Fall 2022

Empirical Studies on Automated Software Testing Practices

Alireza Salahirad

Empirical Studies on Automated Software Testing Practices

by

Alireza Salahirad

Bachelor of Science
Shahid Beheshti University 2014

Master of Science
University of South Carolina 2018

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2022

Accepted by:

Gregory Gay, Major Professor

Marco Valtorta, Committee Member

Csilla Farkas, Committee Member

Pooyan Jamshidi, Committee Member

Ehsan Mohammadi, Committee Member

Dr. Cheryl L. Addy, Interim Vice Provost and Dean of the Graduate School

ACKNOWLEDGMENTS

Years of hard work as a computer scientist have culminated in this dissertation. My efforts alone would not have taken me to this point. First and foremost, praises and thanks to God, the Almighty, for countless blessings in my life. Next, my sincere gratitude goes to my supervisor, Professor Gregory Gay, for his immense knowledge, support, mentorship, and patience throughout this process. Without his support, I would not have been able to make this journey. Furthermore, I would like to thank my co-advisor, Professor Ehsan Mohammadi, for all his guidance. I also thank the members of my dissertation committee, Professor Marco Valtorta, Professor Csilla Farkas, and Professor Pooyan Jamshidi, for their valuable feedback. I am grateful to all my friends for all their encouragement and help, Shervin Ghasemlou, Mohammadreza Haghpanah, Hazhar Rahmani, Hussein Almulla, Ying Meng, Russell Kan, and all my other friends I have missed mentioning. Besides, I would like to thank Alex Forward and all my colleagues at PhotoniCare Inc. Being part of this elite team and working on numerous diverse advanced projects has been one of the most exciting chapters of my life. My family has always been one of my most significant sources of support. Last but not least, my sincere thanks go to them for all their encouragement and sacrifices during the years I have been away from them.

ABSTRACT

Software testing is notoriously difficult and expensive, and improper testing carries economic, legal, and even environmental or medical risks. Research in software testing is critical to enabling the development of the robust software that our society relies upon. This dissertation aims to lower the cost of software testing without decreasing the quality by focusing on the use of automation. The dissertation consists of three empirical studies on aspects of software testing. Specifically, these three projects focus on (1) mapping the connections between research topics and the evolution of research topics in the field of software testing, (2) an assessment of the metrics used to guide automated test generation and the factors that suggest when automated test generation can detect real faults, and (3) examination of the semantic coupling between synthetic and real faults in service of improving our ability to cost-effectively generate synthetic faults for use in assessing test case quality.

- **Project 1 (Mapping):** Our main goal for this project is to understand better the emergence of individual research topics and the connection between these topics within the broad field of software testing, enabling the identification of new topics and connections in future research. To achieve this goal, we have applied co-word analysis in order to characterize the topology of software testing research over three decades of research studies based on the keywords provided by the authors of studies indexed in the Scopus database.

- **Project 2 (Automated Input Generation):** We have assessed the fault-detection capabilities of unit test suites generated by automated tools with the goal of satisfying eight fitness functions representing common testing goals. Our purpose was not only

to identify the particular fitness functions that detect the most faults but to explore further the factors that influence fault detection. To do this, we gathered observations on the generated test suites and metrics describing the source code of the faulty classes and applied a rule-learning algorithm to identify the factors with the strongest influence on fault detection.

- **Project 3 (Mutant-Fault Coupling):** Synthetic faults (*mutants*), which can be inserted into code through transformative *mutation operators*, offer an automated means to assess the effectiveness of test suites and create new test cases. However, mutants can be expensive to utilize and may not realistically model real faults. To enable the cost-effective generation of mutants, we investigate this semantic relationship between mutation operators and real faults.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software testing is the process of executing *input* against a system-under-development and comparing the resulting *output* to a set of expectations (known as an *oracle*) [1, 2]. This activity aims to evaluate the correctness of software functionality through observations of the systems' reactions to direct and indirect stimuli.

Software testing plays a central role in ensuring the reliability of software that powers our society. While many quality assurance techniques exist, testing remains the most well-known and widespread technique for software verification. Software testing is one of the largest areas of software engineering research [3–5], and research in this area has led to significant advances in quality assurance practices in the real world.

However, testing is notoriously difficult and expensive, and improper testing carries economic [6], legal, and even environmental [7] or medical risks [8]. Further research advances in software testing are critical to enabling the development of the robust software that our society relies upon.

1.1 STATEMENT OF THE PROBLEM

The correctness of software is difficult to prove conclusively, as most software has a near-infinite set of possible input combinations. Therefore, testing cannot prove the correctness of software. It can only prove the "incorrectness" of software through the selection of input revealing a fault [9]. It takes time, resources, and money to develop, maintain, and execute software tests. Furthermore, performing testing requires the assignment of clear responsi-

bilities and ensuring that team members have the necessary skills. Since we do not know what inputs reveal faults, various configurations and inputs must be tested, including coverage of different functional outcomes, input sanitization, testing for timeouts, managing unexpected events, and so on. Up to half of the lines of code in a project can be devoted to test cases and testing infrastructure [10]. Selecting input scenarios, running and evaluating these scenarios, and measuring testing progress are all needed.

Our **long-term goal** is to lower the cost of software testing, with a focus on automation. Much of the cost of testing discussed above can be traced directly to the human effort required to conduct most testing activities, such as producing test input and expected output [11]. One way of lowering testing costs may lie in the use of automation to ease this large manual burden. Rather than relying on the human effort at all stages of the testing process, various steps could be performed automatically—for example, selecting potentially failure-revealing input scenarios. This makes it possible to increase test coverage without spending more money. The use of automation also helps to discover faults earlier. Testing automation leads to shorter software development cycles and faster delivery times [12]. However, research in software test automation is important. There are still limitations to overcome in automated testing, despite its many benefits. Some of these challenges include the effectiveness of automation—compared to human-performed testing—as well as up-front costs, training of users of the technologies, the usability of automation, and the cost of maintaining automatically-generated test cases [13].

1.2 Purpose of the Dissertation

This thesis consists of three **specific projects** that address challenges that hinder the attainment of the long-term goal of reducing the cost of software testing. Each project is centered around an *empirical study* relating to an important topic in the software testing field, with a particular focus on automation.

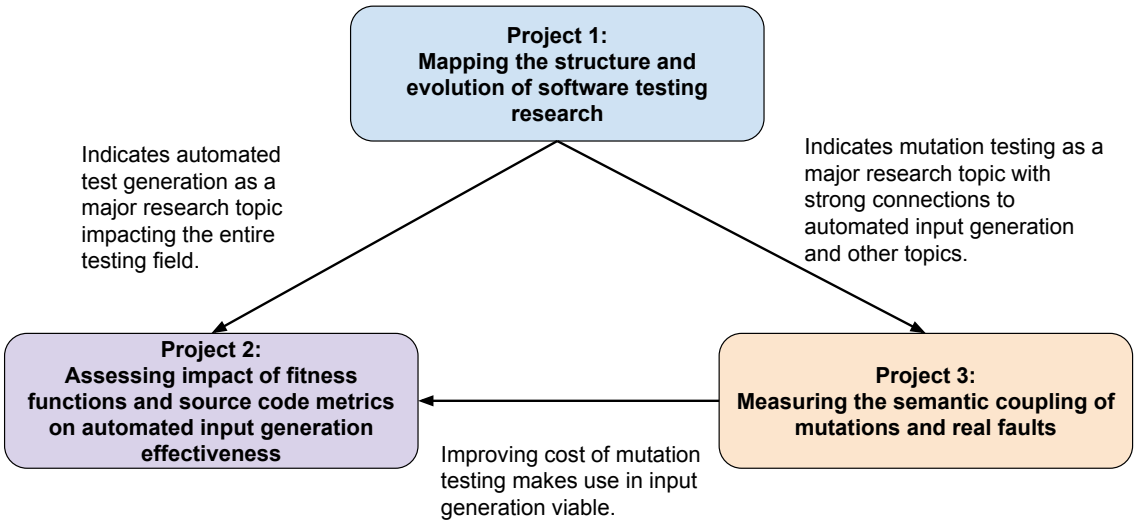


Figure 1.1: The three projects that make up this dissertation, with connections.

Specifically, these three projects focus on (1) mapping the connections between research topics and understanding the evolution of research topics in the field of software testing through the use of network analysis techniques, (2) assessment of the criteria used to guide automated test input generation and exploration of the factors that influence the ability of automated input generation to trigger failures, and (3) examination of the semantic coupling between synthetic and real-world faults to identify the types of synthetic faults best suited for use in assessing and improving test case quality.

These three projects are illustrated and connected in Figure 1.1, and are described in more detail below. We apply network analysis techniques in the first project to quantitatively map the software testing research field. This project offers an evidence-based method to characterize research topics in software testing and, more importantly, to identify how these topics are connected to explore how to exploit existing topic synergies best or identify new connections to explore. The findings of this project help motivate further research on the automation of the testing process.

The mapping produced in Project 1 identified automated test generation as one of the most important topics in all of the software testing. It is a major focus in testing research, as automating aspects of test creation—such as input selection—offers great promise to

reduce costs by alleviating common effort-intensive tasks. Most software functions have a near-infinite number of possible inputs. It is difficult and time-consuming for humans to identify input scenarios likely to trigger faults in systems. Therefore, in Project 2, we explore many of the criteria currently used to generate test input automatically and identify the limitations and strengths of those criteria, as well as the source code factors that affect automated test generation. This research offers insight into automated input generation that can help improve how it is performed in the future.

The general goal of automated input generation is to select failure-triggering input. Currently, many approaches to automated input generation are based on source code coverage with the hypothesis that tests that execute a large portion of the code structure will be more likely to expose faults in that code. Code coverage is a prerequisite to fault detection, but code must still be executed in a manner that triggers and exposes a fault. An alternative to code coverage is mutation coverage—detection of planted synthetic faults. Mutation coverage can be used to select inputs during automated test generation and may demonstrate a higher probability of fault detection than code coverage alone. The mutation topic was also identified in Project 1 as a major research topic. However, mutation coverage is very expensive. In Project 3, we explore which types of mutations have the most substantial semantic relationship to real-world software faults. Therefore, Project 3 can lead to insights that reduce the cost and increase the effectiveness of mutation testing on complex systems. In turn, this will make mutation a viable source of information for automated input selection.

Project 1 (Mapping)

Testing is one of the largest areas of software engineering research [3], and the field rapidly evolves as new software and hardware advances are introduced. The goal of this project is to enable both researchers and practitioners to better understand (a) *what the predominant research topics are of the field*, (b) *how those topics are connected*, and (c), *how the*

predominant topics have evolved over time.

In the past, most overviews of the field of testing have been based on qualitative examination of research. In this project, we have applied a quantitative network analysis method—co-word analysis—to visualize and analyze the topology of 35 years of software testing research based on the author-assigned keywords of Scopus-indexed publications. Co-word analysis yields an undirected network where the nodes—author-assigned keywords—represent targeted research concepts. Weighted edges connect keywords based on their co-occurrence on publications. Finally, keywords are grouped into clusters, representing densely-connected regions of the network.

Our analysis maps keywords into dense clusters, from which emerge high-level research topics—themes that characterize each cluster—and makes clear the connections between keywords and topics within and across clusters. It also characterizes the periods in which low-level keywords and high-level topics have emerged—identifying emerging research areas, as well as those where research interest has decreased. This snapshot of important disciplinary trends can provide valuable insight into the state of the field, suggest topics to explore, and identify connections (or lack thereof) between keywords and topics that may reveal new insights.

Among others, we have made the following observations:

- Both the most common author-assigned keywords and the keywords that attract the most citations, on average, tend to relate to automation, test creation and assessment guidance, assessment of system quality, and cyber-physical systems.

- These keywords can be clustered into 16 topics: automated test generation, creation guidance, evolution and maintenance, machine learning and predictive modeling, model-based testing, GUI testing, processes and risk, random testing, reliability, requirements, system testing, test automation, test case types, test oracles, verification and program analysis, and web application testing. Below these lie 18 more

subtopics.

- Creation guidance, automated test generation, evolution and maintenance, and test oracles are particularly multidisciplinary topics with dense connections to many other topics. Twenty keywords connect topics, reflecting multidisciplinary concepts, common test activities, and test creation information.

- Emerging research particularly relates to web and mobile applications, ML and AI—including autonomous vehicles—energy consumption, automated program repair, or fuzzing and search-based test generation. Web applications require targeted testing approaches and practices, leading to emerging connections to many topics. Test oracles are also a rapidly-evolving topic with many emerging connections. ML has emerging potential to support automation.

- Research related to random and requirements-based testing may be in decline.

These insights—and the rich underlying networks of keywords—can inspire both current and future researchers in the field of software testing.

Project 2 (Automated Input Generation)

With exponential growth in the complexity of software, the cost of testing has risen accordingly. Means of lowering the cost of testing without sacrificing verification quality are needed. Automation has great potential in this respect, as much of the invested human effort is in service of tasks that can be framed as *search* problems [14]. For example, test input generation can naturally be seen as a search problem [11]. Hundreds of thousands of test cases could be generated for any particular class-under-test (CUT). Given a well-defined testing goal and a numeric scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can systematically search the space of possible test inputs to locate those that meet that goal [15].

The effective use of search-based generation relies on the performance of two tasks—selecting a measurable test goal and selecting an effective fitness function for meeting that goal. Adequacy criteria offer checklists of measurable test goals based on the program source code, such as the execution of branches in the control flow of the CUT [1, 16, 17]. Often, however, goals such as "coverage of branches" are an approximation of a goal that is harder to quantify—we really want tests that will reveal faults [18]. "Finding faults" is not a goal that can be measured and cannot be translated into a distance function. To generate effective tests, we must identify criteria—and corresponding fitness functions—that are correlated with an increased probability of fault detection.

The goal of this project is to examine whether common fitness functions can produce effective test input for triggering and detecting real-world faults, as well as to understand the factors that contribute to the success or failure of automated input generation. In this study, we have used EvoSuite and eight of its white-box fitness functions (as well as the default multi-objective configuration and a combination of branch, exception, and method coverage) to generate test suites for the fifteen systems, and 593 of the faults, in the Defects4J database.

In each case, we seek to understand *when and why* generated test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques and could inspire new approaches. Thus, in each case, we have recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. We have recorded a set of traditional *source code metrics*—sixty metrics related to cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics—for each class associated with a fault in the Defects4J dataset. By analyzing these generation factors and metrics, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite but—through the use of machine learning algorithms—the factors correlating with a high or low likelihood of fault detection.

To summarize our findings:

- Branch coverage is the most effective criterion. However, regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect.
- While EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria. However, a combination of branch, exception and method coverage outperformed each of the individual criteria. It is more effective because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include a high line or branch coverage over either version of the code and high coverage of their own test obligations. Coverage does not ensure success, but it is a prerequisite.
- The most important factor differentiating cases where a fault is occasionally detected and cases, where a fault is consistently detected is the satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes and thrive when the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Our observations suggest that successful criteria thoroughly explore and exploit the

code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault and ensures that it manifests in failure. Criteria such as exception, output, and weak mutation coverage are situationally useful and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

Our observations provide evidence for the anecdotal findings of other researchers [19–23] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. While more research is still needed to understand better the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites, our findings in this revised and extended case study offer lessons in understanding the use, applicability, and combination of common fitness functions.

Project 3 (Mutant-Fault Coupling)

When designing test cases, past experience can be used to estimate the potential effectiveness of the test suite. If we have known software *faults*—mistakes in the source code [1]—we can use detection of these faults to predict whether test cases will be effective against unknown future faults. Essentially, this is an estimation of the sensitivity of the test suite to changes in the source code. In practice, we typically lack a sufficiently large collection of faults to draw reasonable conclusions. Instead, we make use of synthetic faults, known as *mutants* [24].

Mutation testing [25] is a technique in which a user generates many faulty versions of a program—the "mutants" mentioned above—through small modifications of the original code, typically using automated code transformation [24, 26]. *Mutation operators* define

transformations over code structures, such as expressions, operators, or references [27]. For example, a mutation operator may change one arithmetic operator into another—turning $A+B$ into $A*B$—permute the order of two statements, add or remove a `static` modifier, or many other possible changes. There are many mutation operators used in practice [27, 28]. These operators vary in complexity in effect, but all are intended to reflect common, minor mistakes that developers make when writing code.

Mutation testing is a common technique in both testing research and industrial practice. In research, it is the most common method of judging the effectiveness of new testing techniques, particularly those used to automatically generate test cases [26]. Mutation is also employed at companies such as Google to identify areas of improvement in test design [29]. In either case, the core hypothesis is those test suites that detect mutants are also effective at detecting real faults, as they are sensitive to these small changes in the code [30].

This hypothesis hinges on the idea that mutants can serve as stand-ins for real faults. Mutants clearly bear little *syntactic* resemblance to real faults [31]. A glance at any database of real faults, such as Defects4J for Java faults [32], makes it clear that real faults are generally more complex than mutants, often affecting multiple lines of code and require multiple changes to any single line to fix. Instead, the idea that mutants can substitute for real faults is based on the assumption of a *semantic* relationship built on two hypotheses. The first, the "competent programmer hypothesis", suggests that many programs are close to correct and that minor changes will be enough to fix them. The second, the "coupling effect", suggests that the detection of many simple mutants will equate to the detection of a single complex fault affecting the same lines of code [1, 33].

However, the truth of these hypotheses—or even the broader hypothesis that, regardless of a semantic relationship, that high levels of mutant detection will lead to increased probability of faulty detection—is not clear. Even if mutation testing can improve the quality of testing efforts, weak or contradictory empirical results and the immense cost of

applying mutation testing to a large codebase [33] suggest the need for improvement in the implementation and application of mutation testing.

We hypothesize that improving the effectiveness—in terms of both cost and quality—lies in better understanding the semantic relationship between mutants and real faults, known as their *coupling*. In particular, and in contrast to past studies, we turn our focus to examining the mutation operators. That is, which mutation operators produce the most (or least) mutants that lead to the same outcomes as real faults?

In this study, we investigate the degree of coupling between mutants and real faults by executing developer-written test suites against both mutated and faulty versions of classes from multiple open-source Java projects, based on 144 case examples from the Defects4J fault database [32]. In particular, we focus on the *trigger tests*—the tests that detect the real fault. A mutant that is most strongly coupled to a real fault will be detected only by the trigger tests, and those tests will fail for the same reasons—i.e., the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons. We have defined a scale rating the strength of the coupling between a mutant and a corresponding real fault based on the number of failing tests and reasons for failure. This scale, in turn, allows us to contrast 31 mutation operators—applied using the muJava++ framework—based on their tendency to produce mutants with a stronger semantic relationship to real faults.

Understanding this tendency could enable improvements in how mutation testing is applied. Identifying the mutation operators that most closely semantically model real faults allows prioritization of the mutants used during testing. The exclusion of weakly-coupled mutation operators could lead to large cost savings and filtering of ”noise” from test suite adequacy estimation. In addition, understanding semantic coupling enables potential improvements in the implementation of existing mutation operators and may suggest new mutation operators.

1.3 CONTRIBUTIONS OF THE DISSERTATION

Each project offers its own intellectual merit and advancement of state of the art.

Project 1 (Mapping)

The intellectual merit of this project is in its use of data-driven quantitative techniques to provide researchers with perspectives about the hidden structures and the evolution of the software testing field. This technique reduces the risk of biased interpretation and enables the identification of patterns that would be difficult for humans to detect in qualitative analysis.

Our insights and, more importantly, the rich underlying networks of research topics have the potential to inspire both current and future researchers. For potential researchers, a snapshot of important disciplinary trends and authorship patterns can provide valuable insight into the current state of the field. For researchers operating in the field, the mapping data can suggest topics and connections to explore. The connections (or lack thereof) between topics that can reveal new insights. The additional dimension of publication dates also can offer insight into topics that are emerging, from which new connections to existing topics can be forged. We have also made our data available so that others may make additional observations or broaden the horizons of their own research and collaborations.

Project 2 (Automated Input Generation)

The intellectual merit of this project is that it offers researchers and practitioners a greater understanding of the use, applicability, and combination of common fitness functions. The impact of specific fitness functions on fault detection has not been examined in depth before, especially with regard to real faults. Our insights enable more effective use of current search-based test input generation tools and inform the design of future tools.

In addition, our examination of the impact of source code metrics reveals class features

that can affect the effectiveness of search-based input generation, regardless of the specific fitness functions used. Our observations provide evidence for the anecdotal findings of other researchers [19–23] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. We have additionally made our datasets, as well as new Defects4J case examples, available to other researchers to aid in future advances.

Project 3 (Mutant-Fault Coupling)

The intellectual merit of this project lies in enabling a deeper understanding of the semantic relationship between synthetic faults (mutations) and real faults. Mutation testing is advocated on the premise that mutants reflect the mistakes that programmers make. While the technique could be valuable as an analysis of the sensitivity of test cases to small code changes, even if this core hypothesis is not true, insight into this relationship enables improvements in the cost and effectiveness of the practice.

Under the current implementation, mutation testing is too expensive for practical, real-world use. Our research offers practitioners insight into how to choose a potentially useful subset of mutations. By focusing on operators with a stronger semantic relationship to real faults, they can potentially reduce the cost of mutation testing without decreasing its utility in assessing the strength and sensitivity of test suites. Our findings also offer researchers insight that can be used to make recommendations on how to best use mutation testing, how to change the implementation of existing mutation operators, or a starting position for the development of new mutation operators. Again, we also make our data available for other researchers to use as a basis for additional analysis.

1.4 Publications Resulting from the Dissertation

This dissertation has resulted in the following publications:

- Project 1 (Mapping)
  - Alireza Salahirad, Gregory Gay, and Ehsan Mohammadi. “Mapping the structure and evolution of software testing research over the past three decades”. Journal of Systems and Software (2022) [34]: 111518. Available at `https://arxiv.org/abs/2109.04086`.
- Project 2 (Automated Input Generation)
  - Hussein Almulla, Alireza Salahirad, and Gregory Gay. “Using search-based test generation to discover real faults in Guava”. International Symposium on Search-Based Software Engineering. Springer, Cham, 2017 [35]. Available at `https://greg4cr.github.io/pdf/17guava.pdf`.
  - Alireza Salahirad, Hussein Almulla, and Gregory Gay. “Choosing the fitness function for the job: Automated generation of test suites that detect real faults”. Software Testing, Verification and Reliability 29.4-5 (2019): e1701 [36, 37]. Available at `https://greg4cr.github.io/pdf/19fitness.pdf`.
- Project 3 (Mutant-Fault Coupling)
  - Alireza Salahirad, Gregory Gay. “How Closely are Common Mutation Operators Coupled to Real Faults?”. To be submitted, 2022.

1.5 Structure of this Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 presents a brief overview of the practice of software testing.
- Chapter 3 presents Project 1 (Mapping).
- Chapter 4 presents Project 2 (Automated Input Generation).

- Chapter 5 presents Project 3 (Mutant-Fault Coupling).

- Chapter 6 presents an overall conclusion for this dissertation.

CHAPTER 2

BACKGROUND

2.1 SOFTWARE TESTING

Software testing is the process of identifying errors in software. In this process, the expectations of the software product are checked to ensure the product is free of defects [1]. Evaluation involves executing software and system components manually or using automated tools to evaluate one or more properties. Software testing is a major activity in the verification and validation (V&V) process [38]. Verification determines whether the software has been built according to its requirements. Verification ensures the quality of software applications, designs, and architectures by checking that they match the developers' own expectations of correctness. Validation is the process of assessing whether the software meets the needs of its customers. During the process, the software is evaluated to ensure it fulfills its desired use in an appropriate environment.

Introducing a new product to the market requires software testing as the penultimate step. Costly issues can arise if proper testing is not performed. The consequences of software bugs can be expensive or even dangerous, which is why testing is crucial. History is full of examples of software bugs causing monetary and human loss. For example, a $1.2 billion military satellite launch failed in April 1999 due to a software bug, making it the costliest accident in history [39]. Many more examples can be found (e.g., [8]).

During software testing, a *test suite* containing one or more *test cases* is executed against a *system-under-test* (SUT) [1]. A test case applies one or more input steps to the SUT and checks the resulting actions taken by the SUT against an embedded set of

expectations. As the name implies, a test suite is a collection of test cases that are intended to be used to test a particular set of behaviors in software. As well as detailed instructions or goals for each collection of test cases, test suites often include information about the system configuration to be used during testing [40].

2.2 Components of a Test Case

Each test case consists of the following components:

1. **Setup:** Setup steps create reference data and perform interactions needed to make the SUT ready for the scenario that is the focus of the test case. Setup can be either specific to a single test case or common for multiple test cases.

2. **Test Input:** The specific parameters to provide to a function of the SUT. Depending on the granularity of testing, the input can range from method calls to API calls to actions within a graphical interface.

3. **Test Steps:** Interactions where input is provided to the SUT and observations are recorded. The steps perform tasks such as simulating a request, exchanging data, evaluating responses, etc.

4. **Test Oracle:** The oracle determines the test results by comparing the system's behavior to a set of embedded assumptions. An oracle can be a predefined specification (e.g., an assertion), output from a past version, a model, or even manual inspection by humans [2].

5. **Teardown:** Teardown is called after the invocation of each test method to perform any necessary post-test actions like erasing test data generated during the test.

An example of a test case, written in the JUnit notation, is shown in Figure 2.1. The test input is a string passed to the constructor of the `TransformCase` class, then a call to

```
@Test
public void testPrintMessage() {
   String str = "Test Message";
   TransformCase tCase = new TransformCase(str);
   String upperCaseStr = str.toUpperCase();
   assertEquals(upperCaseStr, tCase.getText());
}
```

Figure 2.1: Example of a unit test case written using the JUnit notation for Java.

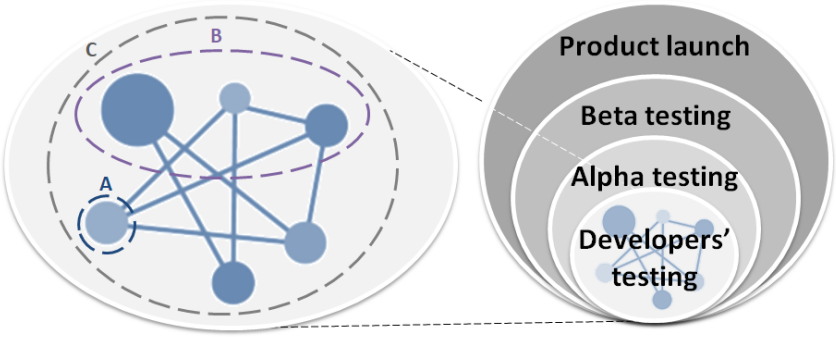


Figure 2.2: Example of granularity levels of the tests to the left and testing phases to the right. A: Unit Testing; B: Integration Testing; C: System Testing.

`getText()`. An assertion then checks whether the output matches the expected output—an upper-case version of the input. The test steps apply the input, and the assertion represents the oracle, which expects the program output to match a pre-recorded value.

2.3 Common Testing Approaches and Practices

As shown in Figure 2.2, testing can take place at multiple levels of granularity, including unit, integration, and system-level testing [1]. During unit testing, the smallest stand-alone elements of the software—generally single classes—are tested in isolation from the rest of the system. During integration testing, module interactions are tested after they have been examined individually during unit testing. Then, during system testing, the SUT is tested through one of its defined interfaces—a programmable interface, a command-line interface, a graphical user interface, or another external interface. System testing generally involves the execution of test cases that verify the compliance of integrated and completed

software with their specifications [41].

Following system testing, multiple human-driven testing practices are common on the final product, including acceptance testing, alpha/beta testing, and exploratory testing [38]. During acceptance testing, an application is assessed to determine if it meets the needs of end users. This usually occurs when software is developed for a specific client. Volunteer users try the product in alpha/beta testing and report any failures observed. In alpha testing, software quality assurance and testing teams perform this type of internal testing with a small pool of volunteers or clients. The alpha test is the last stage of testing the software at the development site. Later, companies release a beta version to external users to gather feedback. A type of software testing called exploratory testing involves testing the system on the spot without creating test cases in advance. The idea behind exploratory testing is to discover, investigate, and learn while coding, and it is used widely in agile software development [42].

Test cases can be designed based on different sources of information. Generally, test design can be based on either sources of information about the software's intended behavior (black-box, or requirements-based testing) or the source code (white-box, or structural testing) [1]. Test design based on both is known as "grey-box testing"). A requirement-based test design involves designing test cases based on test objectives and test conditions derived from requirements or other documentation on program behavior. Requirements-based testing can be further categorized into functional and non-functional testing. Testers verify software functionality according to a set of specifications. The main concern during functional testing is testing how the software behaves—its "functional correctness". In non-functional testing, non-functional requirements such as reliability, usability, and performance are considered. Structural testing is a method of test design that utilizes the internal design of the software to design test cases. Test cases target particular structural elements of the code, such as if-statements or loops, and input is chosen to execute those elements in particular manners recommended by coverage criteria. To implement struc-

tural testing, the test engineer must understand the code's internal executions and how the software is implemented.

Tests are often written after the code-under-test has been developed. However, the practice of test-driven development—which advocates for test creation before code development—has become popular [43].

2.4 The Role of Software Testing in the Software Development Life Cycle

The software development life cycle (SDLC) defines a series of stages or activities that take place with the aim of developing software that meets or exceeds customer expectations and reaches completion within time and cost estimates. The typical SDLC includes the following activities:

1. **Planning:** Analysis of requirements and planning are among software development's most critical aspects. An analysis of the client's needs leads to the generation of the scope document.

2. **Design:** Developers scrutinize the prepared software for compliance with all end-user requirements during the design phase. In addition, if the project is technologically, practically, and financially feasible for the customer. After choosing the best approach to design, the developer selects the appropriate programming languages and technologies for the application.

3. **Implementation:** Software engineers write code according to analyzed requirements during this phase.

4. **Testing:** This stage aims to identify any errors, bugs, or flaws in the software.

5. **Documentation:** Every activity in the project is documented for future reference and improvement.

6. **Deployment and Maintenance:** This stage aims to deploy the released tested software and maintain it, which includes modifying several features over time, observing the system performance, fixing bugs, and implementing requested changes.

These activities can be performed or arranged in a different manner depending on the specific *development process* employed. The software development process consists of breaking down the work into smaller, parallel, or sequential steps to improve the design and product management [38].

The traditional waterfall model presents software development as a sequence of stages, where requirements are specified, the system is designed, code is written, then tests are developed. Each stage builds on the previous stage, and each stage must be completed before the next stage. Completeness and thoroughness are prioritized.

Agile methods have become more popular over time, outside of safety-critical domains such as avionics or medical devices. Agile software development refers to a set of methodologies based on iterative development, which involves cross-functional, self-organized teams collaborating to develop requirements and solutions through early and continuous delivery of small pieces of working software. In waterfall project management, a linear flow is used. Alternatively, agile emphasizes iteration, and small cycles of specification, design, development, and testing are performed. Agile processes are able to make rapid adaptations to changing development conditions or requirements. However, their emphasis on speed and iteration may not be appropriate when there is a high level of risk.

CHAPTER 3

MAPPING THE STRUCTURE AND EVOLUTION OF SOFTWARE TESTING RESEARCH OVER THE PAST THREE DECADES

Background: The field of software testing is growing and rapidly-evolving.

Aims: Based on keywords assigned to publications, we seek to identify predominant research topics and understand how they are connected and have evolved.

Method: We apply co-word analysis to map the topology of testing research as a network where author-assigned keywords are connected by edges indicating co-occurrence in publications. Keywords are clustered based on edge density and frequency of connection. We examine the most popular keywords, summarize clusters into high-level research topics, examine how topics connect, and examine how the field is changing.

Results: Testing research can be divided into 16 high-level topics and 18 subtopics. Creation guidance, automated test generation, evolution and maintenance, and test oracles have particularly strong connections to other topics, highlighting their multidisciplinary nature. Emerging keywords relate to web and mobile apps, machine learning, energy consumption, automated program repair and test generation, while emerging connections have formed between web apps, test oracles, and machine learning with many topics. Random and requirements-based testing show potential decline.

Conclusions: Our observations, advice, and map data offer a deeper understanding of the field and inspiration regarding challenges and connections to explore.

3.1 INTRODUCTION

Software testing refers to the application of input to a system to identify issues affecting its correctness or its ability to deliver services [1]. While many quality assurance techniques exist, testing remains the primary means of assessing software quality.

From nearly the beginning of software development as a discipline, researchers and practitioners have reasoned about testing and quality assurance [44]. Today, testing is one of the largest areas of software engineering research [3], and the field is rapidly evolving as new software and hardware advances are introduced. It is useful, therefore, to understand (a) *what the predominant research topics are of the field*, (b) *how those topics are connected*, and (c), *how the predominant topics have evolved over time*.

"Science of science" describes a research methodology where text, author, and publication metadata are analyzed using quantitative bibliometric and scientometric techniques [45, 46]. Computational methods, such as text mining and citation analysis, map the topical structure of a research field, enabling the discovery of invisible patterns and relationships in the publications that form that field [47, 48].

We have applied co-word analysis to visualize and analyze the topology of 35 years of software testing research, based on the author-assigned keywords of Scopus-indexed publications. Co-word analysis yields an undirected network where the nodes—author-assigned keywords—represent targeted research concepts. Weighted edges connect keywords, based on their co-occurrence on publications. Finally, keywords are grouped into clusters, representing densely-connected regions of the network.

Our analysis maps keywords into dense clusters, from which emerge high-level research topics—themes that characterize each cluster—and makes clear the connections between keywords and topics within and across clusters. It also characterizes the periods in which low-level keywords and high-level topics have emerged—identifying emerging research areas, as well as those where research interest has decreased. The goal of this study is to provide both current and future researchers with perspectives about testing field,

built on a quantitative base. For researchers, a snapshot of important disciplinary trends can provide valuable insight into the state of the field, suggest topics to explore, and identify connections (or lack thereof) between keywords and topics that may reveal new insights. Among others, we have made the following observations:

- Both the most common author-assigned keywords and the keywords that attract the most citations, on average, tend to relate to automation, test creation and assessment guidance, assessment of system quality, and cyber-physical systems.

- These keywords can be clustered into 16 topics: automated test generation, creation guidance, evolution and maintenance, machine learning and predictive modeling, model-based testing, GUI testing, processes and risk, random testing, reliability, requirements, system testing, test automation, test case types, test oracles, verification and program analysis, and web application testing. Below these lie 18 more subtopics.

- Creation guidance, automated test generation, evolution and maintenance, and test oracles are particularly multidisciplinary topics, with dense connections to many other topics. Twenty keywords connect topics, reflecting multidisciplinary concepts, common test activities, and test creation information.

- Emerging research particularly relates to web and mobile applications, ML and AI—including autonomous vehicles—energy consumption, automated program repair, or fuzzing and search-based test generation. Web applications require targeted testing approaches and practices, leading to emerging connections to many topics. Test oracles are also a rapidly-evolving topic with many emerging connections. ML has emerging potential to support automation.

- Research related to random and requirements-based testing may be in decline.

We believe that these insights—and the rich underlying networks of keywords—can inspire both current and future researchers in the field of software testing. We additionally make our data available so that others may make their own observations or broaden the horizons of their own research.[1]

The remainder of this publication is structured as follows. In Section 3.2, we discuss background concepts and related work. In Section 3.3, we explain our methodology. Section 3.4 answers our research questions. In Section 3.5, we provide advice on the use of this data, as well as exploratory analyses related to under-explored and missing connections. Section 3.6 details threats to validity. In Section 3.7, we offer our conclusions.

3.2 Background and Related Work

Bibliometrics and Co-Word Analysis

Bibliometric analysis is "the application of mathematical and statistical methods to books and other means of communication" [49]. Bibliometric studies perform quantitative analysis of publications and associated metadata—e.g., keywords, authors, institutions, and citations—to identify themes and patterns within a research field [50]. Such analysis is often combined with mapping techniques to visualize hidden structures in the metadata of a particular field [51]. The most common analysis methods used include citation-based, co-word (also known as keyword co-occurrence), and co-authorship analysis [52]. We focus on co-word analysis.

In co-word analysis, natural language processing and text mining techniques are used to discover the most meaningful noun phrases in a collection of documents and visualize their meaning in a two-dimensional map [53]. In this map, co-occurring terms are connected, with "closer" placement resulting from stronger co-occurrence. Co-word analysis is generally based on the number of research publications where two keywords are used

[1] A package containing our data is available at `https://doi.org/10.5281/zenodo.7091926`.

Table 3.1: Comparison of our study to other related work, based on the research field, methodologies, and analyses performed.

Topic	This Study	[62]	[63]	[64–67]	[68, 69]	[70]	[71]	[4]	[5]	[3]
Field	Testing	All SE	All SE	All SE	All SE	Sci. SW	SBSE	Testing	Testing	Testing
Method	Quan., Qual.	Quan.	Quan.	Quan.	Quan.	Quan., Qual.	Quan.	Qual.	Qual.	Qual.
Research Topics	✔	✔	✖	✖	✖	✔	✖	✖	✔	✔
Topic Connections	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Keyword Clustering	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Keyword Connections	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Popular Keywords	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Emerging Topics	✔	✖	✖	✖	✖	✖	✖	✔	✔	✔
Declining Topics	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Underexplored Con.s	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Potential Connections	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Popular Papers	✖	✖	✖	✖	✖	✖	✔	✖	✖	✖
Top Authors	✖	✖	✖	✔	✔	✔	✔	✖	✖	✖
Author Location	✖	✔	✔	✔	✔	✔	✖	✖	✖	✖
Pub. Venue	✖	✖	✔	✖	✖	✔	✔	✔	✖	✖

together to describe the research performed [54]. Because keywords succinctly capture the context of a publication, co-word analysis is an effective method of revealing connections between publications [55] and identifying trends in a field [48].

Scholars have previously used co-word analysis to depict the structure of fields including renewable energy [56], global warming [57], nanoscience and nanotechnology [58], human computer interaction [59], and big data [60, 61]. Our study is the first to apply such techniques to software testing.

Bibliometrics and Software Engineering

Our study is the first to apply scientometric or bibliometric techniques to the software testing field. However, bibliometric techniques have been applied to other aspects of software engineering (SE). In Table 3.1, we contrast our study to related work. Below, we further elaborate on the specific studies. In general, we focus on analysis of research topics and the connections between topics, and do not analyze authorship trends. Our focus and chosen analysis methods enable a deep characterization of the connections between topics and low-level publication keywords in software testing.

Garousi and Mäntylä performed a bibliometric analysis of more than 70,000 general SE publications, finding that the most popular research topics were web applications, mo-

bile and cloud computing, industrial case studies, source code, and automated test generation [62]. Our identified research topics include all of these except source code—which is subsumed by other topics—and case studies. In our study, case studies would be categorized based on the problems they address. They also found that a small number of large countries produce the majority of publications, while small European countries are proportionally the most active in the field.

Garousi and Fernandes used the same set of publications to assess questions related to quantity versus the impact of SE research [63]. They broadly found that journal articles have more impact than conference publications and that publications from English-speaking researchers have more visibility and impact. Both studies also used Scopus to gather publications, but had a different focus from our study (all of "software", rather than software testing). The studies also differ in their analysis methods. Rather than co-word analysis, the authors of both studies used citation-based analyses. Co-word analysis allows examination of the connections between topics.

