur, A. P. (1994). “Constrained mutation in C programs”. In Proceedings of the 8th Brazilian Symposium on Software Engineering (pp. 439-452). Brazilian Computer Society.

[211] Yamashita, A., & Moonen, L. (2013). “Do developers care about code smells? an exploratory survey”. In Reverse Engineering (WCRE), 2013 20th Working Conference on (pp. 242-251). IEEE.

[212] Yao, X., Harman, M., & Jia, Y. (2014). “A study of equivalent and stubborn mutation operators using human analysis of equivalence”. In Proceedings of the 36th International Conference on Software Engineering (pp. 919-930). ACM.

[213] Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. (2011). “Investigating the impact of design debt on software quality”. In Proceedings of the 2nd Workshop on Managing Technical Debt (pp. 17-23). ACM.

[214] Zhang, H. (2009). “An investigation of the relationships between lines of code and defects”. In Software IEEE International Conference on Maintenance, 2009. ICSM 2009. (pp. 274-283). IEEE.

[215] Zhang, L., Hou, S. S., Hu, J. J., Xie, T., & Mei, H. (2010). “Is operator-based mutant selection superior to random mutant selection?”. In Software Engineering, 2010 ACM/IEEE 32nd International Conference on (Vol. 1, pp. 435-444). IEEE.

[216] Zhang, L., Marinov, D., & Khurshid, S. (2013). “Faster mutation testing inspired by test prioritization and reduction”. In Proceedings of the 2013 International Symposium on Software Testing and Analysis (pp. 235-245). ACM.

[217] Zimmermann, T., & Nagappan, N. (2007). “Predicting subsystem failures using dependency graph complexities”. In Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on (pp. 227-236). IEEE.

[218] Zimmermann, T., & Nagappan, N. (2008). “Predicting defects using network analysis on dependency graphs”. In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on (pp. 531-540). IEEE.

[219] Zimmermann, T., Kim, S., Zeller, A., & Whitehead Jr, E. J. (2006). “Mining version archives for co-changed lines”. In MSR (Vol. 6, pp. 72-75).

[220] Zimmermann, T., Premraj, R., & Zeller, A. (2007). “Predicting defects for eclipse”. In Proceedings of the third international workshop on predictor models in software engineering (p. 9). IEEE Computer Society.

APPENDIX

Here we list all statistical tests reported in the dissertation.

Pearson Correlation tests

In Table 2.2 we checked the correlation between total number of bug-fixes per line and mutation score using Pearson Correlation test. In Table 2.3 we investigated the correlation between total number of bug-fixes per line and statement coverage using Pearson Correlation test. In Table 4.7 we investigated the correlation between conflict and smell count using Pearson Correlation test. All these tests were performed with a significance level of 0.05.

Kendall τ_b tests

In Table 2.2 we investigated the correlation between total number of bug-fixes per line and mutation score using Kendall τ_b test. In Table 2.3 we investigated the correlation between total number of bug-fixes per line and statement coverage using Kendall τ_b. In Table 4.9 we investigated the correlation between code smell and count of merge conflicts in semantic conflict category using Kendall τ_b test. All these tests were performed with a significance level of 0.05.

Two-sample t-test

In Table 2.4 we investigated the difference between mean number of bug-fixes for covered vs. uncovered program elements using a two-sample t-test. In Table 2.5 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below mutation score thresholds of 0.25 and 0.5 using a two-sample t-test. In Table 2.6 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below mutation score thresholds of 0.75 and 1.0 using a two-sample t-test. In Table 2.7 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below statement coverage score thresholds of 0.25 and 0.5 using a two-

sample t-test. In Table 2.8 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below statement coverage score thresholds of 0.75 and 1.0 using a two-sample t-test. In Table 2.9 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below mutation score thresholds with uncovered program elements filtered out with the thresholds of 0.25 and 0.5 using a two-sample t-test. In Table 2.10 we investigated the difference between mean number of normalized bug-fix commits per line for both above and below mutation score thresholds with uncovered program elements filtered out with the thresholds of 0.75 and 1.0 using a two-sample t-test. All these tests were performed with a significance level of 0.05.

Mann-Whitney test:

In Table 4.6 we investigated the difference in mean number of smells in conflicts vs. non-conflict commits using Mann-Whitney test with an adjusted significance level of 0.0031 obtained using Bonferroni correction due to multiple tests.

Regression model:

In Table 4.10 we report the Poisson regression model for predicting bug-fix occurrence on lines of code involved in a merge conflict. Below is the model

Number of bug-fixes= β_0 + 3.195 * Inward Dependencies - 0.053 * Outward Dependencies - 3.799 * Noncore author + 0.129 * Number of Authors - 0.373 * Number of Classes + 0.244 * Number of Methods + 0.001 * AST diff + 0.00002571 * LOC diff + 0.427 * Number of Smells