Karanatsiou et al. targeted SE publications from 2010-2017 for analysis, identifying top institutions and scholars from this period [64]. Wong et al. did the same for the periods of 2001-2005 [65], 2002-2006 [66], and 2003-2007 and 2004-2008 [67]. Garousi et al. also performed bibliometric analysis, specifically, on the SE research communities in Canada [68] and Turkey [69]. These studies differ from our own in their focus on the authors of publications, rather than research topics.

Farhoodi et al. reviewed literature related to scientific software, finding that many SE techniques are being applied in the field and that there is still a need to explore the usefulness of specific techniques in this context [70]. Their focus differs in both the analysis techniques, and in their focus on a specific software domain. In Section 3.4, we do observe the emergence of testing research related to scientific software.

De Freitas and de Souza performed a bibliometric analysis on the first ten years of research in search-based software engineering—the use of optimization techniques to au-

tomate tasks [71]. They identified the most cited papers, most prolific authors, and analyzed the distribution of the SBSE publications among conference proceedings, journals, and books. They described networks of collaborations and distributions of publications in various venues and identified the distribution of the number of works published by authors. Their study differs from ours in its focus on a particular research domain, as well as its focus on authors and venues over research topics.

Other Related Work

Purely qualitative analyses of testing research have also been performed. In Table 3.1, we contrast our study to those discussed below. None of these studies perform a full summarization or mapping of the testing field. Instead, they point out research areas that are emerging or that have had a major impact. The topics they discuss tend to form a subset of those in our characterization of the field. In addition, our quantitative analysis methods enable elaborate analyses of the field and the connections between topics not explored in these studies.

Harrold, in 2000, examined past research to identify areas of focus for future research [4]. These areas include improvements in integration testing, use of pre-code artifacts (e.g., specifications) to plan and implement testing activities, development of tools for estimating, predicting, and performing testing on evolving systems, and process improvements. Many of these predictions are now established topics in our map, such as black box testing, evolution and maintenance, and processes and risk.

Bertolino provided a summary of testing research in 2007, and identified achievements in the testing process, reliability testing, protocol testing, test criteria, object-oriented testing, and component-based testing as major advances [5]. She identified outstanding challenges related to testing education, testing patterns, cost of testing, controlled evolution, leveraging users, test input and oracle generation, model-based testing, and testing of specialized domains, among others. Many of her achievements and challenges appear in our

map as either keywords or full research topics.

Orso and Rothermel assessed research performed in the field between 2000-2014, asking colleagues what they believed were the most significant contributions and the greatest challenges and opportunities [3]. The research contributions were categorized into the areas of automated test generation, testing strategies, regression testing, and support for empirical publications. The first three of those areas reflect research topics in our map. Challenges identified included better testing of modern, real-world systems, generation of test oracles, analysis of probabilistic programs, testing non-functional properties (e.g., performance), testing of specialized domains (e.g., mobile), and leveraging of the cloud and crowd. Some of these challenges—e.g., mobile and performance testing—are now research topics in our map.

3.3 METHODOLOGY

Software testing is one of the most popular and fast-growing areas of software engineering research [3]. Although there are many surveys, mapping studies, and systematic literature reviews on individual topics, there is a lack of quantitative examination of the field as a whole—mapping research topics and their connections.

Our primary goal is to provide and analyze a "map" of the field of software testing, based on the many distinct research keywords that form the field and the connections between these keywords, linked through research publications. Our mapping is based on a quantitative method, co-word analysis, that places co-occurring phrases—in our case, author-supplied keywords—in a network. Within this network, keywords appear as nodes, with weighted edges indicating how often keywords are linked in publications. Sets of strongly co-occurring keywords form distinct clusters. This network structure offers a quantitative method to characterize the research field, which can be used as the basis of both qualitative and quantitative analyses.

Using this map, we examine how keywords are linked into *clusters*, characterize clusters using high-level research *topics*, examine the connections between keywords *within* and *across* clusters, and examine how interest in particular keywords and topics have changed *over time*. Specifically, we address the following research questions:

RQ1: What are the most popular individual keywords in software testing, as indicated by the number of publications or citations?

RQ2: What topics characterize the keywords connected *within* each cluster in the map?

RQ3: How are keywords and research topics most strongly linked *across* clusters?

RQ4: What keywords, topics, and connections have emerged or grown in popularity over the past five years?

RQ5: Which keywords and topics have shown the greatest decline in interest?

We begin, in **RQ1**, by examining the individual keywords targeted by authors. We are interested in identifying which keywords have been selected most often, and which receive the most citations per publication on average. We then move into analyses and characterization of the *connections* between keywords.

The goal of **RQ2** is to summarize each cluster. Keywords *within* a single cluster are highly interconnected, providing a basis for identifying *research topics* that encapsulate connected keywords. A topic as a keyword or phrase that connects multiple keywords. For example, "automated test generation" is not just a single keyword, but also a topic that connects other keywords such as "ant colony optimization" and "genetic algorithm" within the same cluster.[2] **RQ3**, then, focuses on the connections *across* clusters, and characterizes how keywords and research topics connect.

RQ4 and **RQ5** focus on an additional dimension, the average age of publications associated with each keyword. In **RQ4**, we identify keywords, topics, and connections between

[2]Both are algorithms often used to generate tests, linking all three keywords as part of the same topic.

keywords and topics that have emerged or grown in popularity in the past five years. In **RQ5**, we examine keywords and topics with the oldest average date of publication—those with a potential decline in interest. These emerging and declining concepts offer insight into how the field is evolving.

To answer these questions, we (1) collected publications related to software testing (Section 3.3), (2) constructed a map, using co-word analysis, of clusters of connected keywords (Section 3.3), (3) removed unrelated or redundant topics (Section 3.3), and (4), analyzed the map and underlying data (Section 3.3).

Data Collection

To gain an inclusive overview of software testing, we gathered publications from the Scopus database. Scopus is a comprehensive meta-database, covering many conference and journal publication venues. We retrieved all publications returned for the search term *"software testing"* on September 26, 2020. Only publications published in English were used. This collection included 57,233 publications.

Following a manual cleaning stage (see Section 3.3), 49,802 publications were included, including 36,774 conference papers, 11,640 journal articles, and 1,388 other articles. Figure 3.1 gives an overview of the number of publications published per year. Our aim was to capture a representative sample of the field, not all possible articles on software testing. When we quote specific numbers of publications, these numbers should not be taken as absolutes, but as the approximate commonality of a topic.

For each study, we gathered the title, author data (names, affiliations, locations), keywords, publication date, venue metadata (e.g., publisher, venue, volume, page numbers), number of citations, DOI, link, and language.

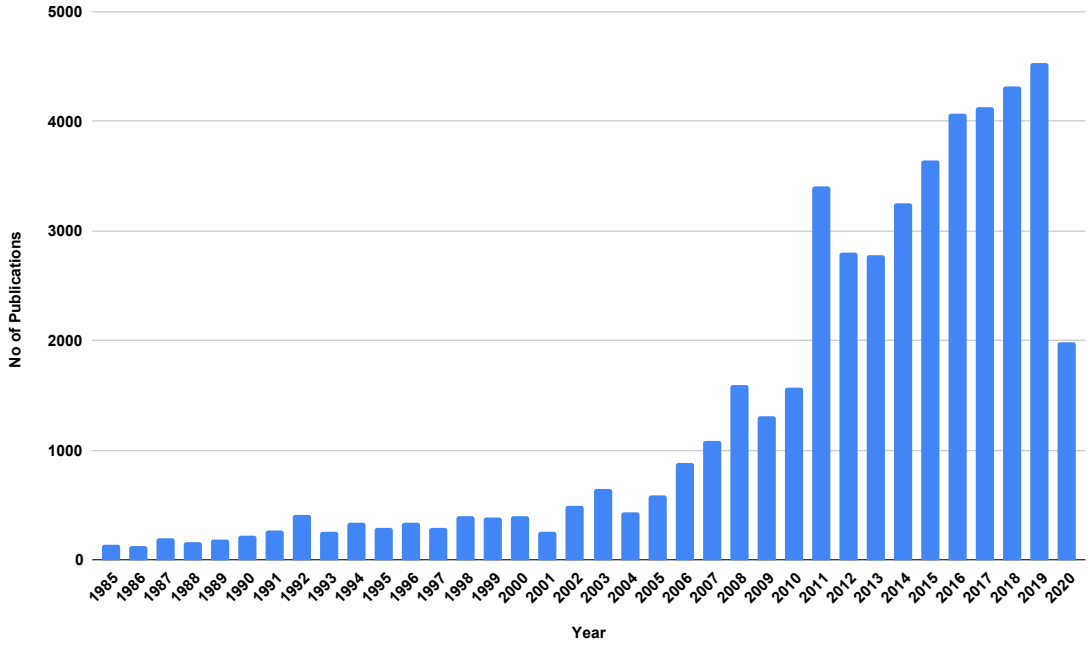


Figure 3.1: Number of publications per year retrieved from Scopus.

Map Construction

To map testing research, we used co-word analysis [53]. Co-word analysis is a natural language processing method that extracts important phrases from a textual dataset and identifies their relationships in a network based on the number of times that two terms co-occur together in all documents. This technique assumes that terms that co-occur more often are more strongly related to each other. As a result, all identified terms are classified into clusters using co-occurrence to measure term similarity and depict the extracted terms and their relationship in a two-dimensional visualization.

We used VOSviewer (Visualization of Similarities Viewer) to analyze the collected data. VOSviewer is a tool that creates maps based on network data [72]. These maps provide visualizations that allow researchers to explore items and relationships. There are various methods for establishing connections between items in these networks, including co-authorship, co-occurrence, citation, bibliographic coupling, and co-citation.

We tested title, abstracts, index keywords, and author-supplied keywords as the unit of

analysis and found that author-supplied keywords are the most promising way to identify research topics and their connections.

In this analysis, we considered 20 as the minimum threshold of keyword occurrences. This threshold places a minimal barrier before a keyword is "important" enough to incorporate. Keywords appearing in fewer than 20 publications were omitted. This threshold was chosen after experimentation as a way to control the level of noise and difficulty of interpretation of the dataset and map, while still avoiding potential loss of interesting and emerging topics. We then iteratively removed keywords that were unrelated to software testing (e.g., publications that used software as part of classroom testing) and merged redundant keywords (e.g., "automated test generation" and "automated test case generation")—see Section 3.3—leaving a final set of 406 keywords.

VOSviewer produces maps based on a co-occurrence matrix—a two-dimensional matrix where each column and row represents an item—a keyword, in our case–and each cell indicates the number of times two keywords co-occur. This map construction consists of three steps. In the first step, a similarity matrix is created from the co-occurrence matrix. A map is then formed by applying the VOS mapping technique to the similarity matrix. Finally, the map is translated, rotated, and reflected. We provide technical details on VOSviewer's algorithm in 3.8.

In VOSviewer, a map is visualized in three ways: The network visualization, the overlay visualization, and the density visualization [73]. We have used the network and overlay visualizations in this study, as well as the raw underlying data.

The network visualization is the standard view, displaying clusters of related items, connected with edges based on their co-occurrence. Figure 3.3 shows the full network visualization that is produced. In Figure 3.2, we highlight a small portion to explain how to interpret the map data.

In this map, each node is a keyword. Figure 3.2 focuses on the keyword "software reliability". All keywords with a sufficiently strong connection to the targeted keyword are

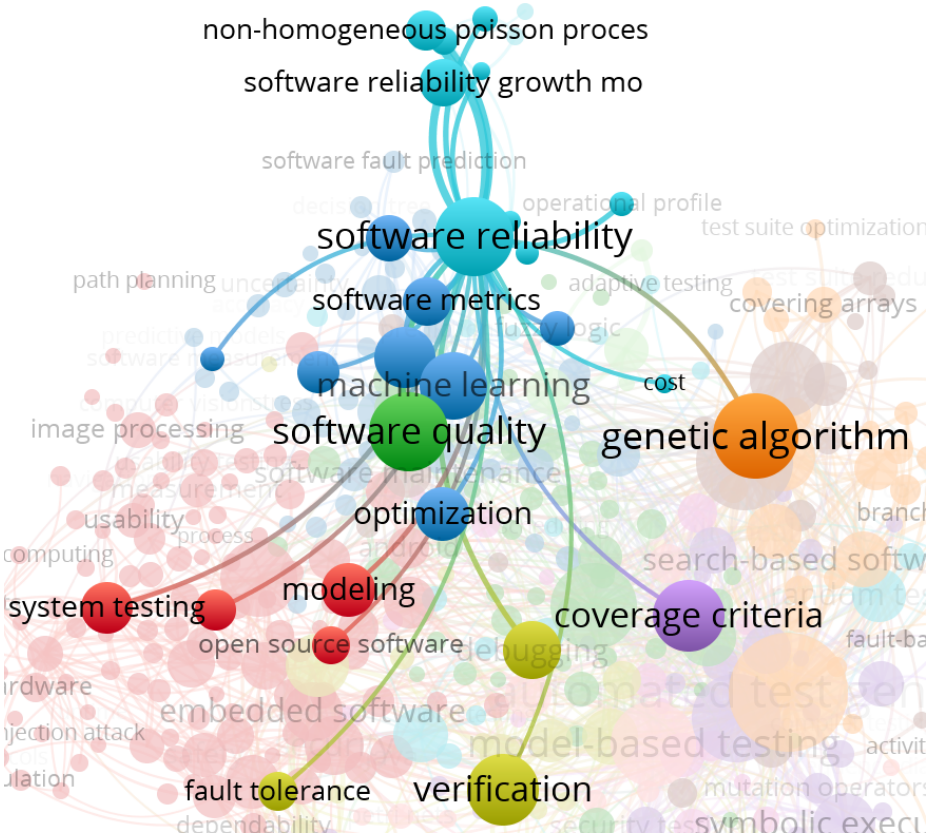


Figure 3.2: Topics associated with software reliability

highlighted, while unrelated keywords are made partially transparent. The size of a node is based on the number of occurrences of the keyword. In Figure 3.2, software reliability is targeted in approximately twice as many publications as "optimization".

Keywords are organized into clusters according to the process described above. Individual keywords can be linked across different clusters. However, the keywords within a cluster tend to be very closely linked with several other keywords within the same cluster. The color of the node indicates its cluster. In Figure 3.2, software reliability is marked in light blue, and other nodes with the same color belong to the same cluster (e.g., "software reliability growth model").

Keywords that co-occur in publications are illustrated with an undirected edge. The thickness of the edge indicates how many publications have targeted both keywords. In Figure 3.2, software reliability and software reliability growth model share a stronger as-

sociation (co-occurring in 35 publications) than software reliability with coverage criteria (5 publications). A user-controlled threshold determines the minimum connection strength for visible edges. We used the default, four publications, to control the level of noise when using the visualization for interpretation. When performing quantitative analyses, we consider all connections, regardless of strength.

The overlay visualization uses colors to indicate certain properties of a node, like the average number of citations that publications targeting a keyword have received, instead of using colors to show the cluster. In our case, we use this visualization to analyze the average age of publications targeting a keyword.

Data Cleanup

The initial data included keywords that were either redundant or irrelevant:

- There are a small number of keywords unrelated to software testing, as the initial sample was gathered using a broad search string. For example, there were keywords related to software-based student examination or software-based testing of hardware. Additional keywords are either too generic to be considered as specific research concepts—e.g., "software testing"—or are research-related terms—e.g., "case study", "empirical study".

- Multiple keywords can refer to the same concept, and can be streamlined into a single keyword—e.g., "automated test generation" and "automated test case generation". The same keyword can appear in singular and plural form—e.g., "test case" and "test cases". There are also American and British English spellings (e.g., "prioritisation" and "prioritization").

To handle irrelevant and redundant keywords, we performed an iterative process. The authors discussed each keyword and came to a consensus. We removed irrelevant keywords from the map, as well as those considered too broad or generic. We removed publications

targeting only those keywords, but retained publications that had additional keywords that remained in our set.

We merged redundant keywords. In performing this process, we limited merging to cases where a redundancy was obvious—primarily pluralization and British/American English. This was to limit the risk of biasing the underlying data that we are using to draw conclusions. We discussed each keyword and its alternatives, and came to a consensus on which keyword to use in all cases. We then replaced the merged keywords with the final keyword for each study and recreated the maps. We performed this process multiple times until we were satisfied that redundant keywords did not remain.

Data Analyses

RQ1 (Popular Keywords): To identify the most common keywords, we sort the keywords by the number of publications that targeted that keyword, and examine those that are targeted in $\geq 0.50\%$ of publications, or ≥ 250 publications. This threshold was chosen by examining the drop-off in significance over the ten most popular keywords and by considering the trade-off between clarity and giving a thorough impression of the testing field. A total of 20 keywords fall above this threshold (4.93% of keywords).

We also have examined which keywords have received the most citations per publication, on average. Here, we examine all keywords with an average number of citations ≥ 20. This threshold yields 23 keywords, and was chosen because it yields a similar quantity to the number of most common keywords, enabling clearer comparison.

RQ2 (Characterization of Clusters into Topics): We perform a qualitative characterization to summarize the field of software testing, supported by the clusters. We perform this summarization by assigning a small number of high-level "topics" to each cluster—keywords or phrases that connect multiple keywords. We have chosen these topics based on our interpretation of the keywords within each cluster. For example, "performance testing" is a keyword that connects, e.g., "load testing", "cloud computing", and "cloud

testing".[3] Because that keyword summarizes many other keywords and connections within that cluster, "performance testing" can also serve as a topic that describes the cluster as a whole.

Some clusters can be summarized by a single topic, while others represent multiple topics. There is no case where *all* keywords are connected to *all* other keywords in the same cluster. Often, two keywords are only indirectly connected through other keywords—e.g., "random testing" is connected to "reliability" and "adaptive random testing", but the latter two are not directly connected in our sample.

We often observed small sub-groupings of keywords within a topic. In such cases, we assign both a topic and a *subtopic*. For example, the topic of test creation guidance can be broken into four distinct types of creation guidance. A keyword can also belong to multiple topics or subtopics, linking topics within a cluster.

To assign topics, all authors examined the keywords within a cluster. We then grouped keywords into one or more groupings. To be grouped, keywords must have a direct or indirect (via a shared keyword) edge. This grouping was made based on our experience, literature, and the map data. We grouped keywords if they were used to perform the same activity, were a technology used to perform an activity, were a source of data for an activity, or had some other clear shared purpose. The proposed groupings were discussed until a consensus could be reached. We then identified either a keyword or concept that characterized each grouping. We discussed the options and reached a consensus on the topic to assign. In cases where a grouping could be split into smaller, but still distinct, groupings, subtopics were identified.

As all keywords must be grouped into a cluster, there are situations where a small number of individual keywords do not relate to the topics assigned to that cluster. We have tried to select topics that are as inclusive as possible.

RQ3 (Connections Across Clusters): In this question, we are interested in characterizing

[3] "load testing" and "cloud testing" are forms of performance testing that target "cloud computing".

how keywords and topics are connected *across* clusters. To do so, we have examined two concepts—measurement of the *density of connections between clusters*, and the identification of *keywords that are connected to many other keywords*.

Connection Density: We can examine how clusters are connected by identifying the cases where the largest percentage of possible connections exist between keywords in two clusters, A and B:

$$\frac{(\text{Number of Connections Between Keywords in Clusters A and B})}{(\text{Number of Keywords in Cluster A}) * (\text{Number of Keywords in Cluster B})} \tag{3.1}$$

A measurement of 1.00 means that all keywords in Cluster A are connected to all keywords in Cluster B.[4]

To identify the most densely interconnected clusters, we measured the connection density between all pairs of clusters. We then focus on the connections with a density ≥ 0.12 (where at least 12% of all possible connections exist). This threshold was chosen after examination of the measurements—23 of the 45 pairs of clusters (50%) have a connection density above this threshold.

Connecting Keywords: Some keywords are singular research concepts, while others serve as "connecting keywords" that link many keywords together. We further characterize connections across clusters by identifying these connecting keywords.

To identify these keywords, we measure the number of keywords that each keyword co-occurs with outside of the cluster where that keyword is located. We analyze all keywords connected to ≥ 100 keywords in external clusters. This threshold was chosen as it yielded the same number of keywords as the threshold for the most popular keyword in RQ1 (20 keywords), enabling direct comparison.

RQ4 (Emerging Keywords, Topics, and Connections): In this question, we are interested in identifying emerging trends in testing research. We can do this by examining individual keywords, research topics, and connections between keywords.

[4]We employ the density because different clusters contain different numbers of keywords. This measurement offers a fairer basis of comparison than the raw number of connections between pairs of clusters.

We have classified any keywords with an average date of publication *later than June 2015* as our set of emerging keywords. This captures an approximate five year period ending with the date we took our sample of publications. A recent date implies one of two things about a keyword: (1) this is a new keyword, or (2), this is a older keyword that has received more attention in recent years.

We are also interested in examining the connections between keywords and topics, as related to the set of emerging keywords. There are a total of 2,029 connections where at least one of the connected keywords is an emerging keyword, of which 1,412 are cross-cluster connections and 617 are within-cluster connections. To focus our analysis, we focus on the cross-cluster connections, allowing us to also examine and characterize the emerging connections between topics.

To identify a subset of those 1,412 connections for further exploration, we use the cross-cluster connection density to identify the pairs of clusters with the highest proportion of emerging connections. We have selected the ten pairs of clusters with the highest proportion of emerging connections for further examination, corresponding to a threshold of $\geq 3.5\%$ of all connections between the two clusters consisting of emerging connections. For each pair of clusters, we group the connections by topic, then examine how the connection between these two topics is being shaped by the emerging connections between low-level keywords.

RQ5 (Declining Keywords and Topics): We address this question following a similar process to RQ4, based on *oldest* average dates of publication. 66 keywords met the threshold in RQ4. Therefore, we also examine the 66 keywords with the oldest dates of publication. This corresponds to a period from November 2001–May 2011.

These keywords represent concepts that are no longer receiving as much interest. This does not imply with certainty that such concepts are no longer relevant, or that they correspond to "solved" challenges. A keyword could represent (a) a topic or concept in decline (e.g., an older technology or approach that has been potentially superseded), (b) a

well-established topic or concept with steady—but not growing—activity, or (c), a topic or concept that had a "boom" period in the past and a lower level of activity in recent years. Keywords may experience a resurgence. However, they have reduced relevancy to current development or testing trends, challenges, and research topics.

Before stating that a particular topic is in decline, we compare the list of keywords and topics with those in RQ4. We say that a topic is declining in interest if both (a) it has several keywords with older average publication dates, **and** (b), lacks keywords with recent average dates. By examining both the oldest and newest keywords, we can more carefully discuss whether a topic is potentially in decline.

3.4 Results and Discussion

Our analyses are based on 406 keywords, which are mapped into 11 clusters. We analyze this map by identifying the most popular keywords by occurrences and citations (Section 3.4) and the overarching research topics of each cluster (Section 3.4), examining how keywords and topics are linked across clusters (Section 3.4), and exploring keywords, topics, and connections that are emerging or in potential decline (Sections 3.4-3.4). We also offer advice and further exploratory analyses in Section 3.5.

A visualization of the keyword map is shown in Figure 3.3. An interactive version of this map can be accessed at `https://greg4cr.github.io/other/2021-TestingTrends/topics.html` or in the replication package.

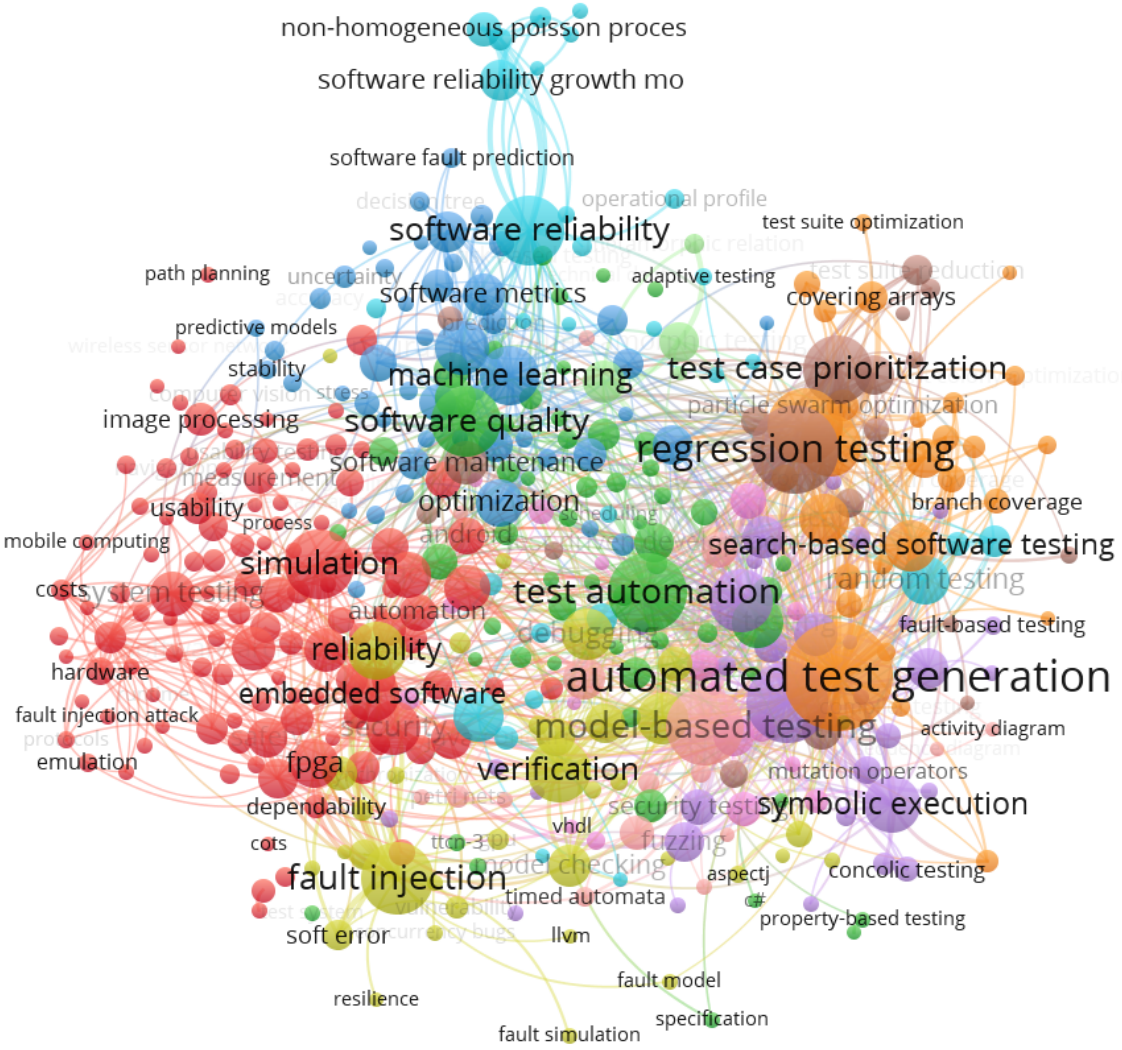


Figure 3.3: A visualization of the connections between publication keywords.

RQ1: Popular Keywords

We begin by identifying the most popular individual keywords, sorted by the number of publications (and percentage of the total sample). These keywords are listed in Table 3.2, along with a description, percentage of the sample, average age of publication (rounded to the year), and average number of citations per publication.

These keywords are those that the authors considered important enough to note as one of the core focuses of their work. There are certainly more than 326 publications in this sample that *use* machine learning, for example. However, the authors may not have listed

Table 3.2: Keywords targeted in at least 0.50% of publications ($\geq$ 250 publications). Each keyword is named and described, and the number of publications where the keyword is targeted, percentage of the sample, average date of publication, and average number of citations per study are included.

Keyword	# Pubs.	Percent	Citations	Date	Description
Automated Test Generation	1068	2.14%	16.36	2013	The use of tools to generate full or partial test cases [11].
Regression Testing	701	1.41%	14.03	2014	A practice where tests are re-executed when code changes to ensure that working code operates correctly [74].
Mutation Testing	596	1.20%	14.83	2014	A practice where synthetic faults are seeded into systems to assess the sensitivity of tests [75].
Test Automation	567	1.14%	9.71	2014	Tools and practices that enable automation of test execution [76].
Model-based Testing	552	1.11%	8.38	2014	Use of behavioral models to analyze the system, to design or generate test cases, or to judge results of testing [11].
Genetic Algorithm	519	1.10%	10.10	2014	An optimization algorithm that models how populations evolve over time [77]. Often used to automate tasks.
Fault Injection	477	0.96%	6.12	2015	Injection of faults into a system for analysis [78].
Software Quality	445	0.89%	8.14	2012	Means to define, measure, and assure the quality of software [79]. Encompasses correctness and quality (e.g., performance or scalability).
Simulation	442	0.89%	4.53	2013	Simulated execution of a system. May encompass how to simulate [80], testing in simulation [81], or obtaining realistic results [82].
Software Reliability	440	0.88%	12.71	2010	Means to define, measure, and assess the how quality changes over time [83].
Test Case Prioritization	418	0.84%	17.42	2015	Automated techniques that select a subset of tests for execution [84].
Verification	366	0.73%	16.58	2012	Techniques that assess whether software possesses a property of interest, often using formal specifications [1]. Testing is one verification technique.
Coverage Criteria	362	0.73%	13.21	2012	Measurements used to assess the strength of a test suite based on how tests exercise code elements [85].
Combinatorial Testing	349	0.70%	14.35	2015	A technique for generating or selecting test input, based on coverage of representative values [86].
Machine Learning	326	0.65%	8.32	2017	Algorithms that make inferences from patterns detected in data. Used in, e.g., automation [2], predictive modeling [87], or evaluation [88].
Reliability	306	0.62%	13.95	2013	Often a synonym for software quality, but can also refer to hardware quality or a blend of hardware/software.
Symbolic Execution	295	0.59%	14.34	2014	Analyses where software is executed in an abstract form where one symbolic input matches many real inputs [89].
Embedded Software	268	0.54%	4.86	2014	Complex self-contained hardware and software systems [90].
Neural Networks	266	0.53%	8.53	2015	Network structures inspired by the human brain, used in machine learning [91].
Security	265	0.52%	7.60	2015	Practices, tools, and techniques intended to prevent misuse of a system's capabilities or data [92].

machine learning as a keyword. Therefore, these keywords should be interpreted as the research concepts the authors felt were the most important and relevant.

RQ1 (Popular Keywords): The most common keywords tend to relate to automation, test creation and assessment guidance, assessment of system quality, and cyber-physical systems.

Automation offers promise for increasing the quality and efficiency of testing, and many keywords (e.g., automated test generation, test automation) relate to automation. Additionally, genetic algorithms and symbolic execution often enable automation. Test case

Table 3.3: Keywords that received more than 20 citations on average per publication, with description, average number of citations, number of publications where the keyword is targeted, and average date of publication.

Keyword	Citations	# Pubs.	Date	Description
Testing Strategies	55.50	26	2010	Guiding principles for test design and the testing process [93].
Testing and Debugging	48.48	21	2013	Debugging practices isolate and diagnose faults in the source code [94]. This keyword relates to the combination of testing and debugging techniques.
Partition Testing	35.59	54	2005	Test input selection based on division of the system's input domain into partitions, based on a set of rules [95].
Fault-based Testing	34.48	33	2005	Use of pre-specified faults in a program to create and evaluate test suites [96]. Mutation testing is an automated form of fault-based testing.
Constraints	33.76	25	2011	Conditions that must be met to accomplish a goal, e.g., for input to take a particular path in a program [89].
Test Suite Minimization	32.97	39	2014	Process of reducing test suite size by eliminating redundant test cases [97].
Random Testing	32.88	218	2011	Testing software by generating random input [98].
Software Fault Prediction	31.82	38	2016	Prediction of fault-prone code using software metrics and fault metadata [99].
Covering Arrays	28.50	96	2013	The set of test specifications selected during combinatorial testing [86].
Compiler Testing	27.28	32	2014	Specialized testing practices for compilers, e.g., the selection of input programs to ensure the compiler conforms to its target language's semantics and syntax [100].
Object Oriented Modeling	25.75	20	2004	Model formats based on object-oriented design and object interaction [101]
Evolutionary Testing	23.93	46	2011	The use of evolutionary algorithms (e.g., genetic algorithms) to generate test input or automate other tasks [11].
Test Design	23.42	33	2013	The process of defining test cases [102].
Regression Test Selection	23.40	58	2014	Practices to test cases for use during regression testing (e.g., only execute tests for changed code) [103].
Monte Carlo	22.78	23	2015	A family of algorithms used for optimization, numeric integration, and probability assessment [104].
Alloy	22.18	22	2014	Language for expressing complex behavior and constraints in software [105].
Automated Debugging	21.76	38	2012	Automated debugging techniques [106].
Adaptive Random Testing	21.51	95	2012	Random testing techniques designed to ensure input is evenly spread over the input domain [107].
Data Flow Testing	21.47	43	2011	Testing based on the flow of information between variable definitions and usages [108].
Data Flow	21.42	36	2008	Metrics for tracking the flow of information [108]
Software Standards	20.92	26	2009	Constraints, rules, and requirements that software or testing is expected to meet [109].
Synchronization	20.07	29	2015	Practices for ensuring components are able to coordinate when completing tasks in parallel [110].
Sensitivity Analysis	20.00	36	2012	Study of how uncertainty in system output can be traced to sources of uncertainty in its inputs [111].

prioritization enables efficient test execution, and regression testing is a process performed as part of a test execution pipeline. Combinatorial testing suggests an important subset of test input to apply, often as part of automated generation. Models are often used to generate test input. Machine learning, including neural networks, supports prediction tasks related to automation.

Many of the remaining keywords relate to assessments of testing effectiveness or test creation guidance, including mutation testing, fault injection, and coverage criteria. Other keywords (software quality, reliability, verification, security) relate to the overall quality of the system, including its correctness, performance, and security. Finally, embedded soft-

ware and simulation relate to systems combining software and hardware elements, which have high safety demands and unique testing activities [82].

The average number of publications per keyword is 75 (0.15%)—far below the number of publications targeting the top 20 keywords, indicating their importance. It is interesting that the most popular keyword is only a target of 2.14% of the sample, and that only five keywords are targets of over 1% of the sample. We believe this is an indication of the breadth of testing research. There are many challenges associated with testing, from test creation to automation, execution, assessment, and process. There are many ways to address each challenge, including algorithms and tools, human-driven activities, and studies of those working in the field. Even the median—40 publications—is a reasonable body of work on any single concept.

We can compare the most-common keywords with the most-cited. In Table 3.3, we identify keywords that receive, on average, the most citations per publication.

RQ1 (Popular Keywords): The most-cited keywords also relate to automation, test creation and assessment guidance, and assessment of system quality.

Some of these keywords are linked to the most common keywords—fault-based testing, test suite minimization, covering arrays, partition testing, evolutionary testing, and regression test selection, in particular. However, no keywords appear in both lists.

However, both the most common and most-cited keywords share common themes. Many of the keywords relate to automation (e.g., test suite minimization, random testing), test creation and assessment (e.g., testing strategies, data flow testing), or quality assessment (e.g., software fault prediction, sensitivity analysis). For example, Figure 3.4 illustrates that many keywords associated with automated test generation receive a relatively high number of citations on average.

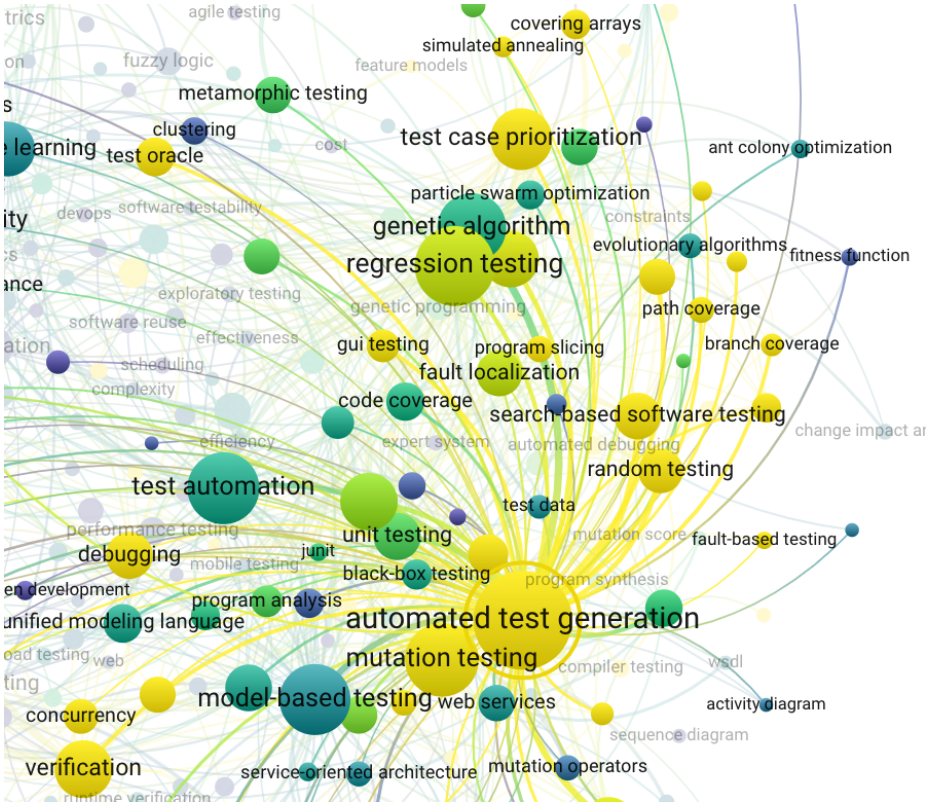


Figure 3.4: A subset of keywords connected to automated test generation, colored by the average number of citations. Nodes in yellow attract a high number of citations (≥ 14).

In general, the keywords in Table 3.3 are associated with a relatively small number of publications. They also have an older average date of publication, approximately 2010 versus 2014, allowing more time to attract citations. We hypothesize that these particular keywords (a) are related to themes that attract attention, and (b) are attached to a small set of publications containing a subset of particularly influential publications.

RQ2: Characterization of Clusters into Topics

By examining the connections between keywords, we can understand the context in which keywords form, grow, and thrive. Therefore, we have identified *research topics* characterizing the keyword clusters. These topics are detailed in Tables 3.4-3.5. We note a cluster ID assigned by VOSviewer, the number of keywords in that cluster, the density of connections between keywords within each cluster (the ratio of the number of existing within-cluster

connections to the total possible within-cluster connections, $\binom{\text{Keywords}}{2}$), and the topics and

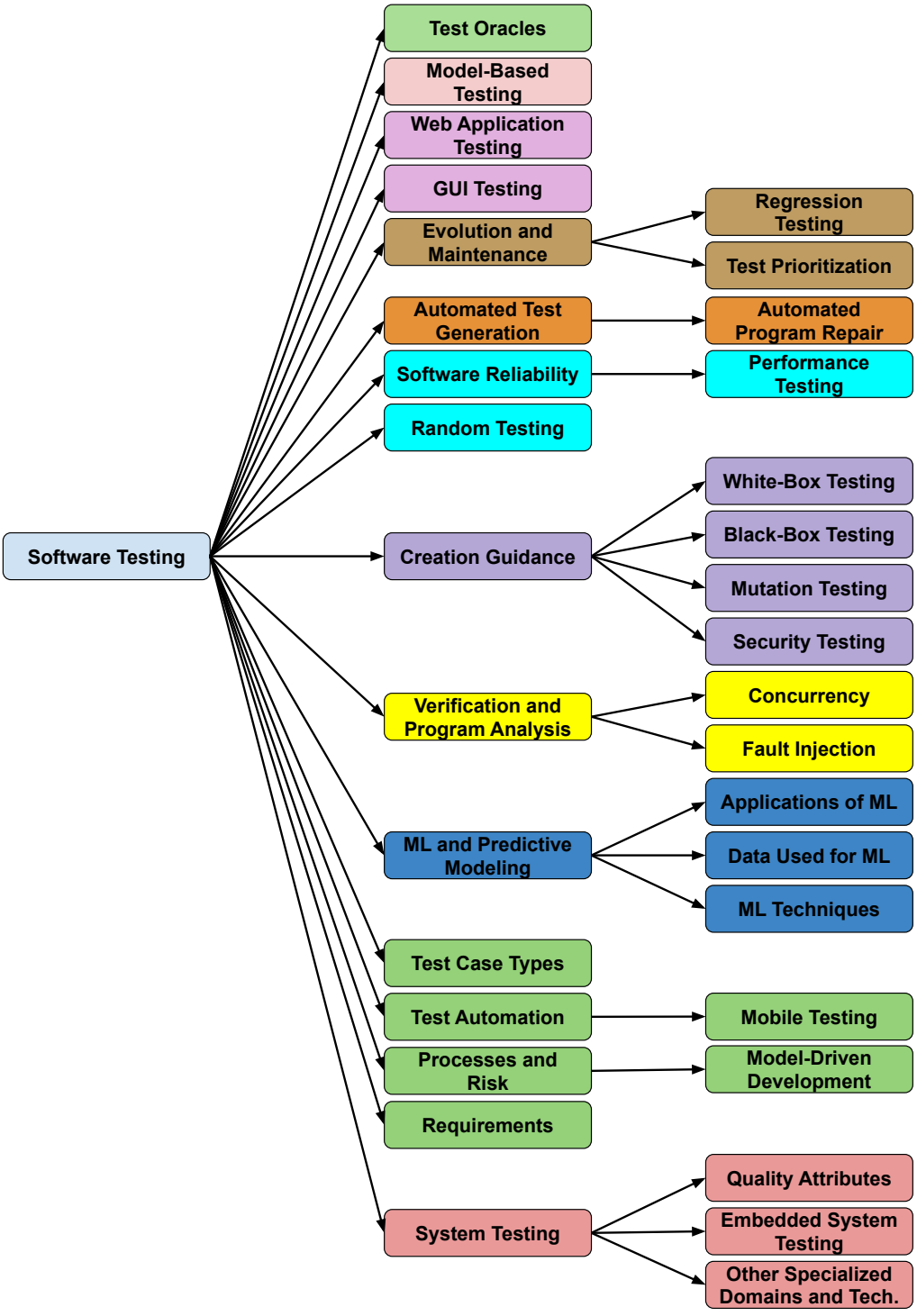


Figure 3.5: Identified research topics (middle layer) and subtopics (final layer), colored by cluster.

Table 3.4: An overview of clusters 4 - 11, including the cluster ID from VOSViewer, the number of keywords, inter-cluster connection density (percentage of possible connections between keywords), identified topics and subtopics, example keywords for each topic, and a brief description of each topic. Clusters are ordered from smallest to largest.

Cluster	Num Keywords	Density	Topics and *(Subtopics)*	Example Keywords	Description
11	4	100%	**Test Oracles**	test oracle, metamorphic relation	Test component that issues a verdict of correctness [2].
10	16	39%	**Model-Based Testing**	model transformation, timed automata	Use of behavioral models to analyze the system, to design or generate test cases, or as oracles [11].
9	18	31%	**Web Testing**	web applications, javascript	Testing techniques, tools, and activities focused on verification of web-based applications [112].
			GUI Testing	graphical user interface, finite state machine	Test design or generation techniques focused on exercising a system through its graphical interface [113].
8	19	47%	**Evolution and Maintenance**	program comprehension, change impact analysis	Practices for controlling and maintaining quality as the system changes over time [114].
			(*Regression Testing*)	regression testing, regression test selection	A practice where tests are re-executed when code changes to ensure that working code operates correctly [74].
			(*Test Prioritization*)	test case prioritization, test case selection	Automated techniques that select a subset of tests for execution [84].
7	28	48%	**Automated Test Generation**	genetic algorithms, branch coverage	The use of tools to generate full or partial test cases [11].
			(*Automated Program Repair*)	fault localization, genetic programming	Automated generation of patches for faulty programs [115].
6	30	24%	**Reliability**	reliability growth, quality control	Means to define, measure, and assess the how quality changes over time [83].
			(*Performance Testing*)	load testing, cloud testing	Testing to assess performance and scalability of a system under different operating conditions [116].
			Random Testing	adaptive random testing, statistical testing	Generation of random input for various purposes (e.g., assessing reliability or performance) [11]
5	32	30%	**Creation Guidance**	certification, test adequacy	Guidance for how a tester might approach test design—e.g., goals, input selection, and assessing test strength.
			(*White-Box Testing*)	coverage criteria, data flow	Test creation based on source code [85].
			(*Black-Box Testing*)	specification-based testing, black-box testing	Test creation based on requirements and other documentation [117].
			(*Mutation Testing*)	mutation score, mutation operators	Test creation based on synthetic faults seeded into a system [75]
			(*Security Testing*)	penetration testing, software vulnerability	Test creation to assess the ability of a system to prevent exploitation of vulnerabilities [118].
4	35	31%	**Verification and Analysis**	dynamic analysis, static analysis	Analyses performed to ensure that software possesses properties of interest (e.g., correctness, resilience) [1].
			(*Concurrency*)	parallelization, synchronization	Analyses of programs that execute over parallel threads or processes [119].
			(*Fault Injection*)	fault model, fault tolerance	Injection of faults into a system for analysis [78].

subtopics assigned to that cluster. For each topic or subtopic, we list two example keywords that fall within that topic, and we briefly describe the meaning of the topic. For additional clarity, Figure 3.5 outlines these topics, colored by the cluster they emerged from.

RQ2 (Characterization of Clusters into Topics): Based on keyword clustering, testing research can be divided into 16 topics, with a further 18 subtopics.

While some of the topics within a cluster may seem independent, they are linked by

Table 3.5: An overview of clusters 1 - 3, including the cluster ID from VOSViewer, the number of keywords, inter-cluster connection density (percentage of possible connections between keywords), identified topics and subtopics, example keywords for each topic, and a brief description of each topic. Clusters are ordered from smallest to largest.

Cluster	Num Keywords	Density	Topics and *(Subtopics)*	Example Keywords	Description
3	48	26%	**Machine Learning**	machine learning	Algorithms that make inferences from patterns detected in data [87].
			(*Applications*)	defect prediction, estimation	Applications of ML in software testing.
			(*Data Used*)	metrics, complexity	Sources of data used to draw conclusions with ML.
			(*ML Techniques*)	neural networks, deep learning	ML techniques used in testing research.
2	58	21%	**Test Case Types**	unit testing, exploratory testing	Practices and levels of granularity for test design.
			Test Automation	test execution, testing tools	Tools and practices that enable automation of test execution [76].
			(*Mobile Testing*)	mobile testing, android testing	Testing techniques, tools, and activities focused on verification of mobile applications [120].
			Processes and Risk	software quality, test-driven development	The organization, management, and testing process of a development team [1].
			(*Model-Driven Development*)	model-driven development, model-driven testing	Development process based on use of models for analysis, code generation, and testing [121]
			Requirements Engineering	requirements engineering, traceability	Requirements indicate correct behavior. Verification often assesses conformance of code to requirements [1].
1	118	20%	**System Testing**	system testing, user interfaces	Test cases that interact with an external system interface [1].
			(*Quality Attributes*)	usability, software performance	Non-functional properties of a system assessed as part of quality assurance [122]
			(*Embedded Systems*)	real-time system, simulation	Complex self-contained hardware and software systems [90].
			(*Other Specialized Domains*)	open source software, image processing, autonomous vehicles	System types (e.g., databases, virtual reality, operating systems) or technologies (e.g., XML, Java) with dedicated testing approaches.

connections between the underlying keywords. It is important, therefore, to examine both topics and keywords to come to a full understanding of a particular cluster. For example, random testing is a topic with widespread applicability. However, it is linked to Cluster 6 because random testing is often used to assess reliability or performance.

Within Cluster 2, the test automation topic encapsulates the emerging subtopic of mobile testing. Mobile testing is not as well-established as web application testing, but is clearly growing as a distinct research area. In the future, it may emerge as an independent research topic—perhaps even as its own cluster. Additionally, the model-driven development subtopic in Cluster 2 is related to—but also separate from—the model-based testing topic in Cluster 10. The latter focuses on technical aspects of modelling, while the former focuses on process and practices that may use these technologies. There are connections between the two, but they contain different keywords.

Table 3.6: Connection density between pairs of clusters. Cross-cluster densities ≥ 0.12 are highlighted. Densities in italics represent within-cluster densities for each cluster.

Cluster	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	*0.20*	0.08	0.09	0.11	0.06	0.06	0.06	0.07	0.07	0.09	0.07
2		*0.21*	0.08	0.09	0.10	0.08	0.09	**0.13**	0.10	**0.14**	0.11
3			*0.26*	0.10	0.08	0.08	**0.12**	**0.13**	0.05	0.07	**0.15**
4				*0.31*	**0.15**	0.08	**0.12**	**0.12**	0.07	0.10	**0.12**
5					*0.30*	0.08	**0.20**	**0.15**	**0.13**	**0.14**	**0.21**
6						*0.24*	0.10	0.11	0.06	0.06	**0.12**
7							*0.48*	**0.23**	**0.14**	**0.14**	**0.15**
8								*0.47*	**0.13**	**0.15**	**0.20**
9									*0.31*	0.10	**0.17**
10										*0.39*	0.09
11											*1.00*

Cluster 1 is the least cohesive cluster. However, we can categorize many keywords under a core topic of system testing. In Cluster 2, there is a topic centered around test case types (e.g., unit testing). System testing is often grouped with these test case types. However, it is also a broader concept encompassing many different types of systems and system interfaces (e.g., embedded systems, operating systems, or databases). Several topics in our characterization also relate to system-level practices or domains, e.g., web, GUI, and performance testing. Those topics are established enough to stand independently, while the system testing topic in Cluster 1 acts as a broad umbrella.

RQ3: Connections Across Clusters

We analyze connections *across* clusters by measuring the connection density between pairs of clusters, and by identifying keywords that bridge clusters.

Connection Density: Table 3.6 shows all cross-cluster densities, with those $\geq 12\%$ are highlighted. 23 of the 45 pairs of clusters meet this threshold, indicating that many research topics are densely connected.

RQ3 (Connections Across Clusters): Clusters 5 (creation guidance), 7 (automated test generation), 8 (evolution and maintenance), and 11 (test oracles) are densely

connected to several clusters. These clusters represent particularly multidisciplinary topics.

Cluster 8 appears in eight pairs, Clusters 7 and 11 appear in seven, and Cluster 5 appears in six. In particular, the pairings between Clusters 7 and 8 and between Clusters 5 and 11 have a higher connection density than the within-cluster densities of Clusters 1 and 2, indicating the dense interconnection between these topics.

As the most common keyword identified in our analysis, automated test generation has connections to keywords and topics in every other cluster. If a testing method exists, there will be an interest in generating tests for it (e.g., Clusters 5, 8, 9, and 10). Oracle creation often requires manual effort, leading to an interest in automated generation or reuse of oracles (Cluster 11). Further, machine learning offers the means to assist or enable automated test generation (Cluster 3).

Test oracles are a necessary component of almost all test cases, leading to dense connections with Clusters 5 (creation guidance), 6 (reliability, performance, random testing), 8 (regression testing), and 9 (web and GUI testing). In addition to the above-mentioned connection to automated generation, machine learning offers a means to automate the creation of oracles (Cluster 3). Oracles are also a natural part of verification and different program analyses (Cluster 4).

Maintenance has implications on multiple aspects of testing, such as costs and quality. Maintenance needs affect the tasks performed during test automation (Cluster 2). Test prioritization also uses the same information that guides test creation to select tests (Cluster 5), and can be assisted using machine learning (Cluster 3). Both regression testing and test prioritization are performed for GUIs and web applications (Cluster 9), and can make use of models (Cluster 10). Further, analyses related to program and test evolution are often connected to other analyses in Cluster 4.

Test creation practices (Cluster 5) also connect broadly. Beyond automated test gener-

ation, test oracles, and test prioritization, several creation practices either have adaptations for model-based testing (Cluster 10) or for web and GUI testing (Cluster 9). In addition, there are connections between verification and test creation practices (Cluster 4)—e.g., black box testing and verification are connected through specifications, and security testing and analysis are related.

Clusters 1, 2, 3, and 6 have the least-dense connections to other clusters. Clusters 1 and 2 are both large clusters with multiple topics and subtopics that are distinct, but closely-related. Connections exist to other clusters, but may be less common, as these two clusters already represent a broad set of keywords. Reliability and performance testing (Cluster 6) and various forms of predictive modelling in Cluster 3 are also often pursued as standalone topics, but can be connected to other topics. Out of all density measurements, the lowest was between Cluster 3 (machine learning) and Cluster 9 (web and GUI testing), with 5% of possible connections existing in publications.

Connecting Keywords: In Table 3.7, we list all keywords that are connected to at least 100 keywords in external clusters.

RQ3 (Connections Across Clusters): Twenty keywords serve as "connectors" between clusters, reflecting multidisciplinary concepts (e.g., software quality), common test activities (e.g., unit testing), and common sources of information for test creation (e.g., coverage criteria).

For comparison, we also list the total number of connected keywords, and the position that the keyword had in Table 3.2 (if it appeared in the most commonly-targeted keywords). Many of the connecting keywords are also among the most common occurring keywords, with automated test generation on top of both lists. The exact positions of keywords shift in the ordering, but 14 of the 20 most common keywords are also connecting keywords. The most common keywords tended to relate to automation, test creation and assessment

Table 3.7: Keywords that are connected to at ≥ 100 keywords in clusters other than the one where the keyword is assigned (both keywords are targeted in at least one study). Each keyword is named and described, and the number of connected keywords (in external clusters, and in total) are listed.

Keyword	**Connected (External)**	**Connected (All)**	**Position in Table 1**	**Description**
Automated Test Generation	217	243	1	See Table 3.2.
Software Quality	164	202	8	See Table 3.2.
Mutation Testing	164	184	3	See Table 3.2.
Regression Testing	157	174	2	See Table 3.2.
Test Automation	152	200	4	See Table 3.2.
Model-based Testing	150	163	5	See Table 3.2.
Coverage Criteria	147	168	13	See Table 3.2.
Verification	143	164	12	See Table 3.2.
Genetic Algorithm	139	161	6	See Table 3.2.
Machine Learning	132	161	15	See Table 3.2.
Test Case Prioritization	119	134	11	See Table 3.2.
Software Maintenance	119	130	-	Practices for controlling and maintaining quality as the system changes over time [114].
Debugging	117	135	-	See Table 3.3.
Unit Testing	114	136	-	A practice where tests are created for a small, isolated unit of code (typically a class) [1].
Software Reliability	114	132	10	See Table 3.2.
Reliability	108	125	16	See Table 3.2.
Fault Injection	106	128	7	See Table 3.2.
Static Analysis	101	120	-	Analyses performed without executing the code (e.g., inspection or symbolic execution) [1].
Mutation Analysis	101	116	-	Analyses of programs or tests performed using injected mutations [27].
Unified Modeling Language	101	111	-	A family of techniques for modelling and analyzing program behavior [101].

guidance, assessment of system quality, and cyber-physical systems. These concepts—especially the first three—are broad, with wide-ranging applicability. That suggests that popularity of a keyword is not only a reflection of a particular concept, but on its multidisciplinary applicability.

In contrast to Table 1, we see a notable rise in the position of software quality, coverage criteria, and machine learning. Software quality and machine learning are both very broad concepts, while coverage criteria are a common source of information and a target for testing, with applications in test creation guidance, automated test generation, quality assessment, prediction, and other areas.

We also see several keywords emerge: software maintenance, debugging, unit testing, static analysis, mutation analysis, and unified modeling language. These include broadly

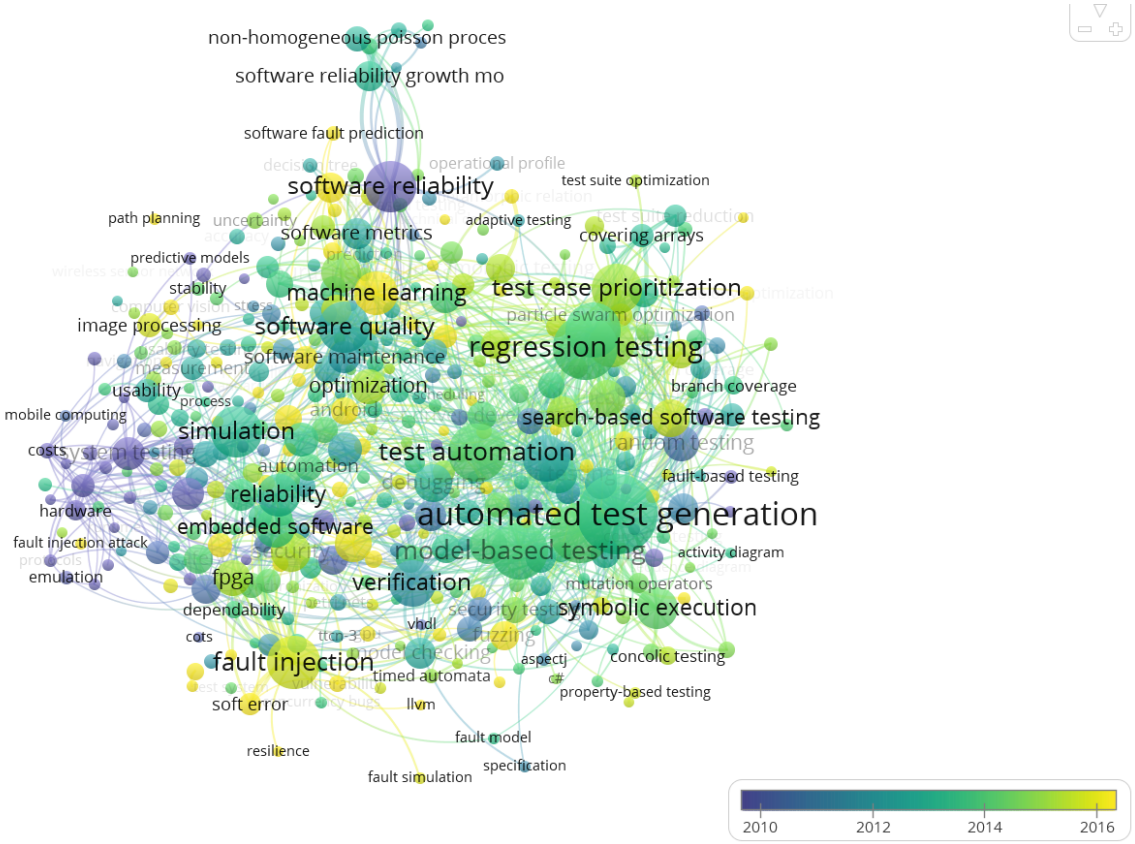


Figure 3.6: The map of keywords, colored by the average year of publication. Note that "2010" should be read as $\leq$ 2010 and "2016" should be read as $\geq$ 2016.

applicable concepts (maintenance, debugging, static analysis, mutation analysis), a common source of information (unified modeling language), and a common testing activity (unit testing).

Six of the most common keywords do not meet the threshold for connecting keywords—simulation (93 external connections), combinatorial testing (96), symbolic execution (83), embedded software (85), neural networks (83), and security (82). All six are multidisciplinary concepts, but are more specific—rather than broad—concepts (combinatorial testing, symbolic execution, embedded software, neural networks).

RQ4: Emerging Keywords, Topics, and Connections

A visualization of the map of keywords, colored by average year of publication, is shown in Figure 3.6. Yellow nodes have an average date of 2016 or newer. Blue nodes have an average date of 2010 or earlier. A gradient between blue and yellow represents 2010–2016. We examine keywords, topics, and connections that have emerged or grown in interest since June 2015.

> An interactive version of this map can be accessed as an overlay at `https://greg4cr.github.io/other/2021-TestingTrends/topics.html` by selecting "Avg. Pub. Year" under the "Color" option.

Keywords and topics: Sixty-six keywords (16.26%) represent new emerging concepts or have received significant recent attention. Figure 3.7 links these keywords to their respective research topic. From these results, we can make several observations:

- Many of the growth areas map to shifts in technology. There is growing interest in web applications, relating to technologies (JavaScript), testing tools (Selenium), and testing techniques. There is a similar emergence of mobile applications, in both the subtopic of mobile testing in Cluster 2 (android testing, mobile testing) and technologies in Cluster 1 (mobile applications, smartphone).
- Machine learning has advanced many fields. Unsurprisingly, it is also one of the largest growth areas in testing. The keyword "machine learning" has an average publication date of October 2016, and keywords have emerged related to applications, data, and specific techniques for ML. "Deep learning" is one of the newest keywords (average date of September 2018).
- Keywords have also emerged targeting ML and AI-based systems. From the embedded systems and "other domains" topics, we see keywords related to autonomous

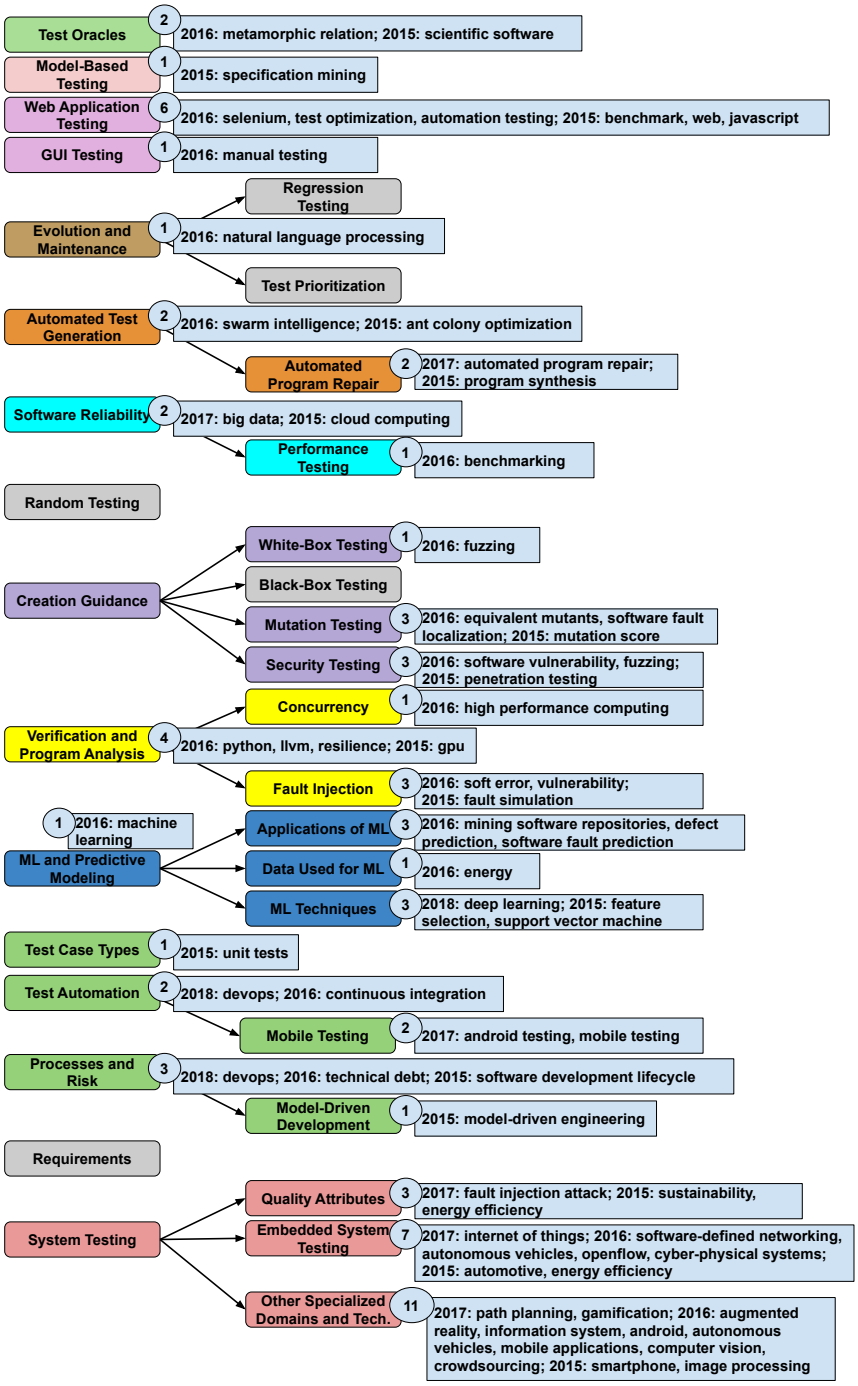


Figure 3.7: Keywords with an average publication date **newer than June 2015**, along with their associated research topic. The number next to the list of keywords indicates the number of emerging keywords. Topics colored in gray are those without emerging keywords.

vehicles, computer vision, image processing, and augmented reality. All of these areas require specialized testing approaches. Autonomous vehicles, in particular, may

grow into its own independent subtopic in the future.

- There is growing interest in energy consumption. This is connected to mobile applications, and a shift to portable devices that rely on batteries. This also reflects growing interest in sustainability and environmental impact of software.
- Automated program repair has emerged as a subtopic. The core keyword has one of the newest average publication dates (March 2017), and its connected keywords (e.g., program synthesis) also have recent dates.
- Fuzzing and search-based approaches (swarm intelligence, ant colony optimization) have emerged as test generation techniques. Fuzzing, notably, has seen application in general and security-focused testing topics. Security-related keywords are also active and growing.

RQ4 (Emerging Keywords, Topics, and Connections): Emerging keywords and topics relate to, or incorporate, web and mobile applications, machine learning and AI—including autonomous vehicles—energy consumption, automated program repair, or fuzzing and search-based test generation.

Connections: We focus our examination on ten pairs of clusters with the highest proportion of emerging connections to the number of possible connections ($\geq 3.5\%$). The connected clusters, and their associated topics, have a rapidly evolving relationship.

- Cluster 11 (test oracles) with Clusters 5 (creation guidance; 8.59% of connections are emerging), 3 (machine learning; 6.77%), 8 (evolution and maintenance; 5.26%), 6 (reliability; 4.17%), 9 (web application and GUI testing; 4.17%), 2 (test case types, test automation, processes and risk, and model-driven development; 3.88%), and 4 (verification and program analysis; 3.57%).
- Cluster 7 (automated test generation) with Clusters 9 (4.36%) and 3 (3.57%).

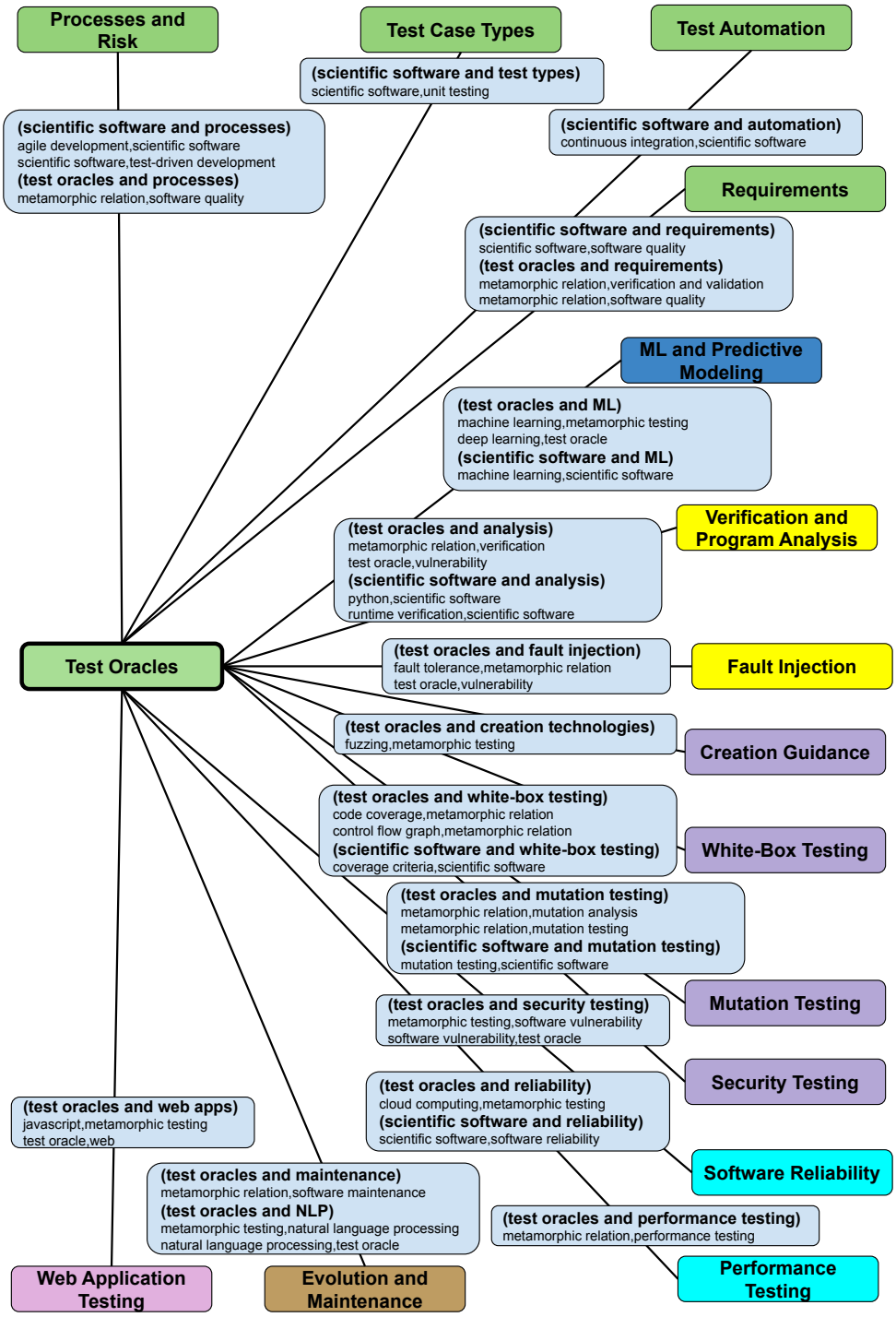


Figure 3.8: Emerging connections, connected by research topic with test oracles, for the cluster pairings with highest ratio of emerging to total connections.

- Cluster 5 with Cluster 9 (4.69%).

The highlighted connections between topics are shown in Figure 3.8 for topics connected

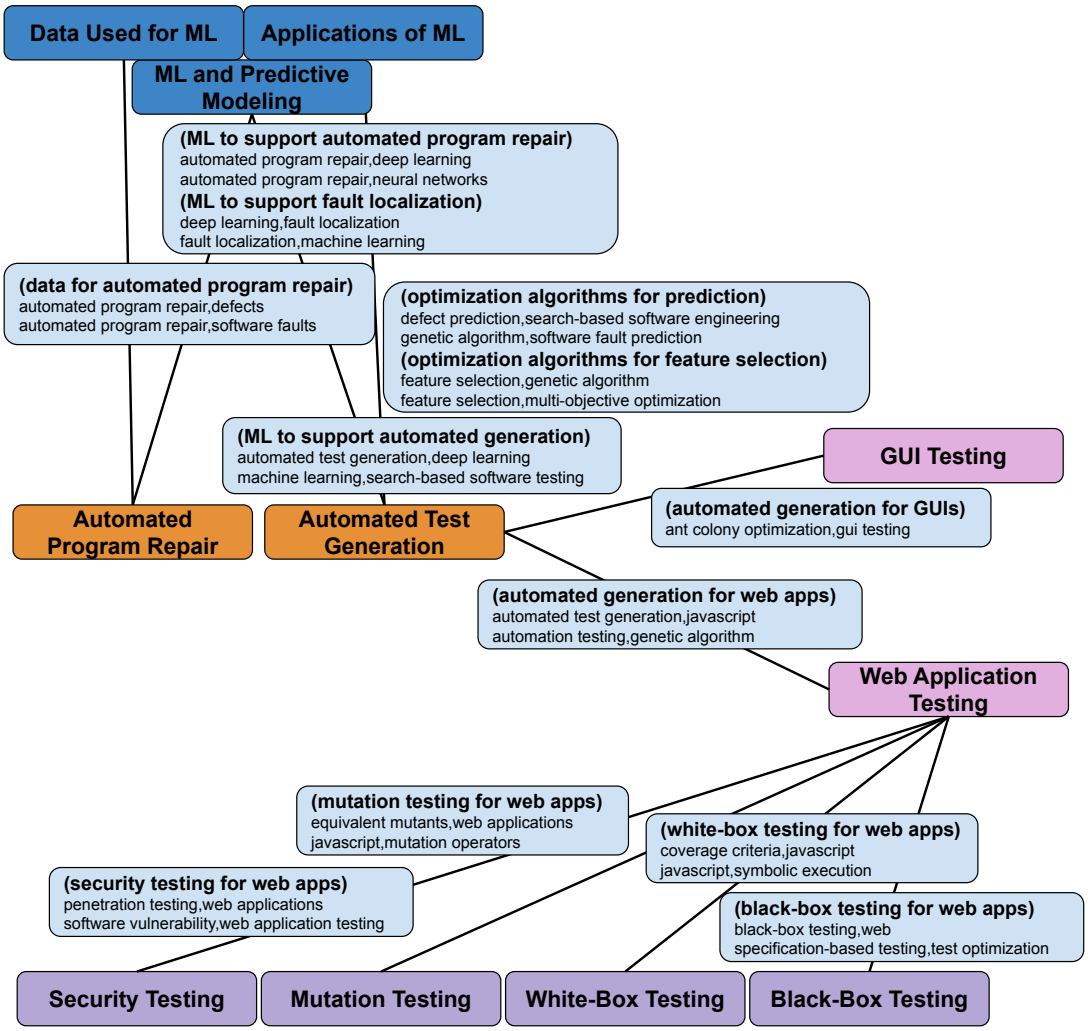


Figure 3.9: Emerging connections, connected by research topic (excluding test oracles), for the cluster pairings with highest ratio of emerging to total connections.

with test oracles, and in Figure 3.9 for other topics. For each connection between topics, a small number of example connections between keywords are shown.

RQ4 (Emerging Keywords, Topics, and Connections): Web applications and scientific computing require targeted testing approaches and practices, leading to emerging connections to many topics. Test oracles are also a rapidly-evolving topic with many emerging connections. Machine learning has emerging potential to support automation.

We make several observations about these emerging connections:

- Test oracles appear often because (a) Cluster 11 is a small cluster, (b) this topic has the largest percentage of emerging keywords, and (c), this topic is naturally connected to all other topics. Research interest in test oracles is growing [2, 123], and effective oracles are needed for emerging domains such as web applications. The relationship between oracles and different testing practices is not well understood yet, leading to many emerging connections. Further, interest is growing in the use of machine learning to generate test oracles [123].

- The keyword "scientific computing" is part of Cluster 11, due to its frequent connection with metamorphic testing. Inspection of the emerging connections makes it clear that software testing for scientific computing is emerging as a distinct domain of interest, with major connections to Cluster 2 and 5.

- As in many other areas of software development, machine learning offers the potential to automate tasks that traditionally require significant human effort, such as test and oracle generation and program repair.

- Test creation practices of many types (including white-box, black-box, mutation, and security) are emerging for web applications.

- New test generation approaches are emerging for GUIs and web applications.

RQ5: Declining Keywords and Topics

Figure 3.10 shows the 66 keywords with the oldest average date, with their associated research topic. In particular, we highlight three research topics or subtopics that we hypothesize may currently be in decline.

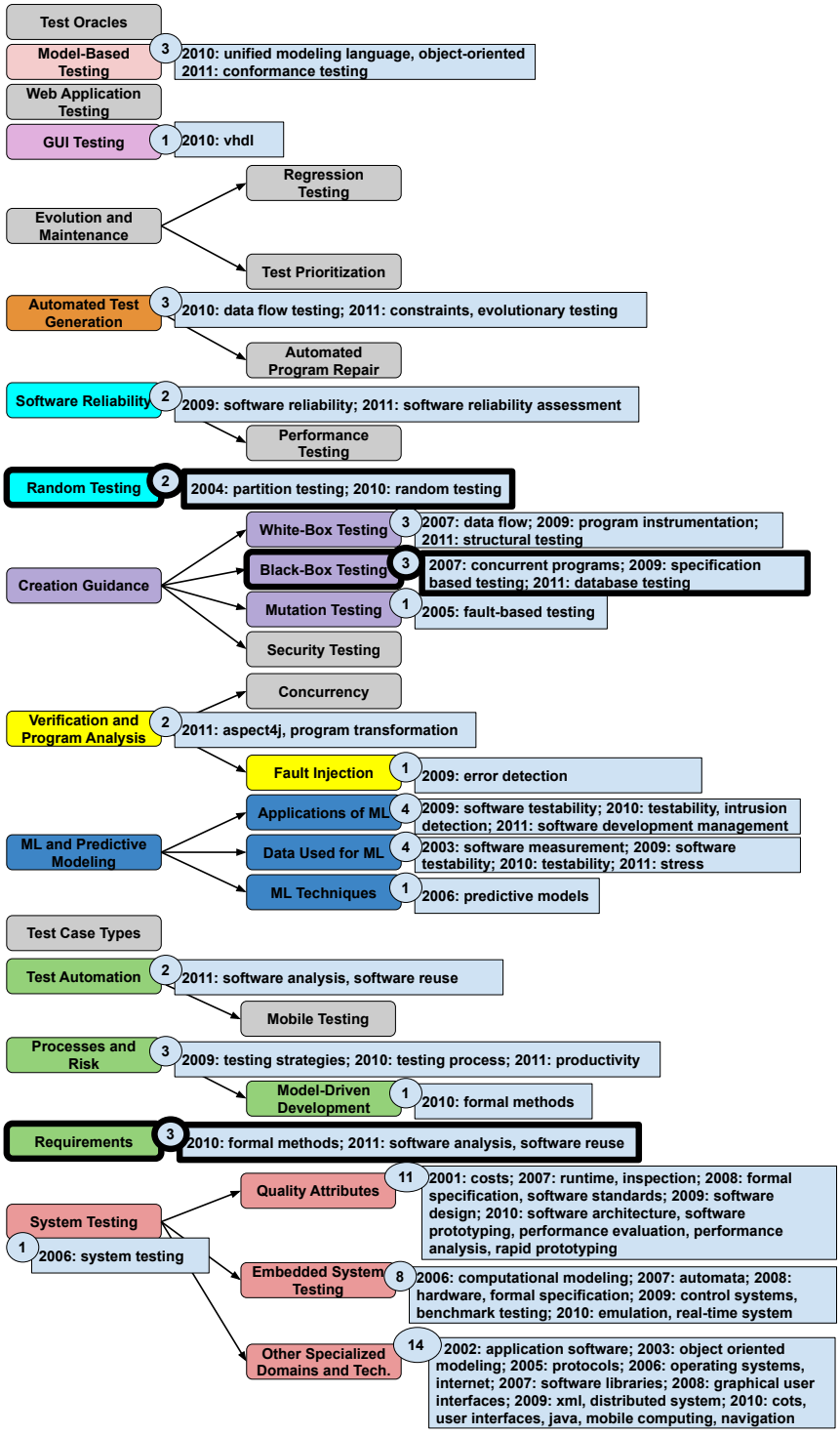


Figure 3.10: Keywords with an average publication date **earlier than June 2011**, along with their associated research topic. Topics colored in gray are those without declining keywords. Topics with both declining keywords and a lack of emerging keywords are highlighted.

RQ5 (Declining Keywords and Topics): Older average dates of publication and lack of emerging keywords suggest that keywords and topics related to random and requirements-based testing may be in decline.

Briefly, we examine these areas:

- Traditional random testing has been supplanted, to some extent, by semi-random approaches. As shown in Figure 3.7, search-based and fuzzing techniques are growing in popularity. Both use sampling heuristics instead of applying pure random generation, retaining some of the benefits of random testing (e.g., scalability) while potentially yielding more effective results.
- Many of the keywords related to requirements and black-box testing have older average publication dates, indicating potential stagnation. Agile processes favor lightweight requirements (e.g., user stories) over formal and complex requirements. We hypothesize that this may have led to a shift in attention towards other sources of information for test creation.

We hesitate to state that these topics are "dying" or are solved challenges. However, we do see evidence that they have not seen notable growth in popularity or the emergence of new keywords in recent years. New application areas, techniques, or changes in development processes may lead to a resurgence in interest in the future.

3.5 Further Analysis and Advice to Researchers

Both the high-level topic overview and the low-level map of connections between keywords can serve as inspiration for prospective and experienced researchers. We offer the following advice on how this data could inspire new research.

An overview of the testing field: For inexperienced researchers, the high-level topics offer an immediate "snapshot" that can be used to guide exploration of different research areas. The keywords illustrate key concepts that form research topics, and offer targeted suggestions on terms the researcher should examine in detail. Connections between those keywords illustrate how those concepts have been connected in practice, which may encourage critical reflection on both the individual concepts and how they relate. The emerging keywords and topics suggest areas that researchers may wish to pay attention to, and emerging connections clarify how these keywords fit into the field.

Understanding the context of a keyword or topic: Researchers can analyze the map to gain a data-driven view of the field for further planing and development. As a starting point, those interested in a keyword or topic can examine how that keyword or topic fits into the broader context of testing research.

- What keywords are often associated with a keyword of interest? This may illustrate the type of research often conducted on this concept (or its associated topic), and natural areas of synergy between keywords or topics.

- Is interest in this keyword or topic growing, declining, or stable? The average date of publication may suggest the current level of interest (or lack thereof).

Identification of under-explored connections between keywords or topics: We hypothesize that the map data could potentially inspire future research through analyses of connections between keywords and topics. There are many ways connections could be analyzed. One is to identify keywords that have *under-explored* connections.

Specifically, an under-explored connection is one where (a) at least one publication has connected the keywords, but (b), the specific number of publications connecting those two keywords is relatively low—indicating potential for additional research exploration. Under-explored connections may serve as inspiration, suggesting concepts that could be connected in further research:

- Within a cluster, under-explored connections may suggest ways that concepts within a particular research topic could be more closely linked. For example, different mechanisms from automated test generation algorithms could be blended into hybrid algorithms.[5] An examination of under-explored connections could offer similar inspiration.

- Across clusters, we could identify either pairs of clusters or topics that could be more deeply connected in future research. In some cases, these may be topics that are already connected (e.g., automated test generation and white-box testing), but where there are opportunities for new research related to specific keywords or aspects of the topic (e.g., specific test generation algorithms).

There are different ways that under-explored connections could be identified and analyzed. As an initial exploration, first, one must identify a lower and upper bound on the number of publications linking keywords. As an example, in the network visualization, four publications are needed for an edge to be shown (by default). Therefore, one could adopt four publications as the threshold for this analysis and capture all connections in a short range of this threshold—e.g., 4-6 publications targeting a pair of keywords.

718 connections have a strength of 4-6 publications. To identify a subset for initial exploration, we can (a) focus on cross-cluster connections, and (b), use the cross-cluster connection density to identify the pairs of clusters with the most under-explored connections. Here, we specifically focus on the cluster pairings where $\geq 2\%$ of all connections between the two clusters consist of under-explored connections. Six cluster pairings met the threshold: Cluster 11 (test oracles) with Cluster 9 (web and GUI testing, 2.63%), Cluster 5 (creation guidance, 2.34%), and Cluster 3 (machine learning, 2.08%), and Cluster 7 (automated test generation) with Cluster 8 (evolution and maintenance, 3.38%), Cluster 10 (model-based testing, 2.46%), and Cluster 5 (2.01%).

[5] As has been done for concolic execution and search-based test generation [124].

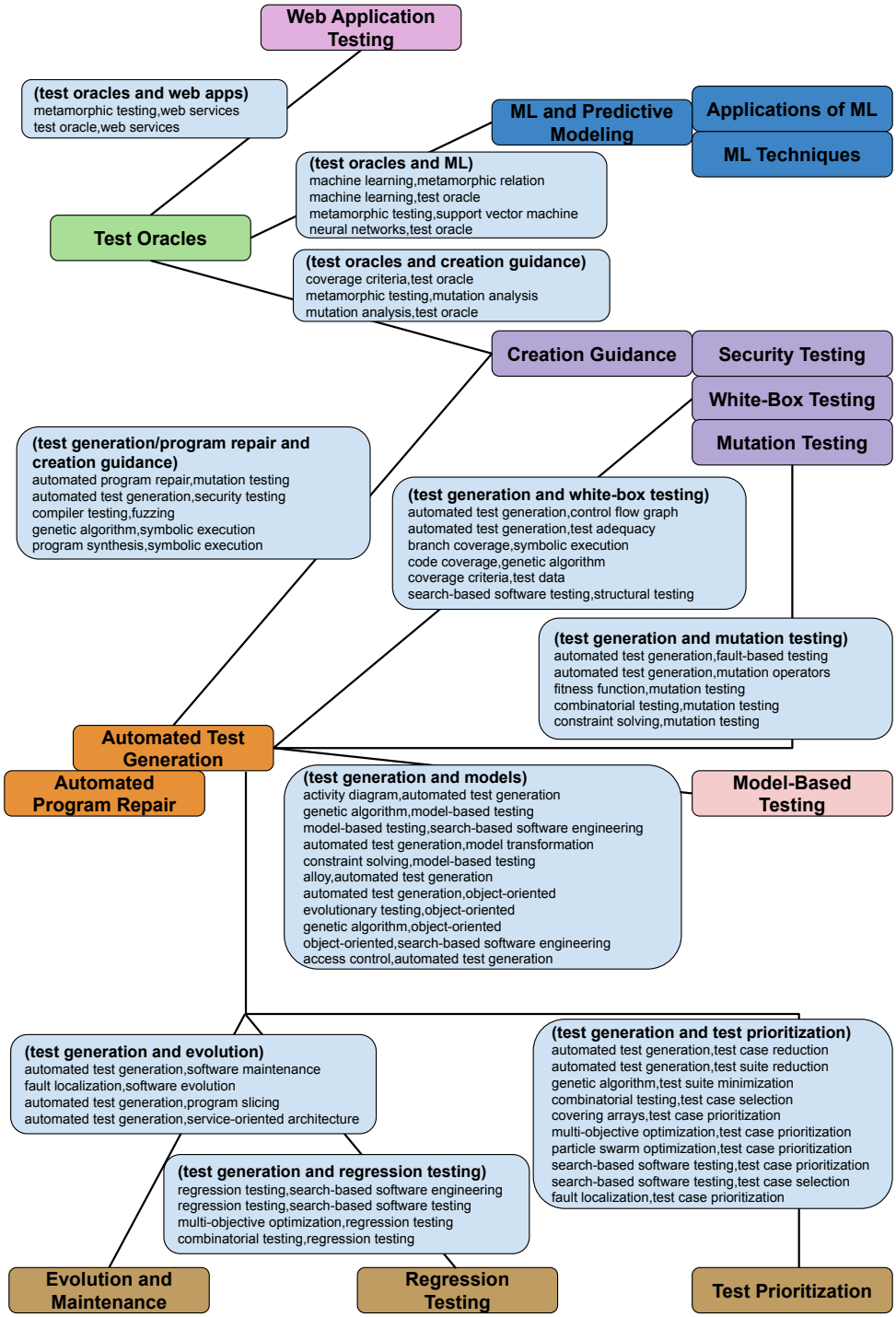


Figure 3.11: Under-explored connections (keywords connected by 4-6 publications), connected by research topic, for the six cluster pairings with highest ratio of under-explored to total connections.

For these cluster pairings, we grouped the connections by research topic, then examined the meaning and potential application of the connections. In Figure 3.11, we illustrate the identified connections, categorized by their associated research topics.

We make several observations about these connections. First, specific suggestions emerge for exploring connections in future research, including (among others):

- The relationship between mutations and test oracles.
- Use of mutation as part of automation program repair and test generation.
- Test generation based on specific modeling formats (e.g., object-oriented models such as activity diagrams).
- Reduction techniques for generated test suites and test cases.
- The relationship between test generation and program evolution (e.g., how often tests should be generated, how tests should be maintained).
- Generation of tests for regression testing.
- The use of specific optimization algorithms for test case prioritization.

In some cases, "under-explored" coincides with "emerging"—for example, test oracles with machine learning and web services. There are also cases where topics are well-connected in research (e.g., test generation and white-box testing) through different keywords (e.g., "coverage criteria" instead of "code coverage"). We retained keywords with minor differences in meaning, as even minor distinctions may be important. However, some connections may be well-explored under a different keyword. Even in such cases, there may be opportunity for further exploration related to these keyword differences, or connections based on concepts and technologies than have not been explored previously (e.g., specific generation algorithms or coverage criteria).

Identifying new connections between keywords: The *absence* of a connection between two keywords does not imply that the concepts cannot be connected. Consider keywords within a single cluster. Keywords lacking a direct connection may represent entirely incompatible concepts. However, in other cases, there may be a natural synergy between the two concepts that had not yet been considered. While the map cannot directly inform researchers which keywords *can* be connected, or how they can be connected, it can serve as a means to prompt brainstorming.

As an example, we can inspect keywords within a cluster that lack a direct connection to specific other keywords in their cluster. Cluster 8 (evolution and maintenance, with subtopics of regression testing and test prioritization) contains 19 keywords. There are 180 cases where two keywords lack a direct link within Cluster 8—e.g., "change impact analysis" and "test case reduction" are not directly connected in publications.

Not all of these cases offer obvious ideas for new research, but consideration of these cases may lead to inspiration. For example, we identified the following ideas:

- The use of change impact analysis as part of program comprehension, test case reduction, test suite minimization, or test suite reduction.
- The use of information retrieval and natural language processing to provide information for test case and suite reduction, selection, and minimization and for regression test selection.
- The use of regression test selection techniques for use as part of test case and suite reduction and test case selection.
- The use of program comprehension techniques for regression test selection.
- The relationship between evolution and maintenance of software with test case prioritization, minimization, and reduction.

- Service-oriented architecture and web services appear in this cluster because of close association with particular keywords (e.g., regression testing), but are only indirectly connected to the majority of the other keywords. The missing connections suggest the need for targeted test case prioritization, selection, reduction, and minimization approaches for service-oriented architectures and web services, as well as examination of the evolution and maintenance of service-oriented architectures and web services.

Similar ideas may emerge from inspecting missing connections within other clusters.

There are many ways that this map could potentially be analyzed beyond the simple exploration in this section. We suggest that researchers attempt to analyze different connection types, connection strength thresholds, and other aspects of the collected metadata (e.g., publication age or number of citations) in order to gain inspiration for new research or insight into the field.

3.6 THREATS TO VALIDITY

Conclusion Validity: VOSviewer was used to perform visualization. The design of this tool and the visualizations it produces could potentially bias the observations made. However, the tool is based on well-understood and established computational principles. Further, it has been used in over 500 bibliometric studies (e.g., [58, 61]), in a large variety of fields and its assumptions have been verified by experts in these fields.[6] We have made efforts to verify the assumptions behind the analyses performed.

External Validity: Our study examined publications from the Scopus database, potentially omitting relevant venues for software testing research. Scopus is the one of the most comprehensive databases covering research publications [125], indexing content from 24,600

[6] A full list of publications is maintained at `https://www.vosviewer.com/publications`.

active conferences or journals and 5,000 publishers.[7] Specifically, Scopus coverage for computer science research has been found to be better than other databases [126]. Scopus also enables efficient export of the data we use to perform our mapping. Although some venues may not be indexed, many of the most important journals and conferences in the software testing field are included.

We used a single search string to build our sample. Other search strings (e.g., "software test") could have complemented the search process. However, our goal is not to capture all studies ever published in software testing. Rather, we require a sufficiently representative sample. We hypothesize that the additional value would be minimal compared to the filtering effort required. We believe that our sample of 57,233 publications is sufficiently large and representative to perform this analysis.

Internal Validity: We based our analysis on publications retrieved using the term "software testing". This pool of papers included publications unrelated to software testing, e.g., the use of software to test hardware or as part of student examination. We performed a manual process to remove unrelated keywords from the mapping. However, it is possible that some publications remain that are unrelated to the targeted research field. We believe that these are not enough to influence our observations.

Our analysis is based on author-supplied keywords, and not other sources of topic information, e.g., titles or abstracts. The use of keywords introduces a risk that publications are mislabeled (e.g., authors used the wrong term), or that important concepts are omitted. Still, author-supplied keywords are a clear and appropriate means to capture the structure of software testing research. Author-supplied keywords are regularly used in other bibliometric analyses [60, 61, 127] and have been found to effectively reflect structures in research fields [60, 128]. Even if relevant keywords are omitted, the concepts the authors felt were most important are reflected. While there is potential inaccuracy, it is likely that the selected keywords are close to correct. Alternative methods carry similar risks. Auto-

[7] List of covered journals and conferences: `https://www.scopus.com/sources.uri`.

mated or external categorization can also be inaccurate and potentially violates the intent of authors. Other sources of information, such as titles or abstracts, introduce noise and are difficult to use to categorize publications.

We applied a threshold of a minimum of 20 studies before a keyword appeared in our dataset or map. We used this threshold to omit minor or highly obscure keywords and to control the level of noise in the map. This risks also omitting emerging keywords. We tried lower and higher thresholds then we concluded that the current threshold is enough to cover terms with lower frequency and provide a meaningful and lower scatter network of the keywords. It should be noted when we tested lower and higher thresholds, the overall patterns did not change significantly.

3.7 CONCLUSION

Testing is the primary means of assessing software correctness and quality. Research in software testing is growing and rapidly-evolving. Based on the keywords assigned to publications, we seek to identify predominant research topics and understand how they are connected and have evolved.

We have applied co-word analysis to characterize the topology of software testing research over four decades of research publications. In this map, nodes represent keywords, while edges indicate that publications have co-targeted keywords. Nodes are clustered based on density and strength of edges. We examined the most common keywords, summarized clusters into research topics, examined how clusters connect, and identified emerging and declining keywords, topics, and connections.

We found that the most popular keywords tend to relate to automation, test creation and assessment guidance, assessment of system quality, and cyber-physical systems. The clusters of keywords suggest that software testing research can be divided into 16 core topics. All topics are connected, but creation guidance, automated test generation, evolution and

maintenance, and test oracles have particularly strong connections to other topics, highlighting their multidisciplinary nature. Emerging keywords and topics relate to web and mobile applications, machine learning, energy consumption, automated program repair and test generation, while emerging connections have formed between web applications, test oracles, and machine learning with many topics. Random and requirements-based testing show evidence of decline.

These insights and the underlying map can inspire researchers in software testing by clarifying concepts and their relationships, or by facilitating analyses of the field (e.g., through identification of under-explored and missing connections). In future work, we will broaden the type and scope of analyses of this map data, and we make our data available so that others can do so as well.

3.8 VOSViewer Technical Details

VOSviewer produces maps based on a co-occurrence matrix—a two-dimensional matrix where each column and row represents an item—a keyword, in our case–and each cell indicates the number of times two keywords co-occur. This map construction consists of three steps. In the first step, a similarity matrix is created from the co-occurrence matrix. A map is then formed by applying the VOS mapping technique to the similarity matrix. Finally, the map is translated, rotated, and reflected.

Forming the similarity matrix: VOSviewer takes as input a similarity matrix. This similarity matrix is obtained from the co-occurrence matrix through normalization. Normalization is done by correcting the matrix for differences in the total number of occurrences or co-occurrences of keywords. VOSviewer uses the association strength as its similarity measure [73]—in this case, the number of publications where two keywords are targeted together. Using the association strength, the similarity $s_{i,j}$ between two keywords i and j

is calculated as:

$$s_{i,j} = \frac{2mc_{i,j}}{w_i w_j} \tag{3.2}$$

where m represents the total weight of all edges in the network (the total number of co-occurrences of all keywords), $c_{i,j}$ denotes the weight of the edge between keywords i and j (the total number of co-occurrences of the keywords), and w_i and w_j denote the total weight of all edges of keywords i or j (the total number of occurrences of keywords i or j and the total number of co-occurrences of these keywords with all keywords that they co-occur with). Specifically:

$$w_i = \sum_j c_{i,j} \tag{3.3}$$

$$m = \frac{1}{2} \sum_i w_i \tag{3.4}$$

The similarity between keywords i and j calculated using Equation 3.2 is proportional to the ratio between the observed number of co-occurrences of keywords i and j and the expected number of co-occurrences of keywords i and j under the assumption that occurrences of the two keywords are independent.

Map formation: The VOS mapping technique constructs a two-dimensional map in which keywords $1, ..., n$ (where n is the total number of keywords) are placed such that the distance between any pair of keywords i and j reflects their similarity $s_{i,j}$ as accurately as possible. Keywords with a high similarity are located close to each other, while keywords with a low similarity are located far from each other.

The goal of the VOS mapping technique is to minimize the weighted sum of the squared Euclidean distances between all pairs of keywords [73]. The higher the similarity between the two keywords, the higher the weight of their squared distance in the summation. The specific function minimized by the mapping technique is:

$$V(x_1, ..., x_n) = \sum_{i<j} s_{i,j} \|x_i - x_j\|^2 \tag{3.5}$$

where x_i denotes the location of keyword i in a two-dimensional space, and where $\|x_i - x_j\|$ denotes the Euclidean distance between keywords i and j. To avoid trivial maps in which

all keywords have the same location, minimization is subject to the constraint that the average distance between two keywords must be equal to 1. Specifically:

$$\frac{2}{n(n-1)} \sum_{i<j} \|x_i - x_j\| = 1 \tag{3.6}$$

The constrained optimization problem—minimizing Equation 3.5, subject to Equation 3.6—is solved in two steps [72]. The constrained problem is first converted into an unconstrained problem. Second, the unconstrained problem is solved using a variant of the SMACOF algorithm, an optimization algorithm commonly used in multidimensional scaling to produce human-understandable network or graph layouts through minimization of a stress function over the positions of nodes in the graph [129].

Clustering of Keywords: Keywords are assigned to clusters, and the number of clusters is determined, through optimization. This is a common approach for clustering nodes in a network [130]. Potential assignments of keywords to clusters are assessed using the function:

$$V(c_1, ..., c_n) = \sum_{i<j} \delta(c_i, c_j)(s_{i,j} - \gamma) \tag{3.7}$$

where c_i is the cluster that keyword i has been assigned to. $\delta(c_i, c_j)$ is a difference function that yields 1 if $c_i = c_j$ and 0, otherwise. γ determines the level of clustering, with higher values yielding a larger number of clusters. This equation is unified with the mapping function minimized in Equation 3.5, and includes the same similarity measurement $s_{i,j}$ calculated in Equation 3.2.

There is no maximum number of keywords per cluster. The minimum number of keywords is controlled using a user-specified parameter. We used the default, a minimum of one keyword. The clustering algorithm will merge small clusters in cases where merging does not affect the result of Equation 3.7. Therefore, any small cluster that remain are ones that affect the results of the equation.

Equation 3.7 is maximized using the smart local moving algorithm [131]. Following the optimization, the assignment of keywords to clusters that maximizes Equation 3.7 is

returned. This process yields a small number of clusters containing keywords that are targeted disproportionately often together in publications.

Translation, rotation, and reflection: The optimization problem introduced in Equation 3.5 does not have a single global optimal solution [72]. However, consistent results are desirable. To ensure that the same co-occurrence matrix always yields the same map, three transformations are applied after optimization:

- **Translation:** The solution is translated to be centered at the origin.
- **Rotation:** Principle component analysis is applied in order to maximize variance on the horizontal dimension.
- **Reflection:** If the median of $x_{1,1}, ..., x_{n,1}$ is larger than 0, the solution is reflected in the vertical axis. If the median of $x_{1,2}, ..., x_{n,2}$ is larger than 0, the solution is reflected in the horizontal axis.

CHAPTER 4

CHOOSING THE FITNESS FUNCTION FOR THE JOB: AUTOMATED GENERATION OF TEST SUITES THAT DETECT REAL FAULTS

4.1 INTRODUCTION

Proper verification practices are needed to ensure that developers deliver reliable software. *Testing* is an invaluable, widespread verification technique. However, testing is a notoriously expensive and difficult activity [132], and with exponential growth in the complexity of software, the cost of testing has risen accordingly. Means of lowering the cost of testing without sacrificing verification quality are needed.

Much of that cost can be traced directly to the human effort required to conduct most testing activities, such as producing test input and expected output. One way of lowering such costs may lie in the use of automation to ease this manual burden [11]. Automation has great potential in this respect, as much of the invested human effort is in service of tasks that can be framed as *search* problems [14]. For example, unit test case generation can naturally be seen as a search problem [11]. There are hundreds of thousands of test cases that could be generated for any particular class under test (CUT). Given a well-defined testing goal, and a numeric scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can systematically search the space of possible test inputs to locate those that meet that goal [15].

The effective use of search-based generation relies on the performance of two tasks—

selecting a measurable test goal and selecting an effective fitness function for meeting that goal. Adequacy criteria offer checklists of measurable test goals based on the program source code, such as the execution of branches in the control-flow of the CUT [1, 16, 17]. Because such criteria are based on source code elements, we refer to them as "white-box" test selection criteria. Often, however, goals such as "coverage of branches" are an approximation of a goal that is harder to quantify—we really want tests that will reveal faults [18]. "Finding faults" is not a goal that can be measured, and cannot be translated into a distance function.

To generate effective tests, we must identify criteria—and corresponding fitness functions—that are correlated with an increased probability of fault detection. If branch coverage is, in fact, correlated with fault detection, then—even if we do not care about the concept of branch coverage itself—we will end up with effective tests. However, the need to rely on approximations leads to two questions. First, *can common fitness functions produce effective tests?* If so, *which of the many available fitness functions should be used to generate tests?* Unfortunately, testers are faced with a bewildering number of options—an informal survey of two years of testing literature reveals 28 viable white-box fitness functions—and there is little guidance on when to use one criterion over another [85].

While previous studies on the effectiveness of adequacy criteria in test generation have yielded inconclusive results [85, 133–135], two factors allow us to more deeply examine this problem—particularly with respect to search-based generation. First, tools are now available that implement enough fitness functions to make unbiased comparisons. The EvoSuite framework offers over twenty options, and uses a combination of eight fitness functions as its default configuration [136]. Second, more realistic examples are available for use in assessment of suites. Much of the previous work on adequacy effectiveness has been assessed using mutants—synthetic faults created through source code transformation [75]. Whether mutants correspond to the types of faults found in real projects has not been firmly established [31]. However, the Defects4J project offers a large database of

real faults extracted from open-source Java projects [32]. We can use these faults to assess the effectiveness of search-based generation on the complex faults found in real software.

In this study, we have used EvoSuite and eight of its white-box fitness functions (as well as the default multi-objective configuration and a combination of branch, exception, and method coverage) to generate test suites for the fifteen systems, and 593 of the faults, in the Defects4J database. In each case, we seek to understand *when and why* generated test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Thus, in each case, we have recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. We have recorded a set of traditional *source code metrics*—sixty metrics related to cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics—for each class associated with a fault the Defects4J dataset. By analyzing these generation factors and metrics, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of machine learning algorithms—the factors correlating with a high or low likelihood of fault detection. To summarize our findings:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite's default combination performs well, the difficulty of simultaneously

balancing eight functions prevents it from outperforming all individual criteria.

- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness

functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

This work extends a prior conference publication [20]. The earlier paper looked at the same core research questions. However, in order to undergo a deeper investigation into the topic, we have contributed an additional 240 faults, from fifteen new systems, to Defects4J—almost doubling the size of the database. Our updated study includes suites generated over those new case examples, adding further observations and points of discussion. We have also used the findings of our separate research into combinations of fitness functions [137] to reformulate and extend our experiments and discussion of the effects of combining criteria. In addition, we have changed how we build and classify data in our treatment learning analysis, added the source code metric analysis, and have included a far deeper examination of the factors indicating success or lack thereof in test generation. Our observations provide evidence for the anecdotal findings of other researchers [19–23] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. While more research is still needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites, our findings in this revised and extended case study offer lessons in understanding the use, applicability, and combination of common fitness functions.

4.2 BACKGROUND

Search-Based Software Test Generation

Test case creation can naturally be seen as a search problem [14]. Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [15, 18]. Given a well-defined testing goal, and a scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [138]. Metaheuristics are often inspired by natural phenomena, such as swarm behavior [139] or evolution [140].

While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness function, and (3), this score is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process [141].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex [18]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [142]. Such approaches have been applied to a wide variety of testing goals and scenarios [18].

Adequacy Metrics and Fitness Functions

When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. These two factors are linked. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*? The same

question applies when adding new tests—if we have not observed new faults, have we not yet written *adequate* tests?

The concept of adequacy provides developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of tests. The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of the software's control flow, and complex boolean conditional statements [1, 16, 17].

Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. These requirements are expressed as a series of *test obligations*—properties that must be met by the corresponding test suite. For example, the branch coverage criterion requires that each program expression that can cause control flow to diverge—i.e., loop conditions, switch statements, and if-conditions—evaluate to each possible outcome. In this case, a test obligation would indicate a particular expression and a targeted outcome for the evaluation of that expression. If tests fulfill the list of obligations prescribed by the criterion, than testing is deemed "adequate" with respect to faults that manifest through the structures of interest to the criterion.

Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [143][1]. It is easy to understand the popularity of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured [144]. These very same qualities make adequacy criteria ideal for use as automated test generation targets. In search-based testing, the fitness function needs to capture the testing objective and guide the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Functions must be efficient to execute, as they will be calculated thousands of times over

[1]For example, see `https://codecov.io/`.

a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates. Adequacy criteria are ideal optimization targets for automated test case generation as they can be straightforwardly transformed into efficient, informative fitness functions [145]. Search-based generation often can achieve higher coverage than developer-created tests [146].

4.3 STUDY

To generate unit tests that are effective at finding faults, we must identify criteria and corresponding fitness functions that increase the probability of fault detection. As we cannot know what faults exist before verification, such criteria are approximations—intended to increase the probability of fault detection, but offering no guarantees. Thus, it is important to turn a critical eye toward the choice of fitness function used in search-based test generation. We wish to know whether commonly-used fitness functions produce effective tests, and if so, why—and under what circumstances—do they do so?

More empirical evidence is needed to better understand the relationships between adequacy criteria, fitness functions and fault detection [143]. Many criteria exist, and there is little guidance on when to use one over another [85]. To better understand the real-world effectiveness, use, and applicability of common fitness functions and the factors leading to a higher probability of fault detection, we have assessed the EvoSuite test generation framework and eight of its fitness functions (as well as the default multi-objective configuration) against 593 real faults, contained in the Defects4J database. In doing so, we wish to address the following research questions:

1. How capable are generated test suites at detecting real faults?
2. Which fitness functions have the highest likelihood of fault detection?
3. Does an increased search budget improve the effectiveness of the resulting test suites?
4. Under what situations can a combination of criteria outperform a single criterion?

5. What factors correlate with a high likelihood of fault detection?

The first three questions allow us to establish a basic understanding of the effectiveness of each fitness function—are *any* of the functions able to generate fault-detecting tests and, if so, are any of these functions more effective than others at the task? However, these questions presuppose that only one fitness function can be used to generate test suites. Many search-based generation algorithms can simultaneously target *multiple* fitness functions [137]. Therefore, we also ask question 4—when does it make sense to employ a set of fitness functions instead of a single function?

Finally, across all criteria, we also would like to gain insight into the factors that influence the likelihood of detection. To inspire new research advances, we desired a deeper understanding of when generated suites are likely to detect a fault, and when they will fail. We have made use of *treatment learning*—a machine learning technique designed to take classified data and identify sets of attributes, along with bounded values of such attributes, that are highly correlated with particular outcomes. In our case, these "outcomes" are associated with whether generated suites from each fitness function detect a fault or not. We have gathered factors from two broad sets:

- **Test Generation Factors** are related to the test suites produced—identifying coverage attained, suite size, and obligation satisfaction.
- **Source Code Metrics** examine the faulty classes being targeted, and ask whether factors related to the classes themselves—i.e., the number of private methods or cloned code—can impact the test generation process.

We have created datasets based off of both sets of factors and applied treatment learning to assess which factors strongly affected the outcome of test generation. We make these datasets, as well as the new Defects4J case examples, available to other researchers to aid in future advances (see Sections 4.3 and 4.3).

In order to investigate these questions, we have performed the following experiment:

1. **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 4.3).
2. **Developed New Case Examples:** We have also mined an additional 240 faults from ten new projects, and added these faults to the Defects4J database (Section 4.3).
3. **Recorded Source Code Metrics:** For each affected class (both faulty and fixed versions), we measure a series of sixty source code metrics, related to cloning, cohesion, coupling, documentation, inheritance, and class size (Section 4.3).
4. **Generated Test Suites:** For each fault, we generated 10 suites per criterion using the fixed version of each CUT. We performed with both a two-minute and a ten-minute search budget per CUT (Section 4.3).
5. **Generated Test Suites for Combinations of Criteria:** We perform the same process for EvoSuite's default configuration—a combination of eight criteria—and a combination of branch, exception, and method coverage (Section 4.3).
6. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section 4.3).
7. **Assessed Fault-finding Effectiveness:** We measure the proportion of test suites that detect each fault to the number generated (Section 4.3).
8. **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that may influence suite effectiveness, related to coverage, suite size, and obligation satisfaction (Section 4.3).
9. **Prepare Datasets**: Datasets were prepared for learning purposes by by adding classifications based on fault detection to each entry in the generation factor and code metric datasets. Separate datasets were prepared for each generation budget and fitness function, as well as sets based on overall fault detection (across all fitness functions and function combinations) for each budget (Section 4.3).
10. **Performed Treatment Learning:** We apply the TAR3 learner to identify factors correlated to each classification for each dataset (Section 4.3).

Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [32][2]. The original dataset consisted of 357 faults from five projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), and Time (27 faults). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings.

In order to expand our study to a larger set of case examples, we have added an additional ten systems to Defects4J. The process of adding new faults is semi-automated, and requires the development of build files that work across project versions. The commit messages of the project's version control system are scanned for references to issue identifiers. These versions are considered to be candidate "fixes" to the referenced issues. The human-developed test suite for that version is then applied to previous project versions. If one or more test cases pass on the "fixed" version and fail on the earlier version, then that version is retained as the "faulty" variant. The code differences between versions are captured as a patch file, which must then be manually minimized to remove any differences that are not required to reproduce the fault.

Following this process, we added 240 faults—bringing the total to 597 faults from 15 projects. The new projects include: CommonsCLI (24 faults), CommonsCSV (12), CommonsCodec (22), CommonsJXPath (14), Guava (9), JacksonCore (13), JacksonDatabind

[2]Available from `http://defects4j.org`

(39), JacksonXML (5), Jsoup (64), and Mockito (38)[3]. The ten new systems were chosen because they are popular projects and have reached a reasonable level of maturity—meaning that the detected faults are often relatively complicated. Two of these systems, Guava and Mockito, were the subjects of recent research challenges at the Symposium on Search-Based Software Engineering [21, 147]. Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 593 faults that we used in our study.

Code Metric-based Characterization

When assessing the results of our study, we wish to gain understanding of *when and why* our test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Gaining such understanding requires fine-grained information about the faults being targeted—and, more specifically, the classes being targeted for test generation. To assist in gaining this understanding, we have turned to traditional *source code metrics*. Using the SourceMeter framework[4], we have gathered a set of 60 cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics for each class associated with a fault included in the Defects4J dataset.

Such metrics, commonly used as part of research on effort estimation [148] and defect prediction [149], are considered to have substantial predictive power. By characterizing the classes that host the Defects4J faults using these code metrics, we can better understand the results of our research. Using the SourceMeter framework, we have measured 60 source code metrics for each class related to a fault in the Defects4J dataset. These metrics are recorded for both the faulty and fixed versions of each affected class. The metrics may be

[3]The new faults have been submitted to Defects4J as a pull request. Until they are accepted, they can be found at the *additional-faults-1.4* branch of `https://github.com/Greg4cr/defects4j`.

[4]Available from `https://www.sourcemeter.com`.

Table 4.1: List of metrics gathered for each class, separated by category. The median and standard deviation are listed for each.

Category	Abbreviation	Metric	Median	Standard Deviation
Clone	CC	Clone Coverage	0.00	0.16
	CCL	Clone Classes	0.00	10.80
	CCO	Clone Complexity	0.00	453.26
	CI	Clone Instances	0.00	27.86
	CLC	Clone Line Coverage	0.00	0.09
	CLLC	Clone Logical Line Coverage	0.00	0.14
	LDC	Lines of Duplicated Code	0.00	161.06
	LLDC	Logical Lines of Duplicated Code	0.00	147.78
Cohesion	LCOM5	Lack of Cohesion in Methods 5	1.00	7.31
Complexity	NL	Nesting Level	4.00	2.94
	NLE	Nesting Level Else-If	3.00	1.93
	WMC	Weighted Methods per Class	55.00	149.58
Coupling	CBO	Coupling Between Object Classes	8.00	11.79
	CBOI	Coupling Between Object Classes Inverse	5.00	61.38
	NII	Number of Incoming Invocations	16.00	172.67
	NOI	Number of Outgoing Invocations	14.00	31.31
	RFC	Response Set For Class	37.50	56.43
Documentation	AD	API Documentation	1.00	0.32
	CD	Comment Density	0.38	0.18
	CLOC	Comment Lines of Code	127.00	460.47
	DLOC	Documentation Lines of Code	102.50	443.12
	PDA	Public Documented API	7.00	28.31
	PUA	Public Undocumented API	0.00	8.77
	TCD	Total Comment Density	0.36	0.17
	TCLOC	Total Comment Lines of Code	144.50	465.29
Inheritance	DIT	Depth of Inheritance Tree	1.00	1.15
	NOA	Number of Ancestors	1.00	1.71
	NOC	Number of Children	0.00	2.07
	NOD	Number of Descendants	0.00	3.39
	NOP	Number of Parents	1.00	0.86
Size	LLOC	Logical Lines of Code	208.50	462.19
	LOC	Lines of Code	382.50	879.00
	NA	Number of Attributes	8.00	14.33
	NG	Number of Getters	3.00	14.96
	NLA	Number of Local Attributes	6.00	10.24
	NLG	Number of Local Getters	2.00	8.58
	NLM	Number of Local Methods	21.00	35.03
	NLPA	Number of Local Public Attributes	0.00	4.35
	NLPM	Number of Local Public Methods	9.00	29.50
	NLS	Number of Local Setters	0.00	5.23
	NM	Number of Methods	28.50	50.96
	NOS	Number of Statements	109.50	279.94
	NPA	Number of Public Attributes	0.00	6.17
	NPM	Number of Public Methods	14.00	44.25
	NS	Number of Setters	0.00	11.33
	TLLOC	Total Logical Lines of Code	275.50	503.28
	TLOC	Total Lines of Code	495.00	919.51
	TNA	Total Number of Attributes	10.00	17.34
	TNG	Total Number of Getters	4.00	20.96
	TNLA	Total Number of Local Attributes	8.00	14.20
	TNLG	Total Number of Local Getters	2.00	10.62
	TNLM	Total Number of Local Methods	27.00	44.33
	TNLPA	Total Number of Local Public Attributes	0.00	4.67
	TNLPM	Total Number of Local Public Methods	12.00	34.82
	TNLS	Total Number of Local Setters	0.00	6.20
	TNM	Total Number of Methods	37.00	79.02
	TNOS	Total Number of Statements	143.00	293.37
	TNPA	Total Number of Public Attributes	0.00	6.36
	TNPM	Total Number of Public Methods	19.00	62.78
	TNS	Total Number of Setters	0.00	14.01

divided into the following categories:

- **Clone Metrics** are used to measure the occurrence of Type-2 clones in the class—code fragments that are structurally identical, but may differ in variable names, liter-

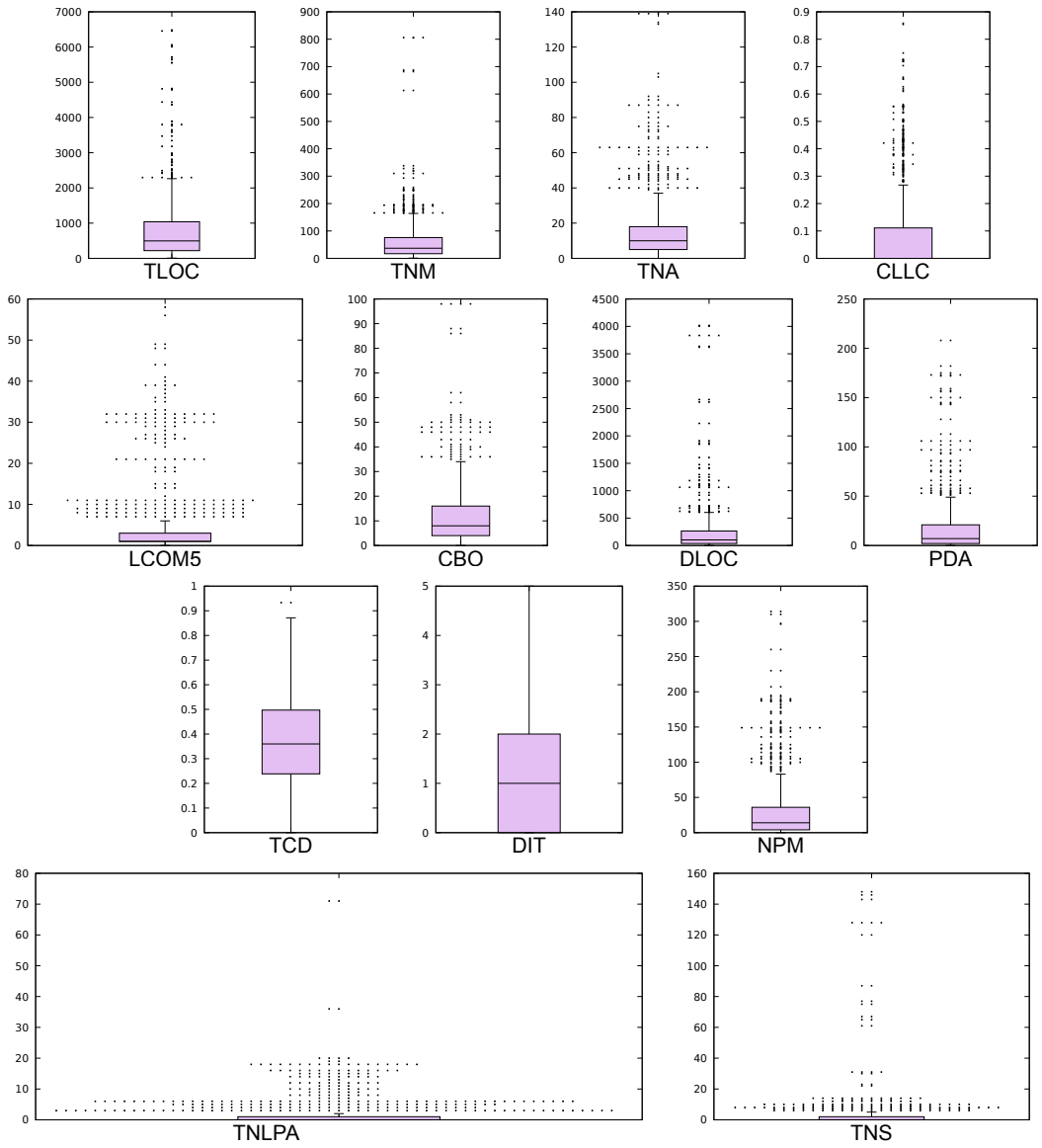


Figure 4.1: Boxplots illustrating the median, first, and third quartile values for select metrics from the dataset.

als, and identifiers [150].

- **Cohesion Metrics** assess the level of cohesion in a class—whether attributes and the operations that use them are organized into one class, and whether there are methods that are unrelated to the other methods and attributes in the class [151].
- **Complexity Metrics** assess the complexity of the class, using information such as the depth of nesting and the number of control-flow paths through methods [152]. Complexity is often used as part of defect prediction, as more complex methods are expected to contain more defects than simpler methods.
- **Coupling Metrics** assess the level of dependency between classes [153]. When designing a system, developers are cautioned to minimize the level of coupling—to make each class as independent as possible. Coupling metrics are used to identify design weaknesses and architectural bottlenecks.
- **Documentation Metrics** measure the degree that a class is documented by its developers [154]. Well-documented code is often thought to be higher quality code, and these metrics can identify classes that may have received less attention.
- **Inheritance Metrics** tracks the relationships between parent and child classes along the class hierarchy [153]. As inheritance defines a form of dependence, such metrics are useful for identifying how code changes can propagate through a system.
- **Size Metrics** characterize the size and complexity of a class based on structural elements such as the number of lines of code, methods, attributes, setters, and getters [155]. Such metrics can be used, at a glance, to identify some of the more complex classes in a system.

Table 4.1 lists the gathered metrics. Detailed definitions may be found on the SourceMeter documentation [156]. Table 4.1 notes the median and standard deviation for all metrics.

Characterizing the "Average" Class

To help illustrate the "average" class from Defects4J, we have included boxplots for several of the measured metrics in Figure 4.1. Each box depicts the first and third quartiles, as well as the median value. Outliers—points more than 1.5 times the interquartile range—are depicted as well. Rather than depict all 60 metrics, we show a subset indicated as important in our case study to help characterize the studied classes. First, to set context:

- **TLOC (Total Lines of Code)** indicates the amount of code is in a class, including comments and whitespace. TLOC include lines in anonymous, nested, and local classes. The median TLOC is 495, but a large standard deviation (919.51) indicates that classes have a wide range of sizes. Studied classes tend to cluster between 0-1,000 TLOC, and the largest class has 6,481 TLOC.
- **TNM (Total Number of Methods)** is the number of methods in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNM is 37, with the maximum being 806. Classes in Defects4J tend to have less than 100 methods. Again, however, there are a number of outliers.
- **TNA (Total Number of Attributes)** is the number of attributes in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNA is 10—with values tending to cluster between 5-20—and the maximum is 139.

The following metrics were indicated in our case study as being able to explain why generated test suites are able—or not able—to detect faults. We will discuss the implications of these findings in Section 4.4. Here, we use these metrics to further characterize the "average" class in Defects4J.

- **CLLC (Clone Logical Line Coverage)** is the ratio of code covered by code duplications in the class to the size of the class, expressed in terms of logical lines of code (non-empty, non-comment lines). **Clone Coverage** is the same measurement, except

that it includes comments and whitespace. The CLLC and CC are both largely concentrated towards the low end of the scale, with a median of 0 for both—no code being duplicated—and a relatively low standard deviation (0.14).

- **LCOM5 (Lack of Cohesion in Methods 5)** measures the lack of cohesion and computes how many coherent classes the class could be split into. Although there are many outliers, the LCOM5 is largely concentrated towards the low end of the scale—with a median of 1—indicating that classes tend to be highly cohesive.
- **CBO (Coupling Between Objects)** indicates the number of classes that serve as dependencies of the target class (by inheritance, method call, type reference, or attribute reference). Classes dependent on many other classes are very sensitive to the changes in the system, and can be harder to test or evolve. The median CBO is 8 and standard deviation is 11.79, indicating that—while classes are connected—the average class does not overly depend on the rest of the system.
- **DLOC (Documented Lines of Code)** simply measures the number of line of code that are comments (in-line or standalone). The median number of documented lines is 102. However, there are a large range of values, with the maximum DLOC at a staggering 4,017.
- **PDA (Public Documented API)** is the number of public methods with documentation. The median PDA is 7. Again, however, there is a large variance in PDA, with a standard deviation of 28.31 and a large number of outlying values.
- **TCD (Total Comment Density)** is the ratio of the comment lines to the sum of its comment and logical lines of code, including nested, anonymous, and local classes. The **CD (Comment Density)**, also noted as important, is the same measurement excluding nested, anonymous, and local classes. The median TCD is 0.36 and the median CD is 0.38, indicating that the "average" Defects4J class is approximately one-third documentation.
- **DIT (Depth of Inheritance Tree)** measures the length of the path from the class

to its furthest ancestor in the inheritance tree. The median DIT is 1—many classes have a parent. The maximum DIT is 5, but this is an extreme outlier—the standard deviation is 1.15 and almost all classes have 0-2 levels of ancestors.

- **NPM (Number of Public Methods)** indicates the number of methods that are publicly-accessible in the class, including inherited methods. Closely related is the **TNLPM (Total Number of Local Public Methods)**, which includes nested, anonymous, and local classes, but excludes inherited methods. The median NPM is 14, with the majority concentrated below 50. The median TNLPM is slightly lower—12. Both are lower than the TNM, indicating that many classes have a large number of private methods.
- **NLPA (Number of Local Public Attributes)**, similarly, indicates the number of publicly-accessible attributes in the class, excluding inherited attributes. The **TNLPA (Total Number of Local Public Attributes)** includes nested, anonymous, and local classes. Both metrics are less consistent than many of the others, with a large number of outlying values. However, the median for both is 0—indicating that there are often no public attributes in a class. This is notably lower than the TNA, indicating that class attributes are generally private.
- **TNS (Total Number of Setters)** records the number of setter methods in the class, including inherited methods. Our study also indicated the **NS (Number of Setters)**—excluding nested, anonymous, and local classes—and **TNLS (Total Number of Local Setters)**—excluding inherited setters—as important. This set of attributes also has a relatively large number of outliers, but is concentrated towards the low end of the scale. All three of these metrics have a median value of 0, indicating that most classes in Defects4J have no setter methods.

Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [22]. In this study, we used EvoSuite version 1.0.5, and the following fitness functions:

Branch Coverage (BC): A test suite satisfies branch coverage if all control-flow branches are taken during test execution—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates the predicate to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how "close" the targeted predicate is to being true, using a cost function based on the predicate formula [145].

Direct Branch Coverage (DBC): Branch coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. When a test covers a branch indirectly, it can be more difficult to understand how coverage was attained. Direct branch coverage requires each branch to be covered through a direct method call.

Line Coverage (LC): A test suite satisfies line coverage if it executes each non-comment source code line at least once. To cover each line of source code, EvoSuite tries to ensure that each basic code block is reached. The branch distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line in the code, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): The goal of exception coverage is to build test suites that force the CUT to throw exceptions—either declared or undeclared. As the number of possible exceptions that a class can throw cannot be known ahead of time, the fitness function

rewards suites that throw more exceptions. As this function is based on the number of discovered exceptions, the number of "test obligations" may change each time EvoSuite is executed on a CUT.

Method Coverage (MC): Method Coverage simply requires that all methods in the CUT are executed at least once, through direct or indirect calls. The fitness function for method coverage is discrete, as a method is either called or not called.

Method Coverage (Top-Level, No Exception) (MNEC): Generated test suites sometimes achieve high levels of method coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

Output Coverage (OC): Output coverage rewards diversity in the method output by mapping return types to a list of abstract values [157]. A test suite satisfies output coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.

Weak Mutation Coverage (WMC): Test effectiveness is often judged using mutants [75]. Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards reaching and discovering mutated statements [158].

Rojas et al. provide a primer on each of these fitness functions [159]. In order to study the effect of combining fitness functions, we also generate test suites using two combinations. The first is EvoSuite's default configuration—a combination of all of the above methods (called the **"Default Combination"**). The second is a combination of branch, exception, and method coverage (called the **"BC-EC-MC Combination"**). This combination was identified as an effective baseline in our prior work studying combination efficacy on the five original systems from Defects4J [137].

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. EvoSuite generates assertion-based oracles. Generating oracles based on the fixed version of the class means that we can confirm that the fault is actually detected, and not just that there are coincidental differences in program output. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be "correct" in order to guard against future issues. Tests that fail on the faulty version, then, detect behavioral differences between the two versions.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits each fitness function. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. This resulted in the generation of 118,600 test suites (two budgets, ten trials, ten configurations, 593 faults).

Generation tools may generate flaky (unstable) tests [22]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, fewer than one test tends to be removed from each suite (see Table 4.2).

Test Generation Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault. We refer to this as the likelihood of fault detection.

Table 4.2: Statistics on generated test suites (each statistic is explained in Section 4.3). Values are averaged over all faults (i.e., the average number of obligations, average number of tests removed, etc.). "Default Combination" is a combination of the eight individual fitness functions. "BC-EC-MC Combination" combines the branch, exception, and method coverage fitness functions.

Method	Budget	Total Obligations	% Obligations Satisfied	Suite Size	Suite Length	# Tests Removed	% Line Coverage (Fixed)	% Line Coverage (Faulty)	% Branch Coverage (Fixed)	% Branch Coverage (Faulty)
Branch Coverage (BC)	120	295.15	58.32%	36.33	195.96	0.38	61.98%	62.04%	58.96%	58.87%
	600		65.00%	44.27	268.95	0.75	67.35%	67.23%	65.44%	65.19%
Direct Branch (DBC)	120	295.15	54.60%	38.32	217.55	0.35	60.01%	59.59%	56.37%	55.93%
	600		61.79%	48.27	310.52	0.73	65.53%	64.96%	63.32%	62.65%
Exception Coverage (EC)	120	12.47	99.41%	11.99	35.54	0.23	21.35%	21.36%	15.82%	15.99%
	600	12.57	99.38%	12.12	36.09	0.26	21.60%	21.63%	15.97%	16.15%
Line Coverage (LC)	120	329.90	61.29%	30.32	162.08	0.28	62.27%	61.69%	53.60%	53.09%
	600		66.79%	34.73	207.65	0.45	67.53%	66.90%	59.11%	58.51%
Method Coverage (MC)	120	31.92	78.99%	22.00	73.78	0.05	37.51%	37.40%	29.18%	29.26%
	600		83.30%	24.32	86.62	0.09	38.91%	38.93%	30.36%	30.52%
Method, No Exception (MNEC)	120	31.92	77.59%	21.68	74.54	0.05	39.06%	38.99%	30.39%	30.48%
	600		82.19%	23.79	88.88	0.10	40.84%	40.78%	31.75%	31.91%
Output Coverage (OC)	120	185.89	42.78%	29.04	133.04	0.12	39.00%	38.82%	32.90%	32.88%
	600		46.17%	32.68	161.32	0.26	40.81%	40.67%	34.47%	34.47%
Weak Mutation (WMC)	120	508.38	56.20%	26.48	164.51	0.16	56.02%	55.87%	49.73%	49.50%
	600		62.94%	32.60	246.57	0.42	61.58%	61.31%	56.19%	55.80%
Default Combination	120	1673.19	53.92%	48.71	358.93	0.55	58.25%	58.05%	53.15%	52.95%
	600	1681.19	60.72%	64.57	550.03	1.08	64.65%	64.07%	60.91%	60.30%
BC-EC-MC Combination	120	345.59	63.81%	50.73	262.03	1.06	62.91%	62.22%	59.95%	59.12%
	600	354.84	69.69%	65.10	368.83	2.04	68.00%	67.12%	66.30%	65.24%

To better understand the generation factors that influence effectiveness, we also collected the following for each test suite:

Number of Test Obligations: Given a CUT, each fitness function will calculate a series of test obligations to cover (as defined in Section 4.2). The number of obligations is informative of the difficulty of the generation, and impacts the size and formulation of tests [160]. Note that the number of test obligations is dependent on the CUT, and does not differ between budgets except in the case of exception coverage. As exception coverage simply counts the number of observed exceptions, it does not have a consistent set of obligations each time generation is performed.

Percentage of Obligations Satisfied: This factor indicates the ability of a fitness function to cover its goals. A suite that covers 10% of its goals is likely to be less effective than one that achieves 100% coverage.

Test Suite Size: We have recorded the number of tests in each test suite. Larger suites are often thought to be more effective [135,161]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

Test Suite Length: Each test consists of one or more method calls. Even if two suites have

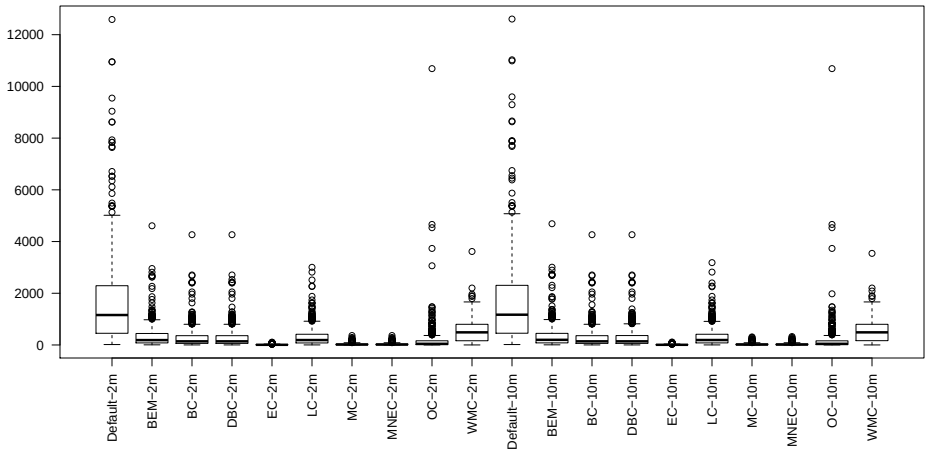


Figure 4.2: Total Obligations

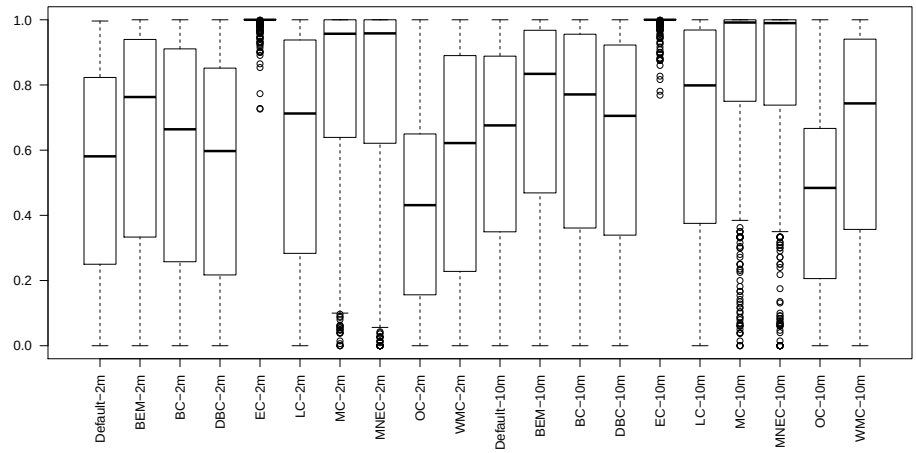


Figure 4.3: % Obligations Satisfied

Figure 4.4: Boxplots of the total obligations and % of obligations satisfied for suites generated for each fitness configuration and search budget.

the same number of tests, one may have much *longer* tests—making more method calls. In assessing the effect of suite size, we must also consider the length of each test case.

Number of Tests Removed: Any tests that do not compile, or that return inconsistent results, are automatically removed. We track the number removed from each suite.

Code Coverage: As the premise of many adequacy criteria is that faults are more likely to be detected if structural elements of the code are thoroughly executed, the resulting coverage of the code may indicate the effectiveness of a test suite. Using EvoSuite's cov-

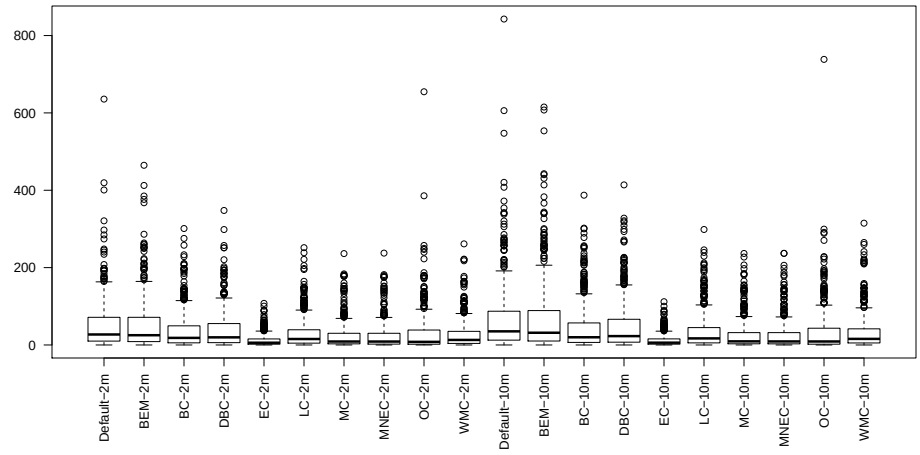


Figure 4.5: Suite Size

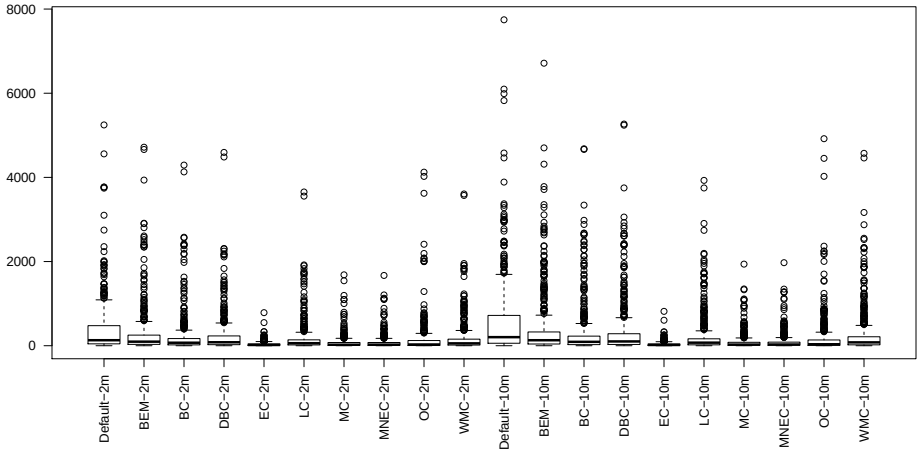


Figure 4.6: % Line Coverage (Fixed)]

Figure 4.7: Boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget.

erage measurement capabilities, we have measured the line and branch coverage achieved by each suite when executed over both the faulty and fixed versions of each CUT. Due to instrumentation issues, we were unable to measure coverage over two systems—JacksonDatabind and Mockito. However, we were able to measure coverage over the remaining 516 faults.

Table 4.2 records, for each fitness function and budget, the average values attained for each of these measurements over all faults for which we were able to take measurements. In Figure 4.4, we show boxplots of the total obligations and % of obligations satisfied for

suites generated for each fitness configuration and search budget. Branch and direct branch coverage will always have the same number of obligations. Line coverage tends to operate in the same approximate range. Exception, method, and MNEC have the fewest obligations, while weak mutation coverage tends to have the most obligations of the individual fitness functions. Naturally, the two combinations have more obligations—the combination of their member functions. In terms of satisfaction, the three fitness functions with the fewest obligations—exception, method, and MNEC—all also have the highest satisfaction rate. Output coverage has the lowest average, and generally lower, satisfaction rates. For all functions other than Exception Coverage, there tends to be large variance in the satisfaction rate.

In Figure 4.7, we show boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget. Most fitness functions yield similar median suite size and variance in results. Exception coverage tends to yield the smallest suites, owing to its small number of test obligations. Method coverage and MNEC yield smaller suites than other fitness functions, but not significantly so. Output coverage tends to have relatively large test suites—comparable in size to branch and direct branch coverage. Again, the two combinations tend to yield larger test suites, but not significantly larger than those for branch, direct branch, and weak mutation coverage. Test suite length largely offers similar observations. However, we do note that the "default combination" tends to yield very *long* test suites, composed of more method calls than suites for other fitness configurations. This is not the case for the BC-EC-MC combination.

Like with obligation satisfaction, there is a large variance in the line coverage attained by test suites, regardless of fitness function. Exception coverage tends to achieve both the lowest coverage and the least variance in coverage. This is reasonable, as the fitness function for exception coverage has no mechanism to encourage class exploration. Naturally, branch, direct branch, line, and weak mutation coverage tend to attain high coverage rates over classes, as all four use coverage-based fitness mechanisms. Exception, method,

MNEC, and output coverage are based on source code elements, but all have fitness representations that are not based on control flow. As a result, they tend to attain lower coverage levels.

Dataset Preparation for Treatment Learning

To understand the factors leading to detection—or lack of detection—of a fault, we have collected two basic sets of data for each fault: **test generation factors** related to the test suites produced and **source code metrics** examining the classes being targeted for unit test generation.

A standard practice in machine learning is to *classify data*—to use previous experience to categorize new observations [162]. We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from classifications to discover *which factors correspond most strongly to a class of interest*—a process known as treatment learning [163]. Treatment learning approaches take the classification of an observation and reverse engineers the evidence that led to that categorization. Such learners produce a *treatment*—a small set of attributes and value ranges for each that, if imposed, will identify a subset of the original data skewed towards the target classification. In this case, a treatment notes the metrics—and their values—that indicate that generated test suites will detect a fault.

For example, the treatment $[NS = [0.00..1.00), TCD = [0.52..0.93]]$—derived from the code metric-based dataset—indicates that the subset of the data where the Number of Setters is less than 1 and the Total Comment Density is between 52-93% has a higher percentage of "Yes" classifications ("Yes" implying that the fault is detected, while a "No" classification indicates that the fault was not detected) than the base dataset. Within this subset, "Yes" classifications account for 72.00% of the class distribution of the subset—compared to 47.44% of the base distribution.

Using the TAR3 treatment learner [164], we have generated five treatments from each

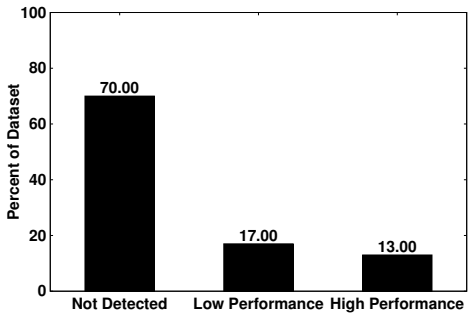


Figure 4.8: 2m Budget

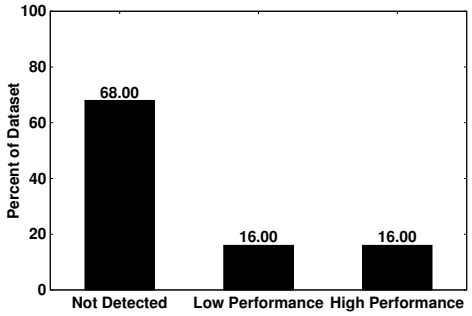


Figure 4.9: 10m Budget

Figure 4.10: Baseline class distribution of the generation factors datasets used for treatment learning.

dataset that target each of the applied verdicts. A user can specify the minimum number of examples that may make up the data subset matching a treatment in order to ensure minimum support for a treatment. We require the subset to contain at least 20% of the total dataset. In addition, a limit can be placed on the number of attributes chosen for a treatment. It is thought that large treatments—those recommending more than five attribute-range pairs—may not have more explanatory power than smaller treatments [164]. Therefore, we also limit the treatment size to five attribute-value pairings.

Generation-Based Datasets

As discussed in Section 4.3—to better understand the combination of factors correlating with effective fault detection—we have collected the following statistics for each generated test suite: the number of obligations, the percent satisfied, suite size, suite length, number

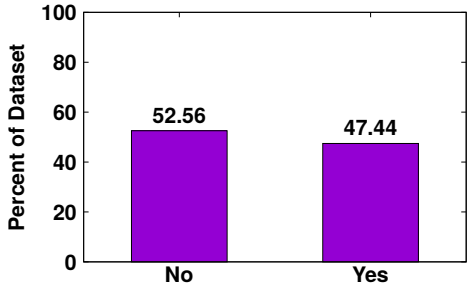


Figure 4.11: 2m Budget

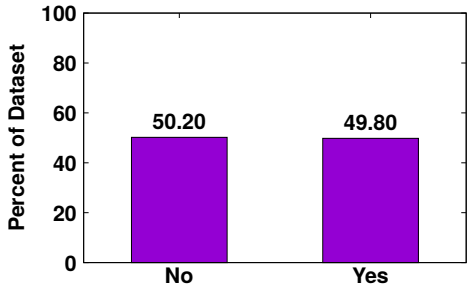


Figure 4.12: 10m Budget

Figure 4.13: Baseline class distributions for the "overall" code metrics datasets used for treatment learning.

of tests removed, as well as branch (BC) and line coverage (LC) over the fixed and faulty versions of the CUT.

This collection of factors forms a dataset that can be used to analyze the impact of these factors on efficacy. Each record in the dataset corresponds to the values of these attributes for each fault, and for each criterion or combination of criteria. In total, this dataset contains 5160 records (516 faults for which we could collect all statistics, times the ten fitness configurations). We build separate datasets for each search budget. We can then use the likelihood of fault detection (D) as the basis for a class variable—discretized into three values: "not detected" ($D = 0$), "low performance" ($D < 70\%$), and "high performance" ($D \geq 70\%$). The class distribution of each dataset is shown in Figure 4.10.

Code Metric-Based Datasets

As with the test generation factors, we have created separate source code metric datasets split by the search budget used for test generation (two minutes per class or ten minutes

per class). Source code metrics do not differ based on the fitness function used in test generation. Therefore, rather than merging all fitness configurations into a single dataset, we have produced separate datasets for each fitness function. We also produced "overall" datasets, classified by whether *any* fitness function or combination detected a fault. In total, this process produced a set of 18 datasets for treatment learning: two based on overall results—split by search budget—and two for each of the eight studied coverage criteria.

In order to learn which metrics predict whether or not a fault is detected, we have added classifications to each characterization dataset. In this case, we have used two classification values—*yes* or *no*, based on whether or not the fault was detected by the generated test suites[5] For each fault, the characterization dataset has an entry for each class fixed as part of patching the fault. We apply a "yes" classification if *any* test suite generated targeting that class, under that search budget, detects the fault. If no test suites detected that fault under that search budget, we apply a "no" classification. The class distributions for the two overall datasets are shown in Figure 4.13. In both cases, the two classes each make up roughly half of the dataset, with a slight edge to the number of "no" classifications.

Accessing Datasets

We have made all datasets used in this study openly available as community resources. They can be downloaded from:

[5]Initially, we used a three-option classification like with the generation factors dataset. However, this did not yield enough examples to yield detailed treatments in many cases. Instead, we elected to use a two-class outcome.

https://github.com/Greg4cr/coverage-exp-data

4.4 Results and Discussion

In Section 4.4, we will outline the basic fault detection capabilities of the generated test suites. Section 4.4 examines the efficacy of each individual fitness function, while Section 4.4 outlines the effect of combining fitness functions. In Section 4.4, we examine the generation factors contributing to fault detection. Finally, in Section 4.4, we examine the source code metrics that impact detection efficacy.

Overall Fault-Detection Capability

In Table 4.3, we list the number of faults detected by each fitness function, broken down by system and search budget. We also list the number of faults detected by *any criterion*, including and excluding the combinations (which we will discuss further in Section 4.4). Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget. Therefore, we list the number of faults found under either budget, as well as the total number of faults detected by each fitness function. These results offer a baseline for further discussion. In our experiments:

The individual criteria are capable of detecting 304 (51.26%) of the 593 faults.
Combinations detect a further 17 faults.

While there is clearly room for improvement, these results are encouraging. The studied faults are actual faults, reported by the users of real-world software projects. The generated tests are able to detect a variety of complex programming issues. Ultimately, our results are consistent with previous studied involving Defects4J—for instance, Shamshiri et al. found that a combination of test generation tools—including suites generated using EvoSuite's

Table 4.3: Number of faults detected by each fitness function. Totals are out of 26 faults (Chart), 133 (Closure), 24 (CommonsCLI), 22 (CommonsCodec), 12 (CommonsCSV), 14 (CommonsJXPath), 9 (Guava), 13 (JacksonCore), 39 (JacksonDatabind), 5 (JacksonXML), 64 (JSoup), 65 (Lang), 102 (Math), 38 (Mockito), 27 (Time), and 593 (Overall).

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsJXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Total
BC	2min	17	16	10	12	10	8	3	10	8	3	21	36	53	4	16	227
	10min	20	19	12	12	10	7	2	9	8	3	23	35	54	4	17	235
	Total	21	21	13	13	11	9	3	10	9	3	25	41	57	4	17	257
DBC	2min	14	16	12	12	11	5	2	9	8	1	20	32	48	4	15	209
	10min	19	19	11	13	10	6	2	10	10	2	22	36	47	4	18	229
	Total	19	22	13	13	11	7	2	10	10	2	26	40	52	4	18	249
EC	2min	8	7	5	4	2	0	0	5	4	1	9	12	13	3	6	79
	10min	10	5	4	3	2	2	0	6	3	1	8	13	12	3	5	77
	Total	10	8	6	4	2	2	0	6	4	1	9	16	15	3	6	92
LC	2min	15	12	7	11	11	4	2	8	9	3	14	31	50	3	15	196
	10min	18	14	10	11	9	8	2	8	10	3	23	32	52	3	14	217
	Total	18	17	11	13	11	8	2	8	11	3	24	37	55	3	15	236
MC	2min	10	6	5	7	5	0	1	5	2	1	7	9	25	2	5	90
	10min	10	10	4	6	4	0	2	4	2	1	7	11	24	2	5	92
	Total	12	10	6	7	5	0	2	5	2	1	8	14	27	2	6	107
MNEC	2min	9	8	5	7	3	1	2	5	5	1	6	10	29	2	5	98
	10min	11	6	2	5	4	2	2	4	4	0	9	13	27	2	3	94
	Total	11	9	5	7	4	2	2	5	5	1	9	13	21	2	6	113
OC	2min	9	7	6	8	2	1	2	4	5	2	7	13	36	2	5	109
	10min	13	9	5	9	2	2	2	3	4	2	8	17	33	2	6	117
	Total	13	12	6	9	2	2	2	4	5	2	9	18	38	2	8	132
WMC	2min	13	15	6	13	8	4	2	10	7	3	19	31	42	3	14	190
	10min	18	19	9	15	8	6	3	9	7	2	22	32	48	3	14	215
	Total	18	22	9	15	8	6	3	10	9	3	25	37	51	3	15	234
Default	2min	15	17	11	14	10	4	2	9	9	1	17	29	48	5	14	205
	10min	17	20	11	14	11	7	2	10	11	1	22	35	57	5	14	237
	Total	18	22	11	14	11	7	2	10	11	1	23	36	59	5	15	245
BC-EC-MC	2min	16	19	10	13	11	5	2	10	9	1	21	38	53	4	15	227
	10min	20	30	12	12	11	6	3	10	9	1	27	40	55	4	21	261
	Total	20	31	12	13	11	6	3	11	10	1	29	41	56	4	21	269
Any criterion (no combinations)	2min	18	23	15	14	11	9	3	10	10	3	31	43	61	5	16	272
	10min	23	29	15	17	11	10	3	10	11	3	33	45	59	5	18	292
	Total	23	31	16	17	11	10	3	10	12	3	37	46	62	5	18	304
Any criterion (w. combinations)	2min	18	24	15	15	11	10	3	10	10	3	33	44	62	5	16	279
	10min	23	37	15	17	11	10	3	11	12	3	35	46	64	5	21	313
	Total	23	37	16	17	11	10	3	11	13	3	39	47	65	5	21	321

branch fitness function—could detect 55.7% of the faults from the original five systems from Defects4J [22].

Shamshiri's work—as well as our studies on Guava [147] and Mockito [21]—offer explanation of the broad reasons why test generation fails to detect particular faults. Some of these reasons include a general inability to gain coverage—particularly over private methods—challenges with initialization of complex data types, and a general lack of the context needed to set up sophisticated series of method and class interactions.

In this work, we are focused on the capabilities and applicability of common fitness functions. In the following subsections, we will assess the results of our study with respect to each research question. In Section 4.4, we compare the capabilities of each fitness function. In Section 4.4, we explore combinations of criteria. In Section 4.4, we explore the generation factors that indicate efficacy or lack of efficacy. Finally, in Section 4.4, we explore the source code metrics that indicate efficacy or lack of efficacy.

Table 4.4: Average likelihood of fault detection, broken down by fitness function, budget, and system. % Change indicates the average gain or loss in efficacy when moving from a two-minute to a ten-minute budget.

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsJXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Overall
BC	2min	45.00 %	4.66 %	26.67 %	33.18 %	57.50 %	25.00 %	17.78 %	50.77 %	14.62 %	60.00 %	18.13 %	34.00 %	27.94 %	9.21 %	34.81 %	22.60 %
	10min	48.46 %	5.79 %	28.33 %	31.36 %	60.83 %	25.00 %	20.00 %	60.00 %	13.08 %	60.00 %	21.72 %	40.15 %	32.75 %	8.42 %	39.26 %	25.24 %
	% Change	7.69 %	24.19 %	6.25 %	-5.48 %	5.80 %	0.00 %	12.50 %	18.18 %	-10.53 %	0.00 %	19.83 %	18.10 %	17.19 %	-8.57 %	12.77 %	11.72 %
DBC	2min	34.23 %	5.11 %	27.50 %	35.91 %	60.00 %	18.57 %	20.00 %	55.38 %	14.62 %	20.00 %	16.56 %	30.00 %	24.51 %	8.16 %	31.11 %	20.62 %
	10min	40.77 %	6.09 %	26.67 %	36.82 %	65.83 %	25.71 %	17.78 %	54.62 %	14.36 %	22.00 %	20.31 %	38.77 %	28.63 %	8.42 %	40.37 %	23.88 %
	% Change	19.10 %	19.12 %	-3.03 %	2.53 %	9.72 %	38.46 %	-11.11 %	-1.39 %	-1.75 %	10.00 %	22.64 %	29.23 %	16.80 %	3.23 %	29.76 %	15.78 %
EC	2min	22.31 %	1.35 %	5.83 %	12.27 %	16.67 %	0.00 %	0.00 %	20.00 %	3.33 %	20.00 %	6.41 %	7.54 %	6.37 %	6.05 %	9.26 %	6.56 %
	10min	21.54 %	0.98 %	5.83 %	12.27 %	16.67 %	1.43 %	0.00 %	22.31 %	1.79 %	20.00 %	5.94 %	9.23 %	7.06 %	5.26 %	9.63 %	6.64 %
	% Change	-3.45 %	-27.78 %	0.00 %	0.00 %	0.00 %	- %	- %	11.54 %	-46.15 %	0.00 %	-7.32 %	22.45 %	10.77 %	-13.04 %	4.00 %	1.29 %
LC	2min	38.85 %	4.14 %	21.25 %	22.73 %	59.17 %	20.71 %	21.11 %	51.54 %	15.13 %	60.00 %	12.03 %	31.23 %	25.78 %	5.79 %	30.00 %	19.87 %
	10min	46.15 %	4.81 %	22.08 %	21.36 %	50.00 %	25.71 %	20.00 %	53.08 %	17.95 %	56.00 %	17.50 %	34.31 %	29.22 %	5.79 %	36.67 %	22.24 %
	% Change	18.81 %	16.36 %	3.92 %	-6.00 %	-15.49 %	24.14 %	-5.26 %	2.99 %	18.64 %	-6.67 %	45.45 %	9.85 %	13.31 %	0.00 %	22.22 %	11.97 %
MC	2min	30.77 %	1.58 %	9.17 %	16.36 %	18.33 %	0.00 %	11.11 %	25.38 %	0.77 %	4.00 %	7.34 %	7.54 %	10.98 %	0.53 %	8.15 %	7.77 %
	10min	30.77 %	2.26 %	7.92 %	12.27 %	16.67 %	0.00 %	12.22 %	23.85 %	1.03 %	8.00 %	6.56 %	7.69 %	10.88 %	1.58 %	8.15 %	7.71 %
	% Change	0.00 %	42.86 %	-13.64 %	-25.00 %	-9.09 %	- %	10.00 %	-6.06 %	33.33 %	100.00 %	-10.64 %	2.04 %	-0.89 %	200.00 %	0.00 %	-0.87 %
MNEC	2min	23.46 %	2.18 %	9.17 %	14.55 %	12.50 %	0.71 %	11.11 %	20.00 %	4.10 %	2.00 %	7.50 %	6.62 %	12.16 %	1.05 %	6.67 %	7.59 %
	10min	30.77 %	1.88 %	7.50 %	15.00 %	12.50 %	4.29 %	12.22 %	24.62 %	5.90 %	0.00 %	7.50 %	7.54 %	12.06 %	0.79 %	5.19 %	8.09 %
	% Change	31.15 %	-13.79 %	-18.18 %	3.12 %	0.00 %	500.00 %	10.00 %	23.08 %	43.75 %	-100.00 %	0.00 %	13.95 %	-0.81 %	-25.00 %	-22.22 %	6.67 %
OC	2min	1.15 %	2.03 %	14.17 %	24.55 %	10.00 %	0.71 %	14.44 %	8.46 %	7.95 %	40.00 %	5.31 %	7.85 %	16.57 %	3.68 %	9.63 %	9.31 %
	10min	23.85 %	2.56 %	16.25 %	25.00 %	10.83 %	2.14 %	18.89 %	13.08 %	7.69 %	40.00 %	5.00 %	10.92 %	16.76 %	2.89 %	12.22 %	10.25 %
	% Change	12.73 %	25.93 %	14.71 %	1.85 %	8.33 %	200.00 %	30.77 %	54.55 %	-3.23 %	0.00 %	-5.88 %	39.22 %	1.18 %	-21.43 %	26.92 %	10.14 %
WMC	2min	38.08 %	4.44 %	19.17 %	31.36 %	41.67 %	15.00 %	16.67 %	44.62 %	8.97 %	26.00 %	15.00 %	24.15 %	23.04 %	5.79 %	25.19 %	17.59 %
	10min	46.15 %	5.56 %	20.00 %	33.64 %	45.00 %	17.86 %	18.89 %	42.31 %	7.69 %	18.00 %	18.28 %	32.15 %	27.45 %	5.53 %	27.04 %	20.34 %
	% Change	21.21 %	25.42 %	4.35 %	7.25 %	8.00 %	19.05 %	13.33 %	-5.17 %	-14.29 %	-30.77 %	21.88 %	33.12 %	19.15 %	-4.55 %	7.35 %	15.63 %
Default	2min	47.31 %	4.51 %	32.08 %	44.09 %	52.50 %	16.43 %	17.78 %	47.69 %	14.10 %	20.00 %	15.63 %	23.85 %	25.78 %	11.84 %	25.93 %	20.56 %
	10min	48.08 %	7.07 %	34.58 %	45.91 %	50.00 %	21.43 %	20.00 %	52.31 %	16.15 %	20.00 %	20.94 %	32.62 %	32.84 %	10.79 %	33.33 %	24.69 %
	% Change	1.63 %	56.67 %	7.79 %	4.12 %	-4.76 %	30.43 %	12.50 %	9.68 %	14.55 %	0.00 %	34.00 %	36.77 %	27.38 %	-8.89 %	28.57 %	20.10 %
BC-EC-MC	2min	43.08 %	5.64 %	30.42 %	37.27 %	60.00 %	20.71 %	17.78 %	50.00 %	14.87 %	20.00 %	19.38 %	40.46 %	30.39 %	10.26 %	35.93 %	24.03 %
	10min	53.85 %	8.05 %	30.83 %	38.18 %	50.00 %	26.43 %	21.11 %	56.15 %	14.62 %	20.00 %	25.00 %	48.15 %	34.31 %	10.53 %	47.04 %	27.84 %
	% Change	25.00 %	42.67 %	1.37 %	2.44 %	-16.67 %	27.59 %	18.75 %	12.31 %	-1.72 %	0.00 %	29.03 %	19.01 %	12.90 %	2.56 %	30.93 %	15.86 %

Comparing Fitness Functions

From Table 4.3, we can see that suites differ in effectiveness between criteria. Overall, branch coverage outperforms the other criteria, detecting 257 faults. Branch is closely followed by direct branch (249 faults), line coverage (236), and weak mutation coverage (234). These four fitness functions are trailed by the other four, with exception coverage showing the weakest results (92 faults). These rankings do not differ much on a per-system basis. At times, ranks may shift—for example, direct branch coverage occasionally outperforms branch coverage—but we can see two clusters form among the fitness functions. The first cluster contains branch, direct branch, line, and weak mutation coverage—with branch and direct branch leading the other two. The second cluster contains exception, method, method (no-exception), and output coverage—with output coverage producing the best results and exception coverage producing the worst.

Due to the stochastic nature of the search, one suite generated by EvoSuite may not always detect a fault detected by another suite—even if the same criterion is used. To more clearly understand the effectiveness of each fitness function, we must not track only whether a fault was detected, but how *reliably* it is detected. We are interested in the likelihood of detection—if a fresh suite is generated, how likely is it to detect a particular

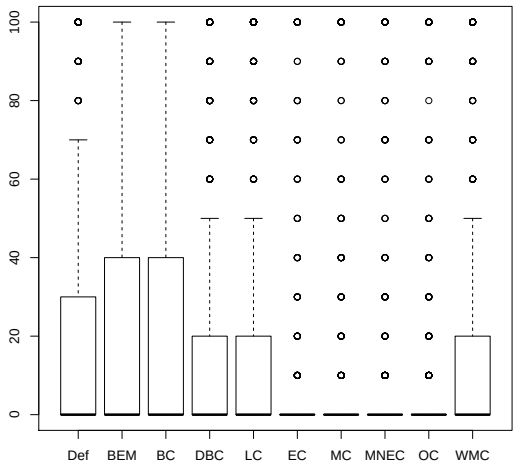


Figure 4.14: 2m Budget

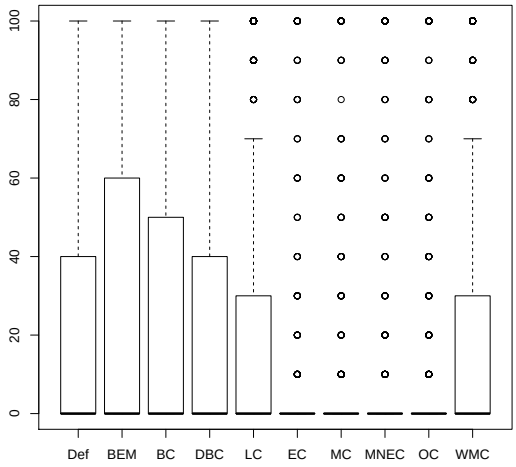


Figure 4.15: 10m Budget

Figure 4.16: Boxplots of the likelihood of detection for each fitness function and combination. “Def” = Default Combination, “BEM” = BC-EC-MC Combination.

fault. To measure the likelihood, we record the proportion of detecting suites to the total number of suites generated for that fault. The average likelihood of fault detection is listed for each criterion, by system and budget, in Table 4.4. Figure 4.16 shows boxplots of the likelihood of detection for each fitness function and combination of functions.

We largely observe the same trends as above. Branch coverage has the highest overall

likelihood of fault detection, with 22.60% of suites detecting faults given a two-minute search budget and 25.24% of suites detecting faults given a ten-minute budget. Direct branch coverage and line coverage follow with a 20.62-23.88% and 19.87-22.24% success rate, respectively. While the effectiveness of each criterion varies between system—direct branch outperforms all other criteria for Closure, for example—the two clusters noted above remain intact. Branch, line, direct branch, and weak mutation coverage all perform well, with the edge generally going to branch coverage. On the lower side of the scale, output, method, method (no exception), and exception coverage perform similarly, with a slight edge to output coverage.

The boxplots in Figure 4.16 echo these results. All methods have medians near 0%, reflecting that many faults are not detected. However, for both budgets, branch coverage has a larger upper quartile than other fitness functions—indicating a large variance in results, but also that branch coverage yields more suites with a higher likelihood of detection than other methods. At the two-minute mark, it has a higher upper whisker as well. Direct branch coverage, line coverage, and weak mutation coverage follow in terms of third quartile size and upper whisker.

Branch coverage is the most effective criterion, detecting 257 faults. Branch coverage suites have, on average, a 22.60-25.24% likelihood of fault detection (2min/10min budget).

From Table 4.4, we can see that almost all criteria benefit from an increased search budget. Direct branch coverage and weak mutation benefit the most, with average improvements of 15.78% and 15.63% in effectiveness. In particular, it is reasonable that direct branch coverage benefits more than traditional branch coverage. In traditional branch coverage, branches executed through indirect chains of calls to program methods contribute to the total coverage. In direct branch coverage, only calls made directly by the test cases

count towards coverage. Therefore, the test generator requires more time, and more method calls, to attain the same level of branch coverage. As a result, direct branch coverage attains slightly worse results than traditional branch coverage given the same time budget.

In general, all distance-driven criteria—branch, direct branch, line, weak mutation, and, partially, output coverage—improve given more time. These criteria all have complex, informative fitness functions that are able to guide the search process. Discrete fitness functions, such as those used by method coverage or exception coverage, benefit less from the budget increase. In such cases, the fitness function is unable to guide the search towards better solutions. More time is of benefit—as the generator can make more guesses. However, such time is not guaranteed to be beneficial, and does not necessarily result in improved test suites.

Distance-based functions benefit from an increased search budget, particularly direct branch and weak mutation.

Further increases in generation time beyond ten minutes may yield further improvements in the likelihood of fault detection. However, there is likely to be a plateau due to test obligations that the generator cannot satisfy, due to limitations in coverage of private code or manipulation of complex objects. Further, if test generation requires more time than it takes for a human to write tests, then the benefits of automation are more limited. Therefore, there is likely to be a limit to the gain from increasing the search budget.

We can perform statistical analysis to assess our observations. For each pair of criteria, we formulate hypothesis H and its null hypothesis, $H0$:

- H: Given a fixed search budget, test suites generated using criterion *A* will have a different distribution of likelihood of fault detection results than suites generated using criterion *B*.

Table 4.5: P-values for Nemenyi comparisons of fitness functions (two-minute search budget). Cases where we can reject the null hypothesis are **bolded**.

	Default Combination	**BC**	**DBC**	**EC**	**LC**	**MC**	**MNEC**	**OC**	**WMC**
BC	0.87	-	-	-	-	-	-	-	-
DBC	1.00	0.97	-	-	-	-	-	-	-
EC	**< 0.01**	**< 0.01**	**< 0.01**	-	-	-	-	-	-
LC	1.00	0.59	1.00	**< 0.01**	-	-	-	-	-
MC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	-	-	-	-
MNEC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	1.00	-	-	-
OC	**< 0.01**	**< 0.01**	**< 0.01**	0.77	**< 0.01**	0.96	0.94	-	-
WMC	0.91	0.07	0.75	**< 0.01**	0.99	**< 0.001**	**< 0.01**	**< 0.01**	-
BC-EC-MC Combination	0.56	0.99	0.79	**< 0.01**	0.27	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**

Table 4.6: P-values for Nemenyi comparisons of fitness functions (ten-minute search budget). Cases where we can reject the null hypothesis are **bolded**.

	Default Combination	**BC**	**DBC**	**EC**	**LC**	**MC**	**MNEC**	**OC**	**WMC**
BC	1.00	-	-	-	-	-	-	-	-
DBC	1.00	1.00	-	-	-	-	-	-	-
EC	**< 0.01**	**< 0.01**	**< 0.01**	-	-	-	-	-	-
LC	0.95	0.71	1.00	**< 0.01**	-	-	-	-	-
MC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	-	-	-	-
MNEC	**< 0.01**	**< 0.01**	**< 0.01**	1.00	**< 0.01**	1.00	-	-	-
OC	**< 0.01**	**< 0.01**	**< 0.01**	0.50	**< 0.01**	0.90	0.95	-	-
WMC	0.73	0.38	0.92	**< 0.01**	1.00	**< 0.01**	**< 0.01**	**< 0.01**	-
BC-EC-MC Combination	0.47	0.81	0.23	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**

- $H0$: Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate $H0$ without any assumptions on distribution, we use the Friedman non-parametric alternative to the parametric repeated measures ANOVA [165]. Due to the limited number of faults for several systems, we have analyzed results across the combination of all systems. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

At both budgets, the Friedman test confirms with p-value < 0.001 that the results for all fitness functions are not drawn from the same distribution. To differentiate and rank methods, we apply the post-hoc Nemenyi test in order to assess all pairs of fitness functions. The resulting p-values are listed in Tables 4.5-4.6.

The results of these tests further validate the "two clusters" observation. For the four

Table 4.7: Number of faults uniquely detected by each suites generated using each fitness function (with and without considering combinations) for each budget, and for the combination of budgets.

	Two-Minute Budget		Ten-Minute Budget		All Budgets	
Function	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**	**Number of Faults (W. Combinations)**	**Number of Faults (No Combinations)**
Branch Coverage	5	13	2	7	0	1
Direct Branch Coverage	4	6	3	6	0	1
Exception Coverage	3	4	3	5	1	2
Line Coverage	2	5	3	4	0	0
Method Coverage	0	0	0	1	0	0
Method, No Exception	0	1	1	1	0	0
Output Coverage	1	3	2	7	1	2
Weak Mutation Coverage	3	6	4	7	0	0
Default Combination	2	-	5	-	0	-
BC-EC-MC Combination	4	-	13	-	1	-

criteria in the top cluster—branch, direct branch, line, and weak mutation coverage—we can always reject the null hypothesis with regard to the remaining four criteria in the bottom cluster. This is also true in the opposite direction. The performance of the four criteria in the bottom cluster—exception, method, MNEC, and output coverage—is drawn from a different distribution to the criteria in the other cluster. Within each cluster, we usually fail to reject the null hypothesis.

Branch, direct branch, line, and weak mutation coverage outperform, with statistical significance, method, MNE, output, and exception coverage (both budgets).

Another way to consider performance is—regardless of overall performance—to look at whether a criterion leads to suites that detect faults that other criteria would not detect. Table 4.7 depicts the number of faults uniquely detected by each fitness function for each search budget, then the number of faults uniquely detected regardless of budget. Results are listed when combinations are considered, which we will discuss in Section 4.4, and when only considering the individual criteria.

From these results, we can see that almost all criteria clearly have *situational applicability*—that is, there are situations where their use leads to the detection of faults missed by other criteria. At both search budgets, a total of 38 faults are detected by a single criterion.

Most interestingly, there are six faults that are—regardless of search budget—only detected by a single criterion.

The general efficacy of branch and direct branch coverage clearly can still be seen here, where each detects one fault that nothing else can detect, regardless of budget. However, criteria like exception coverage or output coverage—which have low average performance—can also detect faults that no other criterion can expose. Both criteria detect two faults, regardless of budget, that nothing else can catch. At each independent budget level, exception, output, and weak mutation coverage each detect a number of faults that other criteria miss. This is especially worth noting at the ten-minute budget level, where suites generated to satisfy output and weak mutation coverage detect as many unique faults as suites generated to satisfy branch coverage.

These results suggest that—regardless of absolute efficacy—each criterion results in *different* test suites, each of which exercise the code under test in a distinct manner. Even if a criterion is not universally effective, it offers some form of situational applicability where it could be considered for use, and where it may have some value as part of a portfolio of testing tools.

For example, consider fault 100 for the Math project[6]. To address this fault, an estimation method switches from getting all parameters to only getting unbound parameters. Tests generated for exception coverage cause an exception by passing in a parameter with no measurements—i.e., an unbound parameter. This exception, even if triggered, is not retained by any other fitness function. Exception coverage, by prioritizing exceptions, ensures that the observed failure is retained and passed along to testers.

Output coverage is focused on coverage of abstract value classes for particular types of function output. It is particularly well-defined for numeric types. This makes it well-suited to discovering faults related to such numeric data types. Consider Mockito fault 26.

[6] `https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Math/patches/100.src.patch`

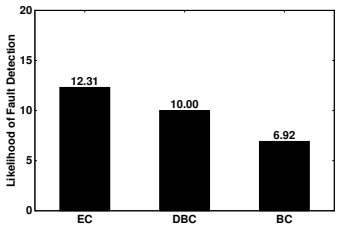
This fault[7] lies in the Mockito framework's code for replicating Java's primitive datatypes. Illegal casts can be made from `integer` to other primitive types. Output coverage is able to trigger this fault by illegally casting `integer` variables to `double` variables. This *contextual* use of the class is not suggested by code coverage, and is not attempted by any of the other criteria.

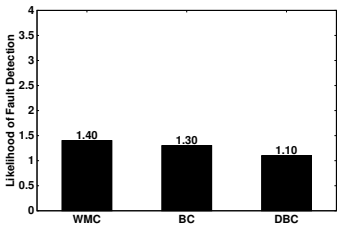
To further understand the situational applicability of criteria, we filter the set of faults for those that the top-scoring criterion is ineffective at detecting. In Figure 4.27, we have taken the faults for five of the systems (Chart, Closure, Lang, Math, and Time), isolated any where the "best" criterion for that system (generally branch coverage, see Table 4.4) has $< 30\%$ likelihood of detection, and calculated the likelihood of fault detection for each criterion for that subset of the faults. In each subplot, we display the average likelihood of fault detection over the subset for any criterion that outperforms the best from the full set.

From these plots, we can see that there are always two-to-four criteria that are more effective in these situations. The exact criteria depend strongly on the system, and likely, on the types of faults examined. However, we frequently see the criteria mentioned above—including exception, weak mutation, and output coverage. Interestingly, despite the similarity in distance functions and testing intent, direct branch and line coverage are often more effective than branch coverage in situations where it has a low chance of detection. In these cases, the criteria drive tests to interact in such a way with the CUT that they are better able to detect the fault. The efficacy of alternative criteria in situations where the overall top performer offers poor results further emphasizes that:

> Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults.

[7] `https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Mockito/patches/26.src.patch`





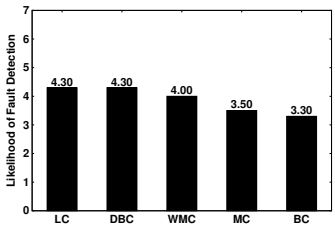
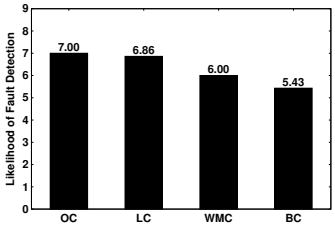


Figure 4.17: Chart (2m) Figure 4.18: Closure (2m) Figure 4.19: Lang (2m)



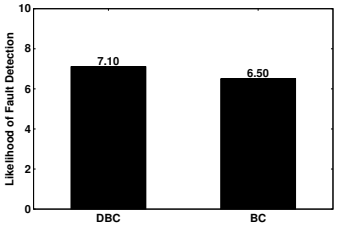
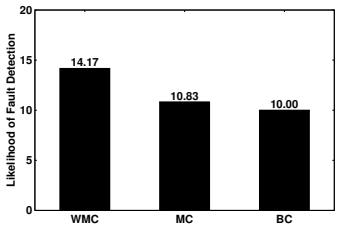
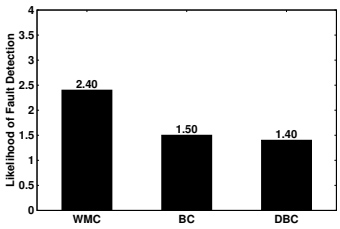


Figure 4.20: Math (2m) Figure 4.21: Time (2m)





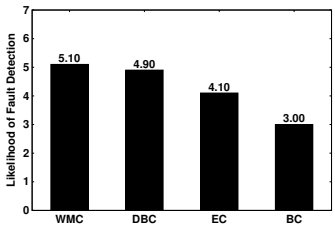
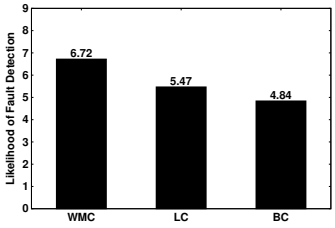


Figure 4.22: Chart (10m) Figure 4.23: Closure (10) Figure 4.24: Lang (10m)



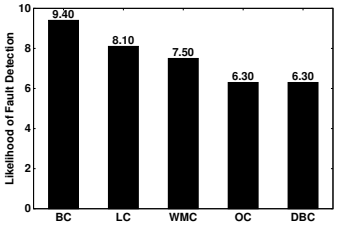


Figure 4.25: Math (10m) Figure 4.26: Time (10m)

Figure 4.27: Average % likelihood of fault detection for fitness functions once data is filtered for faults where the most effective function for that system has $< 30\%$ chance of detection.

Given its strong overall performance, we would recommend that practitioners prioritize branch coverage—of the studied options—when generating test suites. However, we also stress that other criteria should not be ignored. Several—including exception and output coverage—can be quite effective at times, even if they are not effective on average. More research is clearly needed to codify the situations where such criteria can be effective

and should be employed. Alternatively, such criteria could be used in *combination* with generally effective criteria such as branch coverage.

Combinations of Fitness Functions

The analysis above presupposes that only one fitness function can be used to generate test suites. However, many search-based generation algorithms can simultaneously target multiple fitness functions. EvoSuite's default configuration, in fact, attempts to satisfy all eight of the fitness functions examined in this study [159]. In theory, suites generated through a combination of fitness functions could be more effective than suites generated through any one objective because—rather than focusing exclusively on one goal—they can simultaneously target multiple facets of the class under test.

For example, combining exception and branch coverage may result in a suite that both thoroughly explores the structure of the system (due to the branch obligations) and rewards tests that throw a larger number of exceptions. Such a suite may be more effective than a suite generated using branch or exception coverage alone. In fact, rather than generating tests for multiple independent criteria—when one or more of those criteria may only be effective situationally—a tester could, in theory, simultaneously generate for a combination of criteria in an attempt to produce suites effective in all situations.

To better understand the potential of combined criteria, we have generated tests for two combinations. The first is EvoSuite's default combination of the eight criteria that were the focus of this study. The second is a more lightweight combination of branch, exception, and method coverage—a combination of the most effective criterion overall with two that were selectively effective. In a recent study on combination of criteria on a subset of the Defects4J database, the BC-EC-MC combination was suggested as an effective baseline for new, untested systems [137].

Overall, EvoSuite's default configuration performs well, but fails to outperform all individual criteria in most situations. Table 4.3 shows that the default configuration detects

245 faults—fewer than branch and direct branch coverage, but more than the other individual criteria. It also uniquely detects two faults at the two-minute budget level and five at the ten-minute level (see Table 4.7). At the two minute level, this is worse than several individual criteria, but at the ten-minute level, it finds more faults than any individual criterion. According to Table 4.4, the default configuration's average overall likelihood of fault detection is 20.56% (2m budget)-24.69% (10m budget). At the two-minute level, this places it below branch, and direct branch coverage. At the ten-minute level, it falls below branch coverage, but above all other criteria. This places the default configuration in the top cluster—an observation confirmed by statistical tests (Tables 4.5 and 4.6).

However, it fails to outperform branch coverage in almost all situations. In theory, a combination of criteria should be able to detect more faults than any single criterion. In practice, combining *all* criteria results in suites that are fairly effective, but fail to reliably outperform individual criterion. The major reason for the less reliable performance of this configuration is the difficulty in attempting to satisfy so many obligations at once. As noted in Table 4.2, the default configuration must attempt to satisfy, on average, 1681 obligations. The individual criteria only need to satisfy a fraction of that total. As a result, the default configuration also benefits more than any individual criterion from an increased search budget—a 20.10% improvement in efficacy.

EvoSuite's default configuration has an average 20.56-24.69% likelihood of fault detection—in the top cluster, but failing to outperform all individual criteria.

Our observations imply that combinations of criteria could be more effective than individual criteria. However, a combination of all eight criteria results in unstable performance—especially if search budget is limited. Instead, testers may wish to identify a smaller, more targeted subset of criteria to combine during test generation. In fact, we can see the wisdom in such an approach by examining the results for the focused BC-EC-MC combination.

From Table 4.3, we can see that the BC-EC-MC combination finds a total of 269 faults—more than any individual criterion. From Table 4.4, this combination has a 24.03% (two-minute) to 27.84% (ten-minute) likelihood of detection. Again, this is better than any of the individual criteria. This combination also detects four faults uniquely at the two-minute budget, 13 at the ten-minute budget, and one fault regardless of the budget.The boxplots in Figure 4.16 show that the BC-EC-MC combination has a higher third-quartile box than any other method, indicating that it returned more results in that range than other methods. Statistical tests place this configuration in the top cluster, where it also outperforms weak mutation coverage with significance (Tables 4.5 and 4.6).

There are still situations where this combination can be outperformed by an individual criterion, particularly at the two-minute budget level. This combination requires fewer obligations than the eight-way default combination—around 354, on average—but still more than any individual criterion. This means that the combination benefits from a larger search budget (15.86% average improvement), and offers more stable performance at higher budgets. Still, this combination is clearly quite effective.

Of the studied criteria, exception coverage is unique in that it does not prescribe static test obligations. Rather, it simply rewards suites that cause more exceptions to be thrown. This means that it can be added to a combination with little increase in search complexity. The simplicity of exception coverage explains its poor performance as the *primary* criterion. It lacks a feedback mechanism to drive generation towards exceptions. However, exception coverage appears to be very effective *when paired with criteria that effectively explore the structure of the CUT*. Branch coverage gives exception coverage the feedback mechanism it needs to explore the code. Adding exception coverage to branch coverage adds little cost in terms of generation difficulty, and generally outperforms the use of branch coverage alone.

An example of effective combination can be seen in fault 60 for Lang[8]—a case where

[8] `https://github.com/apache/commons-lang/commit/a8203b65261110c4a30ff69fe0da7a2390d82757`.

two methods can look beyond the end of a string. No single criterion is effective, with a maximum of 10% chance of detection given a two-minute budget and 20% with a ten-minute budget. However, combining branch and exception coverage boosts the likelihood of detection to 40% and 90% for the two budgets. In this case, if the fault is triggered, the incorrect string access will cause an exception to be thrown. However, this only occurs under particular circumstances. Therefore, exception coverage alone never detects the fault. Branch coverage provides the necessary means to drive program execution to the correct location. However, two suites with an equal coverage score are considered equal. Branch coverage alone may prioritize suites with slightly higher (or different) coverage, missing the fault. By combining the two, exception-throwing tests are prioritized and retained, succeeding where either criterion would fail alone.

Method coverage adds another "low-cost" boost. In general, a class will not have a large number of methods, and methods are either covered or not covered. Thus, even if method coverage is not a particularly helpful addition to a combination, its inclusion does not substantially increase the number of obligations that the test generator is tasked with fulfilling. An example where the addition of method coverage boosts efficacy can be seen in Lang fault 34[9]. This fault resides in two small (1-2 line) methods. Calling either method will reveal the fault, but branch coverage alone can easily overlook them because their invocation does not substantially improve branch coverage of the class as a whole. The addition of method coverage adds a useful "reminder" for the generator to invoke these simple methods.

A combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds

[9] https://github.com/apache/commons-lang/commit/496525b0d626dd5049528cdef61d71681154b660

lightweight situationally-applicable criteria to a strong, coverage-focused criterion.

It is unlikely that the BC-EC-MC combination is the strongest possible combination, and we can see some situations where a single criterion is still the most effective. In fact, it is unlikely that any one criterion or combination of criteria will ever universally be the "best". The most effective criteria depend on the type of system under test, and the types of faults that the developers have introduced into the code. Still, there is a powerful idea at the heart of this combination. When generating tests, a strong coverage-focused criterion should be selected as the primary criterion. Then, a small number of targeted, orthogonal criteria can be added to that primary criterion. More investigation is needed into the situational applicability of criteria in order to better understand when any one criterion or a combination of criteria will be effective.

Understanding the Generation Factors Impacting Fault Detection

Using the TAR3 treatment learner [164], we have generated five treatments from each of the two generation factor datasets for each of the three classifications ("Not Detected", "Low Performance", and "High Performance"). The treatments are scored according to their impact on class distribution, and top-scoring treatments are presented first.

First, the following treatments indicate the factors pointing most strongly to a "high" likelihood of fault detection:

Two-Minute Dataset:

1. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$
2. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, LC (faulty) $> 88.09\%$
3. LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$
4. BC (fixed) $> 79.71\%$, % of obligations satisfied $> 89.59\%$, LC (faulty) $> 88.09\%$
5. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$, LC (faulty) $> 88.09\%$

Ten-Minute Dataset:

1. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (faulty) $> 92.08\%$
2. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (fixed) $> 92.23\%$
3. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, LC (faulty) $> 92.08\%$, LC (fixed) $> 92.23\%$
4. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
5. BC (faulty) $> 86.25\%$, % of obligations satisfied $> 94.34\%$, BC (fixed) $> 85.97\%$, LC (faulty) $> 92.08\%$

In Figure 4.30, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets. Comparing the plots in Figure 4.10 to the subsets in Figures 4.30, we can see that the treatments do impose a large change in the class distribution—a lower percentage of cases have the "Not Detected" class, and more have the other classifications. This shows that the treatments do reasonably well in predicting for success. Test suites fitting these treatments are not guaranteed to be successful, but are likely to be.

Note that some treatments are subsets of other treatments. For example, the third treatment for the two-minute dataset above is a subset of the top treatment. Each treatment indicates a set of attributes and value ranges for those attributes that, when applied *together*, tend to lead to particular outcomes. A smaller treatment that is a subset of a larger treatment, when applied, will lead to a different subset of the overall data set with a different class distribution to the larger treatment. In general, smaller treatments are easier for humans to understand—this is why we have limited treatment size to five attributes. However, within that limit, a larger treatment may have more explanatory power than a smaller treatment. In this case, as treatments are ranked by score, the larger treatment is more indicative of the target class and the additional factors offer additional explanatory power.

We can make several observations. First, the most common factors selected as indicative of efficacy are all coverage-related factors. Even if their goal is not to attain coverage, successful suites thoroughly explore the structure of the CUT. The fact that coverage is

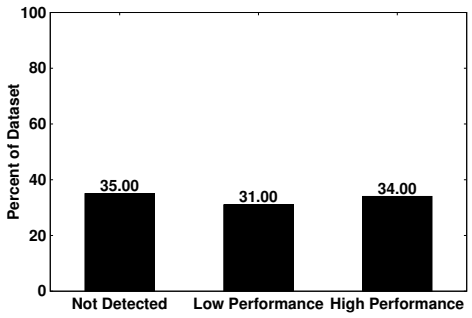


Figure 4.28: 2m Budget

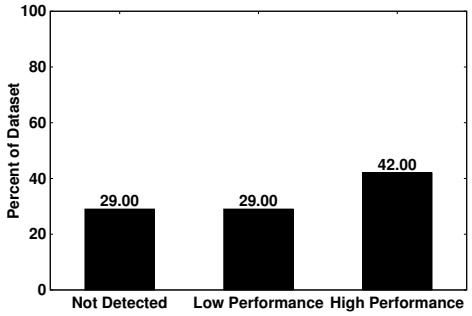


Figure 4.29: 10m Budget

Figure 4.30: Class distributions of the data subsets fitting the top treatments learned from each dataset for the "High Performance" class.

important is not, in itself, entirely surprising—if patched code is not well covered, the fault is unlikely to be discovered.

More surprising is how much weight is given to coverage. Suite size has been a focus in recent work, with Inozemtseva et al. (and others) finding that the size has a stronger correlation to efficacy than coverage level [135]. However, size attributes—number of tests and test length—do not appear in any of the generated treatments. Kendall correlation tests further reinforce this point. For both budgets and for both suite size and length, correlation strengths were all approximately 0.25—"weak to low" correlations.

However, rather than indicating that larger test suites are not necessarily more effective at detecting real faults, it is important to look at the suites themselves. As can be seen in Section 4.3, test suite sizes do not range dramatically between each of the fitness configurations. Within the size ranges of suites in this study, larger suites also do not necessarily

outperform smaller suites. Suites for Output Coverage, which performs poorly on average, are often similar in size to those yielded for the higher-scoring fitness functions. The suites for the combinations are, naturally, the largest. However, the largest suites belong to the "default" combination—which is often outperformed by branch coverage and the BC-EC-MC combination. Exception Coverage, the poorest performing fitness function on average, does have the smallest test suites. However, other factors, such as its low coverage of source code, seem to play a larger role in determining suite efficacy than size alone.

The other factor noted as indicative of efficacy is the percent of obligations satisfied. This too seems reasonable. If a suite covers more of its test obligations, it will be better at detecting faults. For coverage-based fitness functions like branch and line coverage, a high level of satisfied obligations naturally correlates with a high level of branch or line coverage. For other fitness functions, the correlation may not be as strong, but it is also likely that suites satisfying more of their obligations also tend to explore more of the structure of the CUT.

Factors that strongly indicate a high level of efficacy include line or branch coverage over either version of the code and high coverage of their own test obligations. Coverage and obligation satisfaction are favored over factors related to suite size or test obligations.

Note, however, that we still do not entirely understand the factors that indicate a high probability of fault detection. From Figure 4.30, we can see that the treatments radically alter the class distribution from the baseline in Figure 4.10. Still, we can also see that suites from the "Not Detected" class still form a significant portion of that class distribution. From this, we can conclude that factors predicted by treatments are a necessary precondition for a high likelihood of fault detection, but are not sufficient to ensure that faults are detected. Unless code is executed, faults are unlikely to be found. Thus, cov-

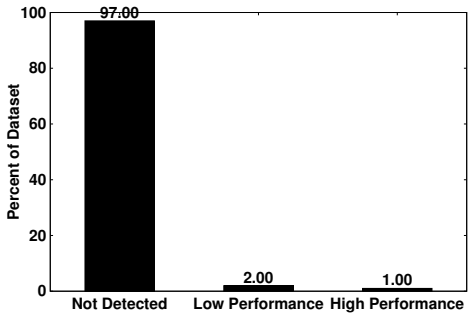


Figure 4.31: 2m Budget

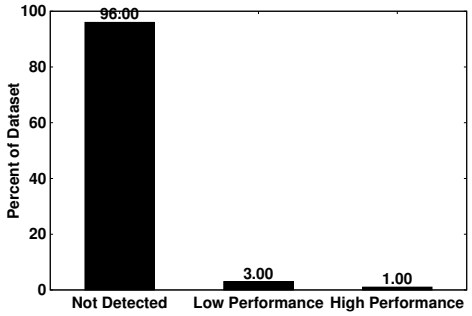


Figure 4.32: 10m Budget

Figure 4.33: Class distributions of the data subsets fitting the top treatments learned from each dataset for the "Not Detected" class.

erage is impotent. However, how code is executed matters, and execution alone does not guarantee that faults are triggered and observed as failures. The fitness function determines how code is executed. It may be that fitness functions based on stronger adequacy criteria (such as complex condition-based criteria [166]) or combinations of fitness functions will better guide such a search. While coverage increases the likelihood of fault detection, it does not ensure that suites are effective.

To better understand factors indicating success, we can also perform treatment learning for the opposite scenario—what indicates that we will *not* detect a fault? Factors that indicate a lack of success include:

Two-Minute Dataset:

1. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$, BC (faulty) $\leq 8.96\%$
2. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$, BC (faulty) $\leq 8.96\%$, BC (fixed) $\leq 9.33\%$
3. LC (faulty) $\leq 11.77\%$, LC (fixed) $\leq 12.41\%$

4. LC (faulty) $\leq 11.77\%$
5. LC (faulty) $\leq 11.77\%$, BC (faulty) $\leq 8.96\%$

Ten-Minute Dataset:

1. LC (fixed) $\leq 15.02\%$, BC (fixed) $\leq 12.12\%$
2. LC (fixed) $\leq 15.02\%$, BC (faulty) $\leq 11.91\%$
3. LC (fixed) $\leq 15.02\%$, BC (faulty) $\leq 11.91\%$, BC (fixed) $\leq 12.12\%$
4. BC (fixed) $\leq 12.12\%$
5. LC (fixed) $\leq 15.02\%$, LC (faulty) $\leq 14.54\%$

In Figure 4.33, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets for the "Not Detected" outcome. Comparing the plots in Figure 4.10 to the subsets in Figures 4.30, we can see a dramatic change in the class distribution. These treatments predict quite clearly a lack of success, with almost no data records from the other classes still matching the treatment.

The factors indicating a lack of success are entirely coverage-based. If coverage—line or branch—is less than approximately 15%, then the odds of effective fault detection are extremely low. This further reinforces the discussion above:

While coverage may not ensure success, it is a prerequisite. If the code is not exercised, then the fault will not be found.

Finally, we can examine one additional classification—"low" efficacy. What factors differentiate situations where a faulty is highly likely to be found from situations where it is still generally found, but with a low likelihood of detection? The factors that suggest this situation include:

Two-Minute Dataset:

1. LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$
2. LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$
3. BC (faulty) $> 79.85\%$, BC (fixed) $> 79.71\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$

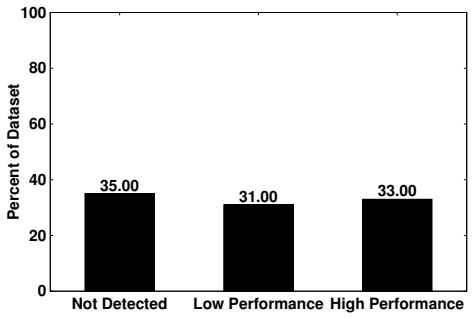


Figure 4.34: 2m Budget

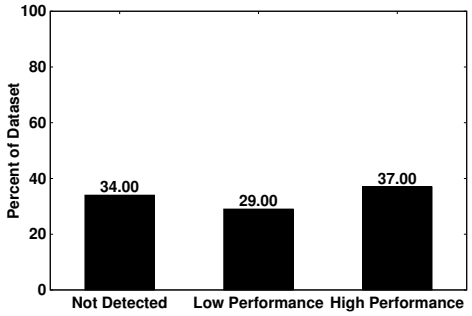


Figure 4.35: 10m Budget

Figure 4.36: Class distributions of the data subsets fitting the top treatments learned from each dataset for the "Low Performance" class.

4. BC (fixed) $> 79.71\%$, LC (fixed) $> 87.89\%$, % of obligations satisfied $> 89.59\%$, BC (faulty) $> 79.85\%$, LC (faulty) $> 88.09\%$
5. BC (fixed) $> 79.71\%$, LC (faulty) $> 88.09\%$, % of obligations satisfied $> 89.59\%$, LC (fixed) $> 87.89\%$

Ten-Minute Dataset:

1. LC (faulty) $> 92.08\%$, BC (fixed) $> 85.97\%$
2. LC (faulty) $> 92.08\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
3. BC (faulty) $> 86.25\%$, BC (fixed) $> 85.97\%$, LC (fixed) $> 92.23\%$
4. BC (faulty) $> 86.25\%$, BC (fixed) $> 85.97\%$, LC (faulty) $> 92.08\%$
5. BC (faulty) $> 86.25\%$, LC (fixed) $> 92.23\%$

These treatments, and their resulting class distributions—illustrated in Figure 4.36—are very similar to the factors predicting "high" performance. This is particularly true for the two-minute dataset, where we simply see a small downgrade in the number of "high"

cases. From this, we can again see that coverage is needed to detect faults, but more data is needed to help ensure reliable detection.

However, we can make one interesting observation from the treatments learned from the ten-minute dataset. The ten-minute dataset includes more effective test suites from the start, allowing us to better differentiate high efficacy from low—but extant—efficacy. The treatments learned from the ten-minute dataset for "low efficacy" are lacking any reference to satisfaction of their obligations—a factor that is always present in the treatments learned for the "high efficacy" classification. We can also see a shift in the resulting class distribution in Figure 4.36 from that in Figure 4.30. The percent of "low" efficacy examples remains the same, but there are fewer "high" cases and more "not detected" cases.

The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations.

Thus, we can observe that the most effective test suites are those that both cover a large percentage of their own obligations and thoroughly exercise the targeted code. In the case of criteria like branch coverage, these two go hand-in-hand. However, this also illustrates why criteria based on orthogonal factors to code coverage—like exceptions—tend to be best used in combination with coverage-based criteria. In future work, we will further explore such factors and others, and investigate how to best ensure effective test suite generation.

Understanding the Code Metrics Impacting Fault Detection

The following treatments were reported by TAR3, from the "overall" (all fitness functions) code metric datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were able to detect a fault. The treatments are scored according to

their impact on class distribution and the number of cases in the treatment-fulfilling subset of the data, and the top-scoring treatments are presented first. Definitions of metrics are listed in Table 4.1.

Two-Minute Overall Dataset:

1. TCD = [0.52..0.93], NOD = [0.00..1.00)]
2. TCD = [0.52..0.93], NS = [0.00..1.00)]
3. TCD = [0.52..0.93], TNLS = [0.00..1.00)]
4. TCD = [0.52..0.93], TNS = [0.00..1.00)]
5. TCD = [0.52..0.93], TNLS = [0.00..1.00)], NS = [0.00..1.00)]

Ten-Minute Overall Dataset:

1. TCD = [0.52..0.93], CD = [0.53..0.93]
2. TCD = [0.52..0.93]
3. PDA = [30.00..208.00], DLOC = [348.00..4017.00]
4. TNLPM = [38.00..287.00], DLOC = [348.00..4017.00]
5. CD = [0.53..0.93]

Figure 4.39 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the "Yes" classification for each respective budget. Comparing to the baseline distribution in Figure 4.13, we can see that the class distribution has shifted heavily in favor of the "Yes" class—from 47.44-71.95% and 49.80-73.90% respectively. As approximately 25% of the examples still have a "No" verdict, these treatments are not perfectly explanatory. Some classes may match the treatment and still evade fault detection. However, the shift in distribution still suggests that the metrics and value ranges named in the treatments have explanatory power. The treatments indicate that:

Generated test suites are effective at detecting faults in well-documented classes.

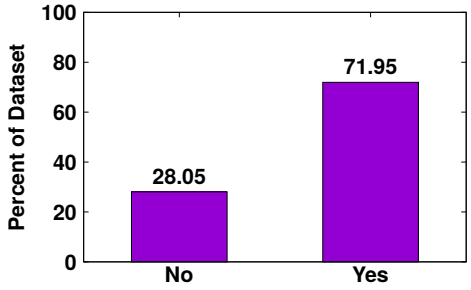


Figure 4.37: 2m Budget

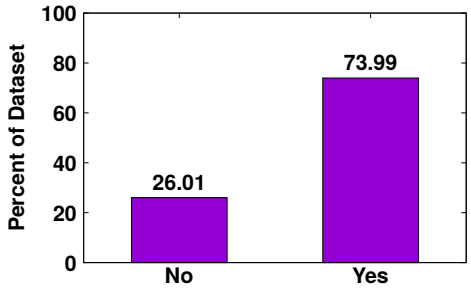


Figure 4.38: 10m Budget

Figure 4.39: Class distributions for the subsets of the two overall datasets fulfilling the top-ranked **"Yes"** treatment for each.

The most consistently-identified metric from the two-minute dataset—and one that appears in treatments for the ten-minute dataset as well—is that a high Total Comment Density (52-93%) tends to indicate that the fault is more likely to be found. This is above the 75^{th} percentile of the results depicted in Figure 4.1. From the ten-minute dataset, we can also see that suites tend to detect faults in classes with a Comment Density of 52-93%, 348-4,017 Documented Lines of Code, and with 30-208 documented public methods (PDA).

Test generation will yield better results if more of the class is publicly accessible.

A Total Number of Local Public Methods from 38-287 indicates that generated suites are more likely to detect a fault. From Table 4.1, we can see that this is well above the median TNLPM (12.00), indicating that allowing direct access to more of your methods will yield better test generation results. A number of treatments also suggest a (Total) Number of (Local) Setters (TNS, NS, TNLS) of 0. Having no setters implies that all attributes are

publicly-accessible. In addition, while Public Documented API is intended to capture how well-documented the class is, it also illustrates this point—a high PDA indicates not just that there are a large number of documented methods, but that a large number of methods are public as well.

The top treatment for the two-minute dataset also suggests a Number of Descendants (NOD) of 0. This value reflects the vast majority of classes, and also appears in the treatments for the "No" classification. Therefore, we consider it to be a coincidental factor.

The test suites for Exception, Method, Method (Top Level, No Exception), and Output Coverage did not detect enough faults to yield useful treatments for the "Yes" classification. Test suites for the remaining four criteria—Branch, Direct Branch, Line, and Weak Mutation Coverage—yielded treatments largely echoing the "overall" dataset. However, multiple treatments for those criteria-specific datasets included the metric-value pairings CLLC = [0.24..0.86] and CC = [0.27..0.91]. These metrics—Clone Logical Line Coverage (CLLC) and Clone Coverage (CC)—tell us that:

> Faults are easier to detect if a large proportion of the class contains duplicate code.

These value ranges are the high end of the scale, and do not include the majority of classes, indicating that the importance of these metrics is not coincidental. Intuitively, if there is a lot of duplicate code, the overall class structure will be easier to cover. Test generation methods driven by code coverage will be able to quickly achieve high levels of coverage, making it easier to reach and execute the code containing the fault.

The following treatments were reported by TAR3, based on the all-fitness-function datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were **not able** to detect a fault. Figure 4.42 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the "No" classification for each.

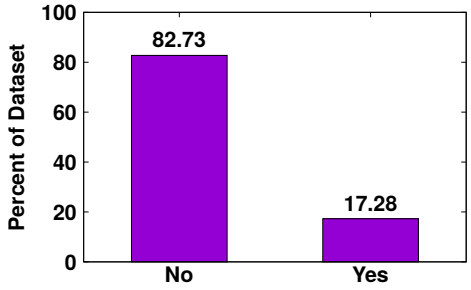


Figure 4.40: 2m Budget

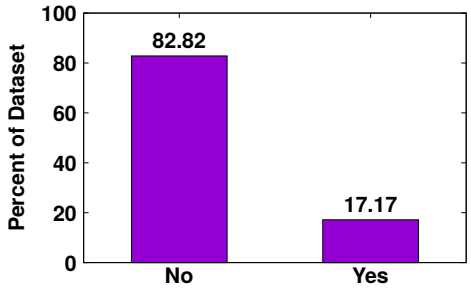


Figure 4.41: 10m Budget

Figure 4.42: Class distributions for the subsets of the two "overall" datasets fulfilling the top-ranked **"No"** treatment for each.

Two-Minute Overall Dataset:

1. DIT = [1.00..2.00)], PDA = [1.00..4.00)]
2. NPM = [0.00..3.00)], NOC = [0.00..1.00)]
3. NPM = [0.00..3.00)], NOD = [0.00..1.00)]
4. NPM = [0.00..3.00)], TNLPA = [0.00..1.00)]
5. CBO = [17.00..98.00], NOC = [0.00..1.00)]

Ten-Minute Overall Dataset:

1. NPM = [0.00..3.00)], LCOM5 = [1.00..2.00)]
2. NS = [0.00..1.00)], CBO = [17.00..98.00]
3. NPM = [0.00..3.00)], NOC = [0.00..1.00)]
4. NPM = [0.00..3.00)], NOD = [0.00..1.00)]
5. NPM = [0.00..3.00)], NOD = [0.00..1.00)], NLPA = [0.00..1.00)]

Comparing to the baseline distribution in Figure 4.13, we can see that the class distribution has shifted heavily in favor of the "No" class—from 52.56-82.73% and 50.20-82.82%

respectively. Once again, not all examples in the subset have a "No" classification. However, the sharp shift in distribution suggests that the treatments have explanatory power. From the produced treatments, we can see that:

Test generation methods struggle with classes that have a large number of private methods or attributes.

The majority of the treatments include a Number of Public Methods (NPM) between 0-3 methods. This is an extremely low number of public methods—far below the median—indicating that much of the class is private. In these cases, the generated suites are likely to have low overall code coverage, and are more likely to miss a fault. One treatment also includes a PDA of 1-3 methods. In this case, this metric's importance is likely not to be as an indicator of the level of documentation, but an indicator that there are a low number of public methods.

Treatments also include a (Total) Number of Local Public Attributes (TNLPA/NLPA) of 0—indicating that any extant attributes are private. Like with the "Yes" treatments, the attributes serve a different purpose than methods in test generation—serving as a way to configure class state and drive coverage of the code contained in the methods. A lack of public attributes limits the ability of the generation framework to control class state.

It can be more difficult to generate tests for classes with dependencies or inherited state.

A Coupling Between Objects of 17-98 is well above the mean of 8, indicating that generated suites are likely to miss a fault in a class coupled to a large number of other classes. In such cases, the test generation framework would need to set up dependencies and put them in the state needed to expose the fault in the targeted class—a non-trivial task.

A related metric is the Depth of the Inheritance Tree (DIT). A DIT of 1 indicates that the target class has a parent class. The existence of a parent class indicates that some methods and attributes are inherited from a parent. This, in itself, is not a problem as the test generator does not need to properly set up a parent like in the last case (and many classes in the dataset have a parent). While we have discussed the metric-value pairs in each treatment as largely being independent, the pairs in a treatment can be related. In this case, the treatment pairs the DIT of 1 with a low PDA. This implies situations where part of the class is inherited from a parent, and where the class has a low number of public methods of its own. If much of the complexity of the class is inherited and new functionality is largely private, the test generation framework may have difficulty in driving execution to the fault location.

Once again, the fact that the vast majority of classes have a NOC/NOD of 0 and that this metric-value pair appears for both targeted classes suggests that these two metrics are coincidental, and we have chosen to ignore them in our discussion. LCOM5 = 1 matches the vast majority of classes in the dataset, and is not particularly informative. Therefore, we also suspect that it is coincidental. All three of these metrics—NOC, NOD, and LCOM5—are paired with a low NPM. We suspect that the low NPM is the true indicator of test efficacy.

One treatment includes the metric-value pair $NS = 0$. This pairing also appeared for the "Yes" treatments, and captures a majority of case examples. We believe it is coincidental for this classification, but not the "Yes" classification, as the treatments for the "Yes" classification included a number of related metrics and often included this pairing. In this case, it was paired with a high CBO—a more informative metric.

The datasets for all eight coverage criteria offered very similar results to the overall datasets for the "No" classification. No additional metric-value pairs were observed.

We will summarize the trends observed among the identified metrics, and discuss their implications. First, **test generation methods struggle with classes that have a large**

number of private methods or attributes, and thrive when the class structure is accessible. The clearest indication of this trend is in the treatments produced for the "No" classification, where a low number of public methods appeared in almost every treatment. In the "Yes" treatments, a large number of public methods is prescribed. This finding is further backed by the prevalence of the Public Documented API metric for both classifications, where the "Yes" treatments prescribe for a PDA of 30-208 methods, and in the "No" treatments, where the treatments prescribe 1-3 methods. PDA is a documentation metric, intended to highlight classes in need of more documentation. However, as it measures the proportion of public methods that are documented, it has a strong correlation to the Number of Public Methods (0.63, measured using the Kendall correlation test). In this experiment, PDA seems to further indicate the effect of private methods on test generation effectiveness.

This finding makes intuitive sense. The test generation technique explored in this experiment is driven by various coverage criteria—largely based on different ways of executing structural elements of the code. The obligations of such criteria will inevitably require that code structures within private methods be executed in the mandated manner. In practice, coverage can be measured over private code, and the feedback mechanisms that power search-based test generation will still reward greater coverage of that code. However, without the ability to directly call such methods, the generation technique will struggle to actually obtain coverage. Private methods must be covered *indirectly*, through calls to public methods. While the generation technique will attempt to adjust the input provided to the public methods to obtain higher indirect coverage of private methods, this indirect manipulation may prove unsuccessful due to constraints placed on how the public method can invoke the secondary private method or discontinuity in the scoring method introduced by this indirect manipulation.

Both the "Yes" and "No" treatments touch on the use of private attributes as well, through the (Total) Number of (Local) Setters and Total Number of Public Attributes. Pri-

vate attributes do not directly prevent code from being covered in the same manner as private methods do. However, they may still result in lower coverage and missed faults by limiting the ability of the test generation technique to manipulate the state of the class. Certain methods may only be coverable by setting the class attributes to particular values, and some failures may only be triggered under particular class states. If the class attributes are private, the test generation technique will need to find indirect means of manipulating those attributes. Again, this is a non-trivial task.

Authors have previously hypothesized that private methods are a reason for poor test generation results [19, 20, 22]. Our findings offer evidence supporting this hypothesis. In practice, we would not advocate that developers reduce the use of private methods or attributes—the protection offered through this feature is crucial. Instead, test generation techniques need to be augmented with better means of increasing coverage of private methods. For example, machine learning techniques may be able to form a behavioral model of the indirectly-called method that could offer better feedback than the existing scoring function used to guide search-based generation.

Second, **generated suites seem to be more effective at detecting faults in classes with more documentation**. The metrics that were the most common indicators of success are largely documentation-related metrics such as the (Total) Comment Density, the Documented Lines of Code, and—to a lesser extent—the Public Documented API. As discussed previously, Public Documented API is strongly correlated to the Number of Public Methods, and the documentation connection is likely to be a coincidence. TCD and DCLOC both have a moderate correlation to the PDA (strengths of 0.48 and 0.47, respectively). However, TCD is only weakly correlated to NPM (0.24) and DLOC is only moderately correlated (0.42). Therefore, TCD, CD, and DLOC are important indicators of efficacy in their own right, and are not merely indicative of a higher number of public methods.

There is no reason to expect the presence of documentation to assist automated test case generation techniques, as such techniques do not make use of documentation in any

way. Instead, *it is important to consider what the presence of documentation implies*. One theory is that there is an unintended selection bias in how the case examples were gathered for Defects4J. The studied faults were identified by searching for project commits that referenced bug reports. Bug reports are more likely to be filed for classes that are frequently invoked by the users and developers of a project. In turn, classes that are more heavily used tend to be ones that developers spend more time refining and polishing. Classes that are expected to be used more heavily will, naturally, be better documented. As a result, it could be that well-documented classes are more likely to be identified as subjects for Defects4J than classes with low amounts of documentation.

However, this theory does not completely explain these results. The majority of classes in Defects4J *do not* have a high TCD or DLOC. The median TCD—36%—is likely to be higher than the median for all classes in the wild, but is still well below the TCD prescribed by the treatments—52-93%. The classes with a high TCD are also not necessarily smaller than other examples. The median LLOC—non-comment lines of code—for classes with a TCD greater than 52% is 288.50, compared to an overall median of 208.50. The median TNM is 37.50, slightly above the median of 37. This implies that the well-documented classes are actually larger than the average class in Defects4J. These are not simpler examples. Further, TCD and DLOC do not have a strong correlation with any non-documentation metric, meaning that there is not a simple explanation within the collected date.

Therefore, there must be some additional factor implied by the presence of high levels of documentation. More research is needed to understand the impact of documentation in these case examples. While the presence of documentation should not directly assist automated test case generation techniques, its presence may hint at the maturity, testability, and understandability of the class.

Third, **classes with a large number of dependencies are more difficult to test than more self-contained classes**. This is indicated in the "No" treatments by a Coupling Be-

tween Objects (CBO) of 17-98 classes, and—to a lesser extent—an inheritance tree depth (DIT) of 1, meaning that the target class inherits functionality from a parent. For a class to be included in Defects4J, it must be directly changed by the patch applied to fix the fault. This means that the fault lies in the class that depends on other classes, rather than being a fault *in* one of the dependencies.

If a class depends on other classes, then the test generation technique may also need to initialize and manipulate the dependencies properly as part of the test generation process. By not doing so properly, we may not be able to achieve high coverage of the target class. Further, we may fail to expose the fault even if the code is covered, as we are trying to make use of the target class outside of the "normal" use of the rest of its dependencies. Because of that, we may see the same *incorrect* behavior from both the working and faulty versions of the class, missing the fault. Researchers have discussed the problem of configuring complex dependencies as part of the broader challenges of controlling the execution environment when generating test cases [19, 21, 23]. Again, our observations provide clear motivation for further research on this topic.

Fourth, when structure-based criteria like Branch Coverage are targeted, **test generation techniques are more effective when a large proportion of the class is duplicated code**. This is supported by the prevalence of Clone Logical Line Coverage (CLLC) and Clone Coverage (CC) in the criteria-specific "Yes" treatments. This observation makes intuitive sense. If a large proportion of the code is identical, then that code will be easier to cover through automated generation. Even if the fault does not lie in the duplicated code, it will be easier to guide execution to the faulty code.

Code duplication is discouraged during development, as any changes will need to be made in multiple locations. Further, duplicating code rather than encapsulating it in one location and invoking it throughout the class will not have any benefit, as a high CLLC simply implies that there is not a significant quantity of non-duplicated code. Rather than offering actionable information, this observation simply indicates that classes with a lot of

duplicate code and very little other functionality are easier to test than complex classes.

4.5 Related Work

Those who advocate the use of adequacy criteria hypothesize that criteria fulfillment will result in test suites more likely to detect faults—at least with regard to the structures targeted by that criterion. If this is the case, we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [143]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [85, 133–135, 161, 167–170]. Inozemtseva et al. provide a good overview of work in this area [135]. Our focus differs—our goal is to examine the relationship between fitness function and fault detection efficacy for search-based test generation. However, fitness functions are largely based on, and intended to fulfill, adequacy criteria. Therefore, there is a close relationship between the fitness functions that guide test generation and adequacy criteria intended to judge the resulting test suites. In recent work, McMinn et al. have even proposed using search techniques to evolve new coverage criteria that combine features of existing criteria [171].

EvoSuite has previously been used to generate test suites for the systems in the Defects4J database. Shamshiri et al. applied EvoSuite, Randoop, and Agitar to each fault in the Defects4J database to assess the general fault-detection capabilities of automated test generation [22]. They found that the combination of all three tools could identify 55.7% of the studied faults. Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. In their work, they only used the branch coverage fitness function when using EvoSuite. Yu et al. used EvoSuite to generate tests for 224 of the faults in Defects4J, examining whether such tests could be used for program repair [172]. In our initial study, we expanded the number of EvoSuite configurations to

better understand the role of the fitness function in determining suite efficacy [20]. We have also compared and contrasted test suites generated to achieve traditional branch coverage and direct branch coverage, noting that each fitness function detects different faults [173].

We used the Defects4J faults to understand the effect of combining fitness functions, identifying lightweight combinations of fitness functions that could effectively detect faults [137]. Rojas et al. also examined combining fitness functions, finding that, given a fixed generation budget, multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [159]. Others have explored combinations of coverage criteria with non-functional criteria, such as memory consumption [174] or execution time [175]. Few have studied the effect of such combinations on fault detection. Jeffrey et al. found that combinations are effective following suite reduction [176]. Recent efforts have been made to introduce many-objective search algorithms that can better balance and cover multiple coverage criteria simultaneously [177, 178].

Object-oriented source code metrics have been used for a variety of purposes. For example, Chowdhury et al. used complexity, coupling, and cohesion metrics as early indicators of vulnerabilities [179]. Singh et al. tried to find the relationship between object-oriented metrics and fault-proneness at different fault severity levels [180]. Mansoor et al. used metrics to detect code smells [181]. Cinneide et al. used cohesion and cloning metrics to guide automated refactoring [182]. Tripathi et al. [183] developed models to predict the change-proneness of the classes using code metrics [183]. Relevant to this study, Toth et al. used SourceMeter to gather the same metrics that we used on classes from Java projects on GitHub [184]. They gathered metrics for multiple revisions, focusing on pairs of revisions related to faulty and working versions of the system. They used this dataset for defect prediction. While our purposes differ and there is no overlap in the studied systems, our dataset could potentially be used to augment their study. Recently, Sobreira et al. also assembled a dataset characterizing the faults in Defects4J [185]. Rather than focusing on class characteristics, they focus on the patches used to fix each fault, characterizing them in

terms of size, spread, and the repair actions needed to perform automated program repair.

4.6 THREATS TO VALIDITY

External Validity: Our study has focused on fifteen systems—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized open-source Java systems. We believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar projects.

We have used a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time. Still, our goal is to examine the coverage criteria, not the generation framework. By using the same framework to generate all test suites, we can compare criteria on an equivalent basis.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, this process still yielded 118,600 test suites to use in analysis. We believe that this is a sufficient number to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

Our learning results are based on a single learning technique. Treatment learning was used to analyze the gathered data, as it is designed to offer succinct, explanatory theories based on classified data [163]—fitting the goal of our work. TAR3 was thought to be appropriate, as it is the most common treatment learning approach and is competitive with other approaches [164].

4.7 CONCLUSIONS

We have examined the role of the fitness function in determining the ability of search-based test generators to produce suites that detect complex, real faults. From the eight fitness functions and 593 faults studied, we can conclude:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and ful-

fill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.

- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations provide evidence for the anecdotal findings of other researchers [19–23] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. More research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algo-

rithm, and CUT in determining the efficacy of test suites. In future work, we plan to further explore these topics.

This work is supported by National Science Foundation grant CCF-1657299.

CHAPTER 5

HOW CLOSELY ARE COMMON MUTATION OPERATORS COUPLED TO REAL FAULTS?

5.1 INTRODUCTION

Software testing—the process of applying stimuli to software and judging the resulting reaction—is the most common means of ensuring that software operates correctly. As lapses in testing carry financial [6], environmental [7], and medical risk [8], it is imperative to ensure that testing yields effective results while remaining cost-effective.

When designing test cases, past experience can be used to estimate the potential effectiveness of the test suite. If we have known software *faults*—mistakes in the source code [1]—we can use detection of these faults to predict whether test cases will be effective against unknown future faults. Essentially, this is an estimation of the sensitivity of the test suite to changes in the source code. In practice, we typically lack a sufficiently large collection of faults to draw reasonable conclusions. Instead, we make use of synthetic faults, known as *mutants* [24].

Mutation testing [25] is a technique in which a user generates many faulty versions of a program—the "mutants" mentioned above—through small modifications of the original code, typically using automated code transformation [24, 26]. *Mutation operators* define transformations over code structures, such as expressions, operators, or references [27]. For example, a mutation operator may change one arithmetic operator into another—turning $A+B$ into $A*B$—permute the order of two statements, add or remove a `static` modifier,

or many other possible changes. There are many mutation operators used in practice [27, 28]. These operators vary in complexity in effect, but all are intended to reflect common, minor mistakes that developers make when writing code.

Mutation testing is a common technique in both testing research and industrial practice. In research, it is the most common method of judging the effectiveness of new testing techniques, particular those used to automatically generate test cases [26]. Mutation is also employed at companies such as Google to identify areas of improvement in test design [29]. In either case, the core hypothesis is that test suites that detect mutants are also effective at detecting real faults, as they are sensitive to these small changes in the code [30].

This hypothesis hinges on the idea that mutants can serve as stand-ins for real faults. Mutants clearly bear little *syntactic* resemblance to real faults [31]. Mutation operators tend to make simple, one-line changes to the code[1]. A glance at any database of real faults, such as Defects4J for Java faults [32], makes it clear that real faults are generally more complex than mutants, often affecting multiple lines of code and requiring multiple changes to any single line to fix. Instead, the idea that mutants can substitute for real faults is based on the assumption of a *semantic* relationship, built on two hypotheses. The first, the "competent programmer hypothesis", suggests that many programs are close to correct, and that minor changes will be enough to fix them. The second, the "coupling effect", suggests that detection of many simple mutants will equate to detection of a single complex fault affecting the same lines of code [1, 33].

However, the truth of these hypotheses—or even the broader hypothesis that, regardless of a semantic relationship, that high levels of mutant detection will lead to increased probability of faulty detection—is not clear. Studies, even conducted on the same real faults, have disagreed on the correlation between mutant and fault detection [24, 187]. Others have found that mutants can serve as a stand-in for real faults under specific conditions,

[1]These are called "first-order mutants" [186]. Higher-order mutation has been proposed, allowing for more complex code transformations. However, this concept has not yet been widely explored [33, 186].

but not broadly [26, 188]. Even if mutation testing can improve the quality of testing efforts, weak empirical evidence and the immense cost of applying mutation testing to a large codebase [33] suggest the need for improvement in the implementation and application of mutation testing.

We hypothesize that improving the effectiveness—in terms of both cost and quality—lies in better understanding the semantic relationship between mutants and real faults, also known as their *coupling*. In particular, and in contrast to past studies, we turn our focus to examining specific mutation operators. That is, what degree of coupling do specific mutation operators have with real faults?

We investigate the degree of coupling by executing developer-written test suites against both mutated and faulty versions of classes from multiple open-source Java projects, based on 144 case examples from the Defects4J fault database [32]. In particular, we focus on the *trigger tests*—the tests that detect the real fault. A mutant that is most strongly coupled to a real fault will be detected only by the trigger tests, and those tests will fail for the same reasons—i.e., the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons. We have defined a scale rating the strength of the coupling between a mutant and a corresponding fault, based on number of failing tests and reasons for failure. This scale, in turn, allows us to contrast 31 mutation operators—applied using the muJava++ framework—based on their tendency to produce mutants with a stronger semantic relationship to real faults. Ultimately, we observed:

- Overall, 61.08% of mutants are detected by developer-written tests. 9.92% of the mutants are strongly coupled to real faults, and a further 9.03% are strongly coupled with additional tests failing. 51.03% of the faults have at least one strongly coupled mutant.

- The level of coupling of individual mutants is relatively low—a median score of 2/10.

16 of the 35 mutation operators (45.71%) have a median score < 2.00.

- EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median level of coupling. The average EMM or $ASRS$ mutant strongly substitutes for corresponding faults. $PRVOU_{SMART}$ mutants are common and tend to either strongly couple or not be detected—making this operator potentially useful for improving test suite quality.

- ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors.

- JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the the largest percentage of mutants that are not detected. $PRVOR_{SMART}$ yields subtle mutants with, often, strong coupling. The other operators could be selectively useful, but may yield many equivalent mutants or cause non-trigger tests to fail.

- SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the the largest percentage of mutants that are only detected by non-trigger tests. These mutants *are* detected, but lack a significant relationship with the corresponding real faults.

- Using median level of coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

Understanding the semantic relationship between mutation operators and faults could enable improvements in how mutation testing is applied. For example, identifying strongly-coupled operators allows prioritization of the mutants used during testing. Exclusion of weakly-coupled operators could lead to cost savings and filtering of "noise" from test suite adequacy estimation. In addition, understanding semantic coupling enables potential improvements in the implementation of existing mutation operators—e.g., ensuring that mu-

tants will compile—and may suggest new mutation operators. To inspire future research, we also make our data available for others to analyze[2]

5.2 BACKGROUND

Analysis of Real Faults

To discuss the relationship between mutation operators and real faults, it is important to first establish certain concepts and terminology related to faults and fault analysis. We broadly adopt the same conventions followed by our experimental subject, Defects4J [32]. In this study, when we discuss a fault, each fault meets the following three properties:

1. Each case example consists of a faulty and a fixed source code version. The changes imposed by the fix must be to source code, not to other project artifacts such as configuration or build files.
2. Each fault must be reproducible—all tests pass on the fixed version and at least one test fails on the faulty version, thereby exposing the fault.
3. Each fault is isolated—the faulty and the fixed version differ only by a minimal set of changes, all of which are related to addressing the fault. That is, the changes are free of unrelated code changes, such as refactoring or feature additions.

When discussing faults, we use the following terminology:

- The **trigger tests** are developer-written test cases that expose the fault. This is the set of tests that fail only on the faulty version.
- The **modified classes** are those classes that are altered in order to fix the fault.
- The **loaded classes** are classes loaded by the Java Virtual Machine during execution of the trigger tests.

[2]Available from `https://doi.org/10.5281/zenodo.7261554`.

- The **relevant tests** are the full set of test cases from test classes that load at least one of the modified classes. The relevant tests are the full set of tests that could potentially detect mutations of the modified classes. The relevant tests include all trigger tests, as well as additional tests that pass on both the fixed and faulty versions of modified classes (but could still detect mutations).

Mutation Testing

Mutation testing [25] is a technique in which a user generates many faulty versions of program through modifications of the original code, generally through automated code transformation [24,26]. Usually a single modification is made to each **mutant**, such as changing an expression (i.e., substituting addition for subtraction), permuting the order of two statements, or deleting statements. The mutations introduced generally match one or more models of the types of mistakes that real developers make when building code (**mutation operators**). Each mutation operator reflects a repeatable type of program change, which can be automatically imposed on a program by searching for statements that fit the correct pattern.

While mutations are *individually* simpler than real faults, two hypotheses suggest that, *together*, mutants are helpful for determining the ability of a test suite to detect real faults.

- The "Competent Programmer Hypothesis" [30, 189] states that programmers tend to develop programs that are close to correct. Although there may be faults in the program, such faults can be corrected with a few simple changes. Mutations are intended to represent types of simple changes that are made in practice [189].
- The "Coupling Effect" [30] states that tests that distinguish a large number of mutants from the original program are so sensitive that they also will implicitly distinguish more complex errors as well. Essentially, this hypothesis postulates that mutation testing is an effective sensitivity analysis, and that test suites that detect more mutants are more likely to also detect even subtle real-world faults.

Generally, mutants are introduced with the intent that they not be trivially detected—they are both syntactically valid and semantically useful [1]. That is, effective mutants will compile ("valid"), and will not trivially cause test cases to fail ("useful") [1]. Mutations can be used to assess the effectiveness of a test suite by examining how many mutants are **killed** (that is, detected) by the tests within the test suite.

Detection can be assessed using either **strong** or **weak mutation** [190]. To kill a mutant using strong mutation, the following four conditions must be met [190]:

- (R) test execution must *reach* the mutation in the modified class.
- (I) the test must *infect* program state by causing it to differ between the original and mutated class.
- (P) incorrect state must *propagate* to class output.
- (R) assertions in the test must *reveal* the difference.

Weak mutation only requires the (R,I) steps. A mutant is considered weakly detected if the mutated statement is reached, and the result of calculating that expression is corrupted [191].

In this study, we employ **strong mutation**. We do not consider a mutant killed unless an assertion in the test case is able to reveal the corrupted state through the output of a method call or by inspecting a class variable. We do not attempt to remove equivalent mutants, as such detection is generally undecidable [192] and we do not consider equivalence to be a threat to our results (i.e., if an operator tends to yield many equivalent mutants, it would not be strongly coupled to real faults).

A **mutation score** can be determined by dividing the number of killed mutants by the number of all mutants. Mutants are considered **equivalent** if no test can corrupt program state for that mutant. Deciding equivalence is generally an undecidable problem [192], but techniques exist that can determine equivalence for a subset of mutations [33].

5.3 METHODOLOGY

We hypothesize that improving the quality and cost effectiveness of mutation testing requires examining the *semantic* relationship between mutation operators and real faults. That is, are there mutation operators that tend to trigger the same *outcomes* as real faults? A clearer understanding of how mutation operators tend to couple to real faults carries a number of potential benefits, allowing prioritization of mutation testing efforts (e.g., enabling cost savings by selectively omitting operators that weakly couple to real faults), improved implementation of existing mutation operators, and insights into new mutation operators that could be applied.

In this study, we investigate this semantic relationship by assessing the degree of coupling between mutants and real faults using a spectrum of outcomes, based on the *number of failing trigger tests* and the *reasons for failure*. We execute the developer-written test suite on mutations to the classes (and specific lines) modified to fix a fault, and we examine which mutants are detect and which tests fail. A mutant that is most strongly coupled to a real fault will be detected by the trigger tests—and only the trigger tests—and those tests will fail for the same reasons—i.e., will trigger the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons.

We are interested in addressing the following research questions:

- **RQ1:** What is the degree of coupling between mutants and real faults?
  - **RQ1.1:** Which mutation operators yield mutants that most *strongly* couple to faults?
  - **RQ1.2:** Which mutation operators yield mutants that tend to result in compilation errors or lack of detection?
  - **RQ1.3:** Which mutation operators yield mutants that tend to be detected only by non-trigger tests?

- **RQ2:** What potential cost savings could be achieved by selectively omitting mutation operators weakly coupled to faults?

To answer these questions, we performed the following experiment:

1. **Collected Case Examples:** We have used 144 case examples, from five Java projects, as case examples (Section 5.3).
2. **Generated Mutants:** For each fixed modified class for each case example, we generated mutants for the lines of code that differ between the faulty and fixed versions of the classes. We perform this generation using 31 mutation operators offered by the muJava++ framework (Section 5.3).
3. **Executed Test Suites:** For each mutant, we execute all relevant tests (Section 5.3).
4. **Recorded Failure Information:** For each mutant, we measure the number of failing trigger tests, number of failing non-trigger tests, and the reasons for failure (i.e., exceptions or error messages) (Section 5.3).
5. **Assessed Coupling:** For each mutant, we use the information gathered above to assess the degree of coupling of that mutant to the real fault using a scale that reflects the outcomes of relevant test execution (Section 5.3).

Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [32][3]. Defects4J has been used extensively in test generation [22,88], automated program repair [115], and fault localization [193] experiments. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault.

In this study, we generate mutants for the modified classes for 144 faults from five projects in Defects4J: Chart (22 faults), Closure (22 faults), Lang (41 faults), Math (47

[3]Available from `http://defects4j.org`

Table 5.1: ID numbers of studied faults from Defects4J. Faults in bold lacked any strongly-substituting mutants in our experiment.

Project	Bugs
Chart	1, **2**, 3, 4, **5, 6, 7**, 9, 10, 11, 13, **14**, 15, **16, 17, 18, 19**, 21, **22**, 24, **25**, 26
Closure	9, 12, **23, 34, 52**, 56, **65, 77, 85, 99, 100, 102, 123**, 124, **128**, 131, **147, 161, 162, 164**, 169, **173**
Lang	**2, 4**, 5, **7**, 11, **12**, 16, **19, 20**, 22, 23, 24, 27, 28, **29, 30**, 31, **34, 37**, 39, **40**, 42, **43**, 44, 45, **46, 47, 48**, 49, **51**, 52, 54, 55, **57**, 58, **59**, 60, 61, 62, 63, 65
Math	3, 5, **8**, 9, 11, **15, 17**, 19, **22**, 23, 24, **27, 29, 30, 37**, 40, 41, **43, 46, 47**, 48, 49, 51, 53, **54**, 56, 60, **64, 66**, 67, **68, 69**, 70, 72, 73, 82, 84, 85, **89**, 95, **96**, 97, **98**, 102, 103, 105, 106
Time	2, **3, 4, 5, 6, 7, 12**, 14, 15, 16, 18, **27**

faults), and Time (12 faults). The specific faults utilized are listed in Table 5.1. Some faults from the version used, Defects4J 1.4, were excluded because (a) the lines of code that differ between the faulty and fixed versions of classes yielded no mutants for any of the employed mutation operators, (b) the lines of code that we attempted to mutate used Java features not supported by muJava++, or (c), muJava++ failed to produce usable mutants for some other reason. While we were unable to produce mutants for all faults, the subset of 144 faults yielded a large set of mutants of sufficient variety to produce reasonable experimental results.

Mutant Generation

There are several frameworks for generating mutants for Java, including PIT [194], muJava [195], and Major [27]. Each tool offers its own features and set of mutation operators. In this experiment, we used the framework muJava++, an extended version of muJava with additional and reworked mutation operators[4]. We employ muJava++ because (a) it offers a large number and variety of mutation operators, (b) it can be applied to user-specified lines of code, and (c), it can export mutants as Java files for execution and analysis. We will further explain each point, and how we applied the framework, below.

Mutation Operators: As our goal is to explore the relationship between mutation operators and real faults, we require a mutation framework that can produce mutants for a large

[4]Available from `https://github.com/saiema/MuJava`.

Table 5.2: Mutation operators from muJava++ used in this experiment.

Operator Name	Description
AMC	Changes the access modifier of methods and class fields.
AOD	Replaces an arithmetic operation with one of its members. For example, (a = b + c) becomes (a = b) or (a = c).
$AODU$	Deletes basic unary arithmetic operators. (+, -)
$AOIS$	Inserts short-cut arithmetic operators (++, --).
$AOIU$	Inserts unary arithmetic operators (+, -).
$AORB$	Replaces arithmetic operators (*, /, %, +, -) with other operators.
$AORS$	Replaces short-cut arithmetic operators (++, --) with other operators.
$AORU$	Replaces unary arithmetic operators (+, -)
$ASRS$	Replace short-cut assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) with other operators.
COD	Deletes conditional operators (&&, ||, &, |, ^).
COI	Inserts conditional operators (&&, ||, &, |, ^).
COR	Replaces conditional operators (&&, ||, &, |, !) with other operators.
$CRCR$	A constant C in the code is mutated to be one of (1, 0, -1, -C, C+1, C-1).
EAM	Changes an accessor method name for other compatible accessor method (where methods have the same signature).
EMM	Changes a setter method name for other compatible setter method (where methods have the same signature).
EOA	Replaces an assignment of an object reference with a copy of the object, using the `clone()` method. Only performed if the object has a declared `clone()` method.
EOC	Changes an object reference check to object content comparison through Java's `equals()` method.
ISD	Deletes occurrences of the `super` keyword so that a reference to a variable or method is no longer to the parent class' variable or method.
ISI	Inserts the `super` keyword so a reference to a variable or method in a child class uses the parent variable or method.
JTD	Deletes uses of the keyword `this`.
JTI	Inserts uses of the keyword `this`.
LOI	Inserts logical operators (&, |, ^).
LVR	Replaces a literal with a default value. Numeric literals become (0, 1, -1), Booleans become (`true`, `false`), Strings are replaced with an empty string.
OAN	Changes the number of arguments in method invocations, but only if there is an overloading method that can accept the new argument list.
$PRVOL_{SMART}$	The $PRVO$ operator changes object references in assignment statements to instead refer to other objects of a compatible type. $PRVOL$ mutates references on the left-hand side of the assignment, and mutations must be compatible with the right side. $PRVOL_{SMART}$ only uses references to reachable variables.
$PRVOR_{SMART}$	Same as $PRVOL_{SMART}$, except applied to the right-hand side of the assignment, and mutations must be compatible with the left side.
$PRVOR_{REFINED}$	Same as $PRVOR_{SMART}$, except it also uses literals found inside the method.
$PRVOU_{SMART}$	Same as $PRVOR_{SMART}$, except applied to `return` expressions.
$PRVOU_{REFINED}$	Same as $PRVOR_{REFINED}$, except applied to `return` expressions.
ROR	Replace relational operators with other relational operators (==, !=, $<$, $\leq$, $>$, $\geq$), or replace an entire predicate with true and false.
SOR	Replaces shift operators ($>>$, $<<$, $>>>>$) with other operators.

variety of operators. muJava++ supports 62 mutation operators, although some operators have multiple variations. In our experiments, we were able to produce mutations for 31 of the operators. These 31 operators are explained in Table 5.2.

Mutant Generation: We generate mutants by applying all operators to the *specific lines of code* that differ between the fixed and faulty versions of the modified classes for each

Table 5.3: Number of mutants produced for each operator across each project from Defects4J (and overall), sorted by the total number of mutants.

Operator	Chart	Closure	Lang	Math	Time	Overall
$PRVOU_{REFINED}$	5287	4880	3125	3663	1724	18679
$PRVOU_{SMART}$	540	299	53	2760	54	3706
$PRVOR_{REFINED}$	121	31	297	940	226	1615
$AOIS$	185	120	492	295	68	1160
ROR	139	152	489	223	113	1116
ISI	142	121	312	338	66	979
$PRVOR_{SMART}$	92	159	24	368	77	720
LOI	68	44	278	262	51	703
JTI	84	63	211	269	42	669
$CRCR$	51	27	223	236	59	596
$AORB$	4	4	160	204	24	396
$PRVOL_{SMART}$	95	0	57	216	4	372
COI	44	47	149	67	22	329
$AOIU$	27	5	73	147	22	274
COR	9	90	129	18	6	252
AOD	6	2	57	112	12	189
EAM	11	1	40	0	44	96
COD	5	8	13	2	0	28
$ASRS$	0	0	0	18	0	18
EOC	7	2	6	1	1	17
EOA	8	0	4	0	4	16
LVR	0	4	10	2	0	16
AMC	0	0	12	3	0	15
$AORS$	5	1	3	6	0	15
$AORU$	4	0	1	10	0	15
$AODU$	4	0	1	8	0	13
OAN	0	1	5	5	0	11
SOR	0	0	0	4	0	4
ISD	1	0	2	0	0	3
JTD	1	0	0	1	0	2
EMM	0	0	0	2	0	2
Overall	**6932**	**6061**	**6223**	**10169**	**2617**	**32002**

case example in Defects4J version 1.4. We apply this restriction because we are interested only in the mutants that could potentially semantically replicate an actual fault. It is highly unlikely that a mutant to another class or an unrelated portion of a modified class could replicate the real fault. Therefore, mutants outside of the faulty code potentially mislead the analysis.

We generate all possible mutations for all mutation operators for the lines of code that have been changed to fix the fault. We apply mutations to the fixed code, as the mutants should represent alternative "buggy" versions. Table 5.3 lists the number of mutants produced for each operator for each project from Defects4J, as well as across the dataset.

muJava++ exports each mutant as a Java file that can be substituted for the real file

during test execution. This capability allows us to execute the mutants using Defects4J's built-in test execution capabilities—ensuring that the assumptions and constraints of the dataset hold during mutant execution. It also enables further qualitative analysis through inspection of the mutated code.

Data Collection

To examine the relationship between mutation operators and real faults, we execute the "relevant tests"—all developer-written test cases that execute, directly or indirectly, the faulty code—against all mutated versions of the modified classes for each case example. To perform test execution, we use the `defects4j test` utility offered by the framework. This utility executes the relevant tests in a controlled test execution environment that ensures that the expected behavior of the faulty and fixed code is preserved and that all constraints and assumptions of the dataset hold. To use this utility for test execution for mutants, we perform the following steps for each mutant:

- We checkout the fixed version of the case example.
- We exchange the appropriate modified class for the mutated class.
- We execute `defects4j compile` to compile the project. If the mutant does not compile, we abort execution for that mutant and record the result.
- We execute `defects4j test` in the verbose mode.
- We record any tests that fail or result in an error, and record the reasons for failure. The "reason" can include either a failed assertion or a raised exception.

We then compare the test failures and reasons for failure to the trigger tests for that case example—the developer-written tests that fail for the real fault. The metadata for Defects4J includes the stack traces for each failing developer-written test, which preserves the reasons

that the trigger tests fail for the real fault. This enables direct comparison of both failing test cases and reasons why those tests fail.

When comparing "reasons", we check that the same assertions fail or that the same exceptions are thrown. However, we do not check that the exact same output is issued by the code. For example, consider a situation where an assertion checks that the return value is 10. If a mutant returns 7 and the real fault returns 12, we still consider the reason for failure to be the same—the return value was not 10—even though the mutant and real fault do not return the same incorrect value. Our definition of semantic similarity is satisfied if a mutation and a real fault are detected by the same test cases using the same assertions (or other failure causes).

Based on the test executions, we create a dataset with one line per mutant execution. For each mutant, we record:

- Basic metadata: the project name, the fault ID, the name of the modified class, an identifier for the mutant, the mutation operator applied, and the number of trigger tests for that case example.
- The total number of tests that failed or resulted in an error for that mutant.
- The total number of trigger tests that failed or resulted in an error for that mutant.
- The total number of trigger tests that failed for the same reason for both the real fault and that mutant.

Coupling Categorization

To assess the semantic coupling of mutants to real faults, we have developed the following scale. This scale accounts for the range of possibilities when executing the relevant tests against each mutant:

- **Strong Substitution:** All trigger tests fail, and they all fail for the same reason that they failed for the real fault. No additional tests fail. This represents an exact semantic replacement of the real fault, given the developer-written test suite.
- **Test Substitution:** All trigger tests fail, but one or more fail for differing reasons. No additional tests fail.
- **Partial Substitution:** Some, but not all, trigger tests fail. All failing trigger tests fail for the same reason as the real fault. No additional tests fail.
- **Partial Test Substitution:** Some, but not all trigger tests fail. Not all failing trigger tests fail for the same reasons. No additional tests fail.
- **(Strong/Test/Partial/Partial Test) + Additional Tests Fail:** The same definitions as above apply, but additional non-trigger tests fail.
- **No Substitution:** No trigger tests fail, but additional non-trigger tests fail.
- **Not Detected:** No tests fail.
- **Does Not Compile:** The generated mutant does not compile, preventing test execution.

This scale is used to assess, for each mutant, the results of executing the relevant tests against that mutant in relation to executing the same tests against the real fault. For each mutant in the above dataset, we assign a categorization from this spectrum of possibilities.

In some analyses, we also assign a numeric value for each of the ten categories on this scale, with higher values indicating closer coupling. These values are: (0) Does Not Compile, (1) Not Detected, (2) No Substitution, (3) Partial Test + Additional, (4) Partial Test, (5) Partial + Additional, (6) Partial, (7) Test + Additional, (8) Test, (9) Strong + Additional, and (10), Strong Substitution.

Table 5.4: Number and percentage of detected mutants for each project (and overall).

Project	Detected	Total	Percentage
Chart	4175	6932	60.23
Closure	3072	6061	50.68
Lang	4160	6223	66.85
Math	6625	10169	65.15
Time	1515	2617	57.89
All	**19548**	**32002**	**61.08**

Table 5.5: Categorization of coupling for each mutant, for each project (and overall).

Category	Score	Chart	Closure	Lang	Math	Time	Overall
Compile Error	0	1405 (20.23%)	2724 (44.94%)	785 (12.60%)	1440 (14.16%)	237 (9.05%)	6591 (20.58%)
Not Detected	1	1352 (19.47%)	265 (4.37%)	1278 (20.52%)	2104 (20.68%)	866 (33.07%)	5865 (18.31%)
No Substitution	2	68 (0.98%)	486 (8.02%)	647 (10.39%)	1778 (17.48%)	669 (25.54%)	3648 (11.39%)
Partial Test + Additional	3	74 (1.07%)	251 (4.14%)	436 (7.00%)	40 (0.39%)	18 (0.69%)	819 (2.56%)
Partial Test Substitution	4	453 (6.52%)	0 (0.00%)	256 (4.11%)	110 (1.08%)	0 (0.00%)	819 (2.56%)
Partial + Additional	5	21 (0.30%)	401 (6.62%)	305 (4.90%)	143 (1.41%)	279 (10.65%)	1149 (3.59%)
Partial Substitution	6	2071 (29.82%)	107 (1.77%)	541 (8.69%)	1345 (13.22%)	13 (0.50%)	4077 (12.73%)
Test + Additional	7	131 (1.89%)	1276 (21.05%)	148 (2.38%)	467 (4.59%)	212 (8.10%)	2234 (6.98%)
Test Substitution	8	25 (0.36%)	0 (0.00%)	362 (5.81%)	333 (3.27%)	11 (0.42%)	731 (2.28%)
Strong + Additional	9	374 (5.39%)	494 (8.15%)	568 (9.12%)	1296 (12.74%)	159 (6.07%)	2891 (9.03%)
Strong Substitution	10	958 (13.80%)	57 (0.94%)	897 (14.40%)	1113 (10.94%)	153 (5.84%)	3178 (9.92%)

Table 5.6: Number of real faults with at least one corresponding "strongly substituting" mutant for each project.

Project	With Strongly Substituting Mutants	Total	Percentage
Chart	12	22	54.55%
Closure	6	22	27.27%
Lang	23	41	56.10%
Math	28	47	59.57%
Time	5	12	41.67%
All	**74**	**144**	**51.03%**

5.4 Results and Discussion

Overview of Coupling Between Mutants and Real Faults (RQ1)

Table 5.4 presents an overview of the number and percentage of mutants detected by the test cases (trigger and non-trigger) for each project from Defects4J. As a baseline, we observe:

> Overall, 61.08% of mutants are detected.

This percentage is relatively consistent across projects, with the lowest percentage being 50.68% in the Closure project. The remaining mutants either are not detected by the

test cases (18.31%) or resulted in compilation errors (20.58%).

Table 5.5 categorizes each result according to the scale previously defined in Section 5.3. To aid further analysis, we also assign a numeric score to each category based on the degree of coupling, with higher scores indicating closer coupling. Table 5.6 further indicates the number of faults with at least one mutant categorized as "strong substitution".

> 9.92% of the mutants are strongly coupled to real faults, and a further 9.03% are strongly coupled with additional tests failing. 51.03% of the faults have at least one strongly coupled mutant.

The strongly substituting mutants can serve as stand-ins for the real faults, yielding the same failing test cases and the same test outcomes. Approximately half of the studied faults have at least one strongly-substituting mutant, with relatively consistent results across all projects other than Closure—where only 27.27% of faults have at least one strongly substituting mutant. The overall percentage of mutants belonging to this category falls behind compilation errors (20.58%), "not detected" (18.31%), "no substitution" (11.39%)—where only non-trigger tests fail–and "partial substitution" (12.73%)—where only a subset of trigger tests fail, but those that fail offer the same reasons for failure.

Test substitution cases—where trigger tests fail, but for alternative reasons—are rarer than strong substitutions, but still present. Mutations to these lines still cause test cases to fail, but they do not replicate the semantic effect of the fault. Commonly, these are cases where a mutation causes an exception to be thrown when one was not expected.

The distribution of mutants belonging to each category varies somewhat between projects. The Closure project particularly stands out, as only 0.94% of mutants strongly couple to the faults. In this project—among the detected mutants—the largest category is test substitution with additional failing tests. The Closure compiler has complex validity checking

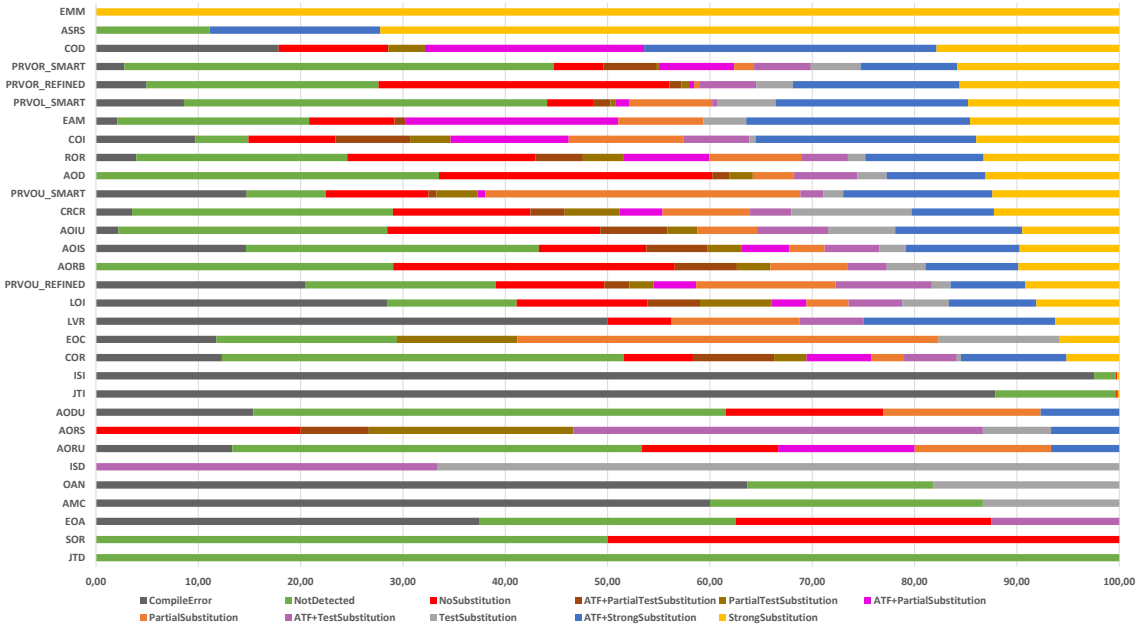


Figure 5.1: Percentage of mutants generated for each operator matching each category, sorted by the percentage strongly substituting.

code[5] for the abstract syntax tree produced during compilation. Many mutations are caught by this code, which throws an exception when a validity condition fails. This produces a large number of test failures—often for alternative reasons than those expected by the test designers.

The percentage of generated mutants belonging to each coupling category are depicted for each of the 31 mutation operators in Figure 5.1. Using the assigned values, we also note the median and average "scores", as well as the standard deviation and number of mutants, for each operator in Table 5.7. Overall, we observe:

> The level of coupling of individual mutants is low—a median of 2.00/10.00. 16 of the 31 mutation operators (45.71%) have a median score of < 2.00.

The average scores are higher, but the average can be influenced by outliers. Therefore, we

[5]E.g., `https://github.com/google/closure-compiler/blob/ca23c597fc17c95ece2aae07f4e503a34c88c61f/src/com/google/javascript/jscomp/ValidityCheck.java`.

Table 5.7: For each mutation operator, the number of mutants, median and average coupling score, and standard deviation in coupling score.

Operator	Number of Mutants	Median	Average	Standard Deviation
EMM	2	10.00	10.00	0.00
$ASRS$	18	10.00	8.83	2.79
ISD	3	7.00	7.67	0.47
COI	329	6.00	5.69	3.36
$PRVOU_{SMART}$	3706	6.00	5.21	3.44
EOC	17	6.00	4.65	2.95
COD	28	5.00	5.79	3.73
EAM	96	5.00	5.69	3.38
$AORS$	15	4.00	5.33	2.33
$CRCR$	596	4.00	4.73	3.46
ROR	1116	4.00	4.65	3.41
$PRVOL_{SMART}$	372	3.00	4.73	3.94
$AOIU$	274	3.00	4.42	3.44
$PRVOR_{SMART}$	720	3.00	4.38	3.72
$PRVOU_{REFINED}$	18679	3.00	3.97	3.50
$PRVOR_{REFINED}$	1615	2.00	4.62	3.79
AOD	176	2.00	4.09	3.52
$AORB$	396	2.00	3.98	3.29
$AOIS$	1160	2.00	3.81	3.57
LOI	703	2.00	3.55	3.54
COR	252	1.00	3.24	3.23
$AORU$	15	1.00	2.73	2.65
$AODU$	13	1.00	2.39	2.68
EOA	8	1.00	1.63	2.18
SOR	4	1.00	1.50	0.50
JTD	2	1.00	1.00	0.00
LVR	16	0.00	3.63	4.01
OAN	11	0.00	1.64	3.02
AMC	15	0.00	1.33	2.65
JTI	669	0.00	0.14	0.51
ISI	976	0.00	0.04	0.48
Overall	**32002**	**2.00**	**4.01**	**3.58**

focus on the median scores. We will discuss specific operators in more detail in the coming subsections.

However, we can make some initial observations. We observe the highest variance from the LVR, $PRVOL_{SMART}$, $PRVOR_{REFINED}$, COD, and $PRVOR_{SMART}$ operators. EMM, JTD, ISD, ISI, and SOR have the lowest variance. The level of variance does not suggest a particular relationship with the level of coupling. Operators with low variance can tend towards strong coupling (e.g., EMM) or not being detected at all (e.g., JTD or SOR). Operators with high variance tend to span the range of outcomes—e.g., COD is split between compilation errors/lack of substitution and strong substitution/strong with additional tests. However, operators with high variance should be considered further in

future research. Implementation details of such operators could be considered. Certain types of statements could potentially be avoided or prioritized with the goal of increasing the percentage of mutants that couple with real faults.

Strongly-Coupled Mutation Operators (RQ1.1)

> EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median level of coupling. The average EMM or $ASRS$ mutant strongly substitutes for corresponding faults. $PRVOU_{SMART}$ mutants are common and tend to either strongly couple or not be detected—making this operator potentially useful.

The EMM operator replaces a setter method reference for another compatible setter. EMM mutants are very rare—both mutations were for fault Math-106[6], where `setIndex(...)` was changed to `setErrorIndex(...)` in different occurrence of this line. ISD mutants—which delete occurences of the `super` keyword—are also quite rare. It is difficult to generalize from operators yielding so few examples, so both should be further examined in future work.

EOC changes a comparison (==) to a reference to `.equals()`. For example, in Lang 39[7], `replacementList[i] == null` is changed to `replacementList[i].equals(null)`. Mutants from this operator are also relatively rare—with only 17 examples, mostly in Chart and Lang. While some of these mutations result in strong substitution, the majority result in partial substitution.

The $ASRS$ operator replaces short-cut assignment operators, e.g., changing += to /=. All 18 mutants for this operator appear in the Math project. Because the Math project

[6] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/106.src.patch

[7] https://github.com/rjust/defects4j/blob/master/framework/projects/Lang/patches/39.src.patch

focuses on mathematical functions, such mutants may also be more likely in this project to match the actual mistakes made by developers. For example, in Math 102[8], multiple ASRS mutations led to the same result as the real fault.

COI and $PRVOU_{SMART}$ both yield a much larger number of faults, appearing across all projects. Naturally, the median coupling for both is lower than the three rarer operators already mentioned, but still generally high. The COI operator inserts conditional operators. For example, in Math 84[9], COI changes an instance of `true` to `!true`. This causes a `return` in the same (incorrect) location as the real fault.

$PRVOU_{SMART}$ replaces object references with other compatible references—variables or methods—in `return` expressions. For example, in Math 5[10], it replaces a reference to INF with calls to float-returning methods (e.g., `this.atan()`) from the class-under-test. Because both operators are widely applicable—but still tend towards a strong relationship to real faults—they are potentially useful for use in assessing and expanding test suites.

Compilation Errors and Non-Detection (RQ1.2)

We observe two primary reasons for low median coupling scores—either a large percentage of mutants result in compilation errors or a large percentage are not detected. We discuss both situations below.

ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors.

[8] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/102.src.patch`

[9] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/84.src.patch`

[10] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/5.src.patch`

All five of these operators have a median score of 0.00, indicating that the majority of mutants result in compilation errors. This is confirmed in Figure 5.1. These operators make changes that can easily "break" code without proper precautions. For example, ISI inserts the `super` keyword, JTI inserts the `this` keyword, and AMC changes access modifiers for methods and fields. All three can yield useful mutants, but they also assume conditions of the code that may not be true—e.g., in the case of ISI, not all classes have parents.

Mutations resulting in compilation errors—for these and other operators—are not useful for evaluating the strength of a test suite. Although some time is saved by not needing to execute the test suite against these mutants, time and effort are still wasted on attempting compilation and analyzing the resulting failure. Cost savings could be achieved from either avoiding the use of such mutation operators altogether or improving their implementation such that compilation errors are avoided.

The developers of mutation frameworks should explore measures that would prevent the generation of non-compiling mutants. For example, mutations of certain types of code structures could be avoided. At the very least, mutation testing frameworks should check whether assumptions are met. In the above examples, the ISI operator implementation should check that a class has a parent before inserting the `super` keyword or the implementation of the JTI operator should check whether a call is to a static method before inserting `this`.

JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the the largest percentage of mutants that are not detected. $PRVOR_{SMART}$ yields subtle mutants with, often, strong coupling to real faults. The other operators could be selectively useful, but may yield many equivalent mutants or cause non-trigger tests to fail.

Mutants that do not compile detract from the effectiveness of mutation testing. Those that are not detected can either detract—if they are equivalent to the original code—or can

be very useful—if they are subtle and require sensitive test cases to detect. Therefore, it is also worthwhile to examine these operators.

The JTD operator deletes uses of the keyword `this`. There were only two mutants of this type in our set, and both were equivalent mutants. There are many situations where this operator could yield equivalent mutants. Therefore, we would suggest only employing this operator in cases where behavior might be affected significantly when the keyword is removed.

The $AODU$ operator—which deletes unary arithmetic operators, e.g., changing -1 to 1—and $AORU$ operator—which replaces such operators—may also be selectively useful when unary operators are employed. However, both also lack strong connections with real faults. When such mutants are detected, tests outside of the trigger tests tend to fail.

SOR replaces shift operators (e.g., $>>$) with other operators. This operator produced four mutants for Math 40[11], where two were not detected and two caused non-trigger tests to fail. Shift expressions are typically only used in specialized code, but it seems this operator could be useful for assessing test suite adequacy when such operators are used.

$PRVOR_{SMART}$ is similar to the previously-discussed $PRVOU_{SMART}$. It replaces object references with other compatible references on the right-hand side of assignment expressions. When mutants are detected, they often strongly substitute for real faults. Some of the not-detected mutants are equivalent. Many, however, could be detected with the addition of further test cases. Therefore, this operator could be useful in improving test suite quality.

Detection By Non-Trigger Tests (RQ1.3)

One further situation that we would like to examine are the mutation operators that tend to yield mutants only detected by non-trigger tests. This is the "no substitution" category in

[11] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/40.src.patch`

our scale.

SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the the largest percentage of mutants that are only detected by non-trigger tests. These mutants *are* detected, but lack a significant relationship with the corresponding real faults.

$PRVOR_{REFINED}$ is an extended version of $PRVOR_{SMART}$, discussed previously. The difference between these operators is that $PRVOR_{REFINED}$ is able to also reference literals found in the method. The use of literals seems to lead to a large number of cases where tests outside of the trigger tests fail, including both "no substitution" and "additional tests fail" outcomes.

$AORB$—which replaces arithmetic operators—and AOD—which replaces an entire arithmetic expression with one of its member variables—yield mutants that span the entire spectrum of possibilities. These operators lead to many "no substitution" outcomes, as well as many cases where mutants are not detected or strongly substitute. Based on these observations, as well as the earlier observations of similar operators often producing mutants that are not detected, it seems that mutation operators related to arithmetic expressions lack a predictable semantic relationship with real faults. Arithmetic expressions are common when programming. Developers *do* make mistakes involving such expressions. However, such expressions also are not predictive of the existence of a fault.

The EOA operator replaces an assignment of a object reference with a clone of that object, in situations where the `clone()` operation is defined. Such mutants are relatively rare, but largely fall into the "not detected" and "no substitution" categories. This operator may be of selective use in cases where `clone()` is implemented.

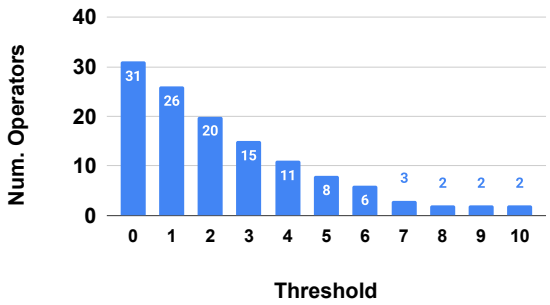


Figure 5.2: Number of operators remaining if the median level of coupling is used as a threshold for determining the subset of operators employed.

Potential Cost Savings Through Filtering Operators (RQ2)

Mutation testing is a notoriously expensive practice, as test cases must be executed against each mutant [29]. For mutation testing to be a viable technique in industrial development, that cost must be reduced. Past research has examined methods of filtering the set of mutants or mutation operators employed (e.g., [196, 197]).

Similarly, the typical degree of coupling of an operator to real faults could be used to select a subset of mutation operators for use in mutation testing—focusing on operators that tend to have a close relationship to real faults. If a threshold is carefully selected, this could lead to a small subset of mutants that are—we hypothesize—useful for assessing the strength of existing test cases and for targeting in the design of additional test cases.

There are many possible methods of using the assessed degrees of coupling for making predictions of a subset of operators that could be useful in assessing the test suites of new projects. We explored one such method—using the median "score" of an operator to determine its inclusion in the subset employed during mutation testing. In this scenario, we would select a score threshold and compare the median score for an operator to this threshold. If it falls above this threshold, we would include it in the subset employed. If not, the operator would be filtered out.

Figure 5.2 indicates the number of mutation operators that would fall in this subset for all thresholds (0–10). Figure 5.3, then, indicates the number of mutants in the set

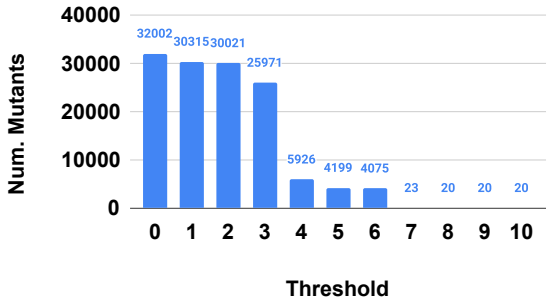


Figure 5.3: Number of mutants remaining if the median level of coupling is used as a threshold for determining the subset of operators employed.

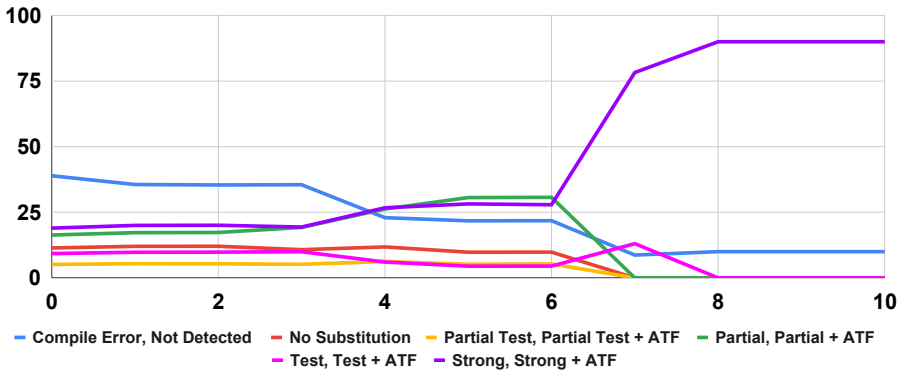


Figure 5.4: Percentage of mutants in the remaining subset (if median level of coupling is used as a threshold) belonging to each coupling category.

considered in our experiment that would be included in that subset. Finally, Figure 5.4 indicates the distribution of mutants in this subset matching the different levels of coupling.

Figure 5.2 shows a steady drop in the number of operators as the score threshold grows higher. However, there are massive reductions in the number of mutants in the subset (Figure 5.3) when the threshold moves from 3–4, due to the loss of the $PRVOU_{REFINED}$ operator, and from 6–7, due to the loss of the $PRVOU_{SMART}$ operator.

There are multiple thresholds that could make sense. Ultimately, a developer should utilize a subset with a reasonable *variety* of operators to ensure that tests are sensitive to different types of faults. In addition, that subset should still contain a reasonably high number of mutants—while still remaining cost-effective—to ensure that tests are robust across a reasonable span of the codebase. If a threshold is too low, the set of mutants will

remain unreasonably expensive to assess. If it is too high, the set of mutants will be too small and lack diversity—reducing the power of mutation testing to assess the sensitivity of a test suite across the codebase.

Based on Figures 5.2–5.3, a threshold of 4–6 seems most reasonable. In the scale defined previously, we would expect the "average" mutant in this subset to range from "partial test substitution" to "partial substitution". These threshold yield a reasonably-sized set of mutants compared to lower thresholds, while still retaining a variety of operators. As shown in Figure 5.4, the distributions of coupling categories for the mutants in the subsets considered in our experiment are relatively stable in this range, with approximately 25% being strongly coupled and approximately 75% being detected. Higher thresholds yield an increasing percentage of strongly coupled mutants, but the total number of mutants in the subset becomes so small that the subset would lose its utility to generally assess the sensitivity of test cases to changes to the code.

Using median level of coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

5.5 Threats to Validity

External Validity: Our study has focused on five open-source Java projects—a relatively small number—from a single fault database, Defects4J. Therefore, we cannot claim that our results will generalize to all types of faults or all systems in all programming languages. Nevertheless, we believe that the projects studied are representative of, at minimum, other small to medium-sized Java systems. The projects are popular case examples, that have been used in many other software testing studies.

We also believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar, projects. As Defects4J has been applied in many other research studies, the use of this dataset allows comparisons of our results with related research, and allows others to more easily replicate or extend our experiments.

Internal Validity: To control experiment cost, we have used 31 mutation operators from a single mutation testing framework. Other frameworks may offer different mutation operators, which could result in different conclusions. However, muJava++ was the only framework that met all of our experimental constraints. muJava++ offers a large number and variety of operators—substantially more than Major, and substantially overlapping with PIT, the other commonly-used Java mutation frameworks. muJava++ also offers mutation operators not available in PIT. We believe that the operators we have employed are sufficiently varied to lead to meaningful conclusions.

We use only the developer-written test suites to examine the semantic coupling between mutants and real faults. These test cases do not form a complete specification of the behavior of code, meaning that strong coupling with respect to the existing developer-written tests may not hold with the addition of more test cases. However, the developer-written tests reflect that situations that the developers felt were most important to consider. In most cases, for the studied case examples, the test suites are extensive. Therefore, we feel that the developer-written test cases are appropriate for use for determining the coupling between mutants and faults.

5.6 Related Work

Mutation testing relies on the core premise that mutations can serve as stand-ins for real faults for analyses related to fault detection. Researchers have examined this premise for years, focusing on two separate—but highly related—questions. First, *what is the nature of the relationship between mutants and real faults?* Second, regardless of that nature, asks

whether mutants be used in the place of real faults. In other words, *is there a correlation between mutant and fault detection?*

Below, we discuss a subset of related work on these two questions. Our study focuses on the first question, assessing the degree of coupling between mutation operators and real faults. Broadly, we differ from the previous work in our methodology for examining this relationship. Our focus is on semantic similarity, rather than syntactic. In addition, we propose the use of a scale of degrees of coupling, rather than a simple binary assessment. We also perform a large-scale experiment, making use of many real faults for a variety of complex Java projects—each containing many different classes. We examine a larger number of mutation operators than related work. Collectively, these factors enable a thorough analysis of how particular mutation operators relate to real faults. We do not examine whether there is a correlation between mutant and fault detection. However, our research is complementary in that the degree of coupling between a mutation operator and real faults could impact the correlation between mutant and fault detection.

DeMillo at al. were the first to propose the coupling hypothesis [30], showing that tests that detect one-line mutants also can detect a more complex multi-line mutant. Their demonstration was simple, but provided evidence for the validity of mutation testing. Another early study, by Duran et al., examined a set of 24 mutants and 12 real faults for a safety-critical C program [198]. They found that mutants replicated all errors—due to either incorrect internal state or output—produced by real faults for this program, and that even simple mutations could create complex erroneous behavior. They conclude that mutations can serve as stand-ins for real faults. Their assessment is simple, and does not examine the role of mutation operators and their coupling with particular faults.

Gopinath et al. [31] studied the syntactic similarity of mutation operators and real faults in C, Java, Python, and Haskell. They found that the differences between fixed and buggy program versions generally involve three to four changes, and those changes are usually not equivalent to edits made by traditional mutation operators. Their findings somewhat

dispute the "competent programmer hypothesis", as they observe that the syntactic difference between buggy and fixed programs is often significant. They also find that different languages have different distributions of code patterns and that mutation operators optimal in one language may not be optimal for others. We differ in our focus on the semantic relationship rather than syntactic. Although there are large syntactic differences, the semantic differences are more subtle and varied in their magnitude.

Chekam et al. proposed the use of machine learning to identify the characteristics that make mutants valuable in revealing real faults, with the goal of reducing the number of mutants used during mutation testing [199]. A model is trained using a supervised learning algorithm that predicts which mutants are more fault-revealing, and those mutants are retained. They trained their model using data from three sets of real faults—Defects4J (Java), Codeflaws (C), and CoREBench (C)—and mutants generated for seven mutation operators. They do not examine the relationship between mutants and real faults directly in their research. However, their model—or other machine learning models—could potentially be used as an alternative way to perform such an assessment.

Jimenez et al. examine the "naturalness" of mutants, the degree to which mutants match the implicit coding norms of a project, with the hypothesis that mutants that are (a) natural, and (b), have a large semantic difference with the original program will be more valuable in assessing the ability of a test suite to detect faults [200]. They examined the impact of naturalness on which mutants were detected by tests that also detect real faults. Their results were negative—they found that naturalness was independent of fault detection and played no significant role in the coupling between mutations and real faults.

Many researchers have examined whether tests that detect mutants tend to also detect real faults. First, Andrews et al. examine whether mutants can be used instead of real faults in testing experiments [26, 201], focusing on a set of eight C programs with hand-seeded faults and an additional program with real faults. They focus on the question of whether the percentage of detected mutants—generated using nine operators—is predictive of the

ratio of real faults detected. Their results suggest that, when using appropriate mutation operators for the program and removing equivalent mutants, mutant detection predicts for fault detection. They also find that real faults were easier to detect than the hand-seeded faults. Their focus is on this correlation, and they do not examine the relationship between mutation operators and faults, or provide advice for selecting mutation operators.

Just et al. [24] found a statistically significant correlation between mutant and fault detection using both developer-written and generated test cases. They also use Defects4J as their set of real faults, and generate mutants for five mutation operators. They find that mutation detection offers a stronger correlation to fault detection than code coverage of test cases. Relevant to our research, they do examine the relationship between mutants and real faults. However, their definition of coupling is simpler than ours—if a test fails when executed on a mutant and a real fault, the mutant and fault are coupled. In addition, they generate mutants for the full code class affected by a fault, while we only generate mutants for the lines of code affected by a fault, and they only use a small number of mutation operators. They find that 73% of real faults are coupled to at least one mutant, but the number of mutants coupled to each fault is small when code coverage is controlled. Moreover, conditional operator replacement, relational operator replacement, and statement deletion mutants are more often coupled to real faults than other operators. They suggested strengthening certain mutant operators, as well as adding mutation operators.

Papadakis et al. performed a follow-up study to the one by Just et al., with an expanded evaluation including both Java and C faults and an alternative methodology [187]. They find that, when test suite size is controlled, the correlation between mutant and fault detection is weak. They examine the coupling between mutants and real faults using a similarity measurement based on test failures and code coverage. Mutants that affect the same statements as a real fault and that are detected by tests that detect the real fault are considered more similar. They find that less than 1% of mutants represent the behavior of real faults, reducing the potential correlation. In contrast to our study, they do not consider the reasons

that tests fail and consider tests independent of each other, and they also do not examine the mutation operators.

Kim et al. further this line of research by investigating an additional factor, which code has been mutated [202]. They explore the influence of code location at the class, method, and statement levels when computing the correlation between mutant and fault detection. They found that the granularity level had a significant influence on correlation, and that test suite size influenced the correlation differently at each granularity level. In particular, filtering mutants based on method and statement levels increased the likelihood of fault detection. They suggest locating error-prone code and prioritizing mutants that affect that code. They also do not examine the coupling between mutation operators and real faults.

Kintis et al. contrast the fault-revealing power of four Java mutation tools (i.e., if developers use mutants from a tool to select test cases, those test cases would also be sufficient to detect faults) [203]. They consider a fault revealed when at least one generated mutant is killed only by the triggering tests (equivalent to either "test" or "strong substitution" in our coupling scale). In their experiments, a research version of the PIT mutation tool was the most effective, revealing 97% of real faults. However, no single tool subsumes all others. Their goal was to contrast the fault-revealing capabilities of mutation tools, so they do not perform an analysis of the relation of mutation operators to real faults.

Recently, researchers have begun to investigate techniques that can synthesize new mutation operators, generally based on machine learning and natural language processing [204–209]. The hypothesis behind much of this work is that the learned mutation operators will be more realistic than traditional mutation operators, increasing their utility. Generally, these techniques are based on the syntactic similarity of mutations to real faults, favoring mutants that replicate patterns in the textual differences between buggy and fixed programs. However, Patra et al. extend a syntactic model with additional semantic metadata that adapts mutants to a local code context [207].

A common assumption is that mutants that are syntactically similar to real faults will

also be semantically similarity. Ojdanic et al. investigate whether this hypothesis is true [209]. They use the same notion of semantic similarity used previously by Papadakis et al. [187]. Their experiments suggest that syntactic similarity of mutants to real faults has no predictive relationship to semantic similarity.

5.7 CONCLUSIONS

We hypothesize that improving the effectiveness—in terms of both cost and quality—of mutation testing lies in better understanding the semantic relationship between mutants and real faults. In particular, we examine the degree of coupling that specific mutation operators have with real faults, based on 144 case examples from the Defects4J fault database and 31 mutation operators offered by the muJava++ framework. We have defined a scale rating the strength of the coupling between a mutant and a corresponding fault, based on number of failing tests and reasons for failure.

Ultimately, we observed that 9.92% of the mutants are strongly coupled to real faults, and 51.03% of the faults have at least one strongly coupled mutant. EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median level of coupling. ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors. JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the the largest percentage of mutants that are not detected. SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the the largest percentage of mutants that are only detected by non-trigger tests. These mutants *are* detected, but lack a significant relationship with the corresponding real faults. We also found that using the median coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

Understanding the semantic relationship between mutation operators and faults could enable improvements in how mutation testing is applied, improved implementation of specific mutation operators, and inspiration for new mutation operators. We plan to explore all three avenues further in future work by (1) further analyzing the operators identified above in experiments on additional fault examples, (2) expanding the range of mutation operators considered and contrasting implementations of operators from different frameworks, (3) empirically evaluating cost savings and impact on mutation score from different filtering methods, and (4), exploring how semantic coupling could be used as the basis for the design and automated generation (via machine learning) of new mutation operators.

CHAPTER 6

CONCLUSION

In this dissertation, we addressed challenges that hinder the attainment of the long-term goal of reducing the cost of software testing by doing *empirical studies* relating to important topics in the software testing field, with a particular focus on automation. We focused on (1) mapping the connections between research topics and understanding the evolution of research topics in the field of software testing through the use of network analysis techniques, (2) assessment of the criteria used to guide automated test input generation and exploration of the factors that influence the ability of automated input generation to trigger failures, and (3) examination of the semantic coupling between synthetic and real-world faults to identify the types of synthetic faults best suited for use in assessing and improving test case quality.

As discussed in Chapter 3, we applied network analysis techniques to quantitatively map the software testing research field. We offered an evidence-based method to characterize research topics in software testing and, more importantly, to identify how these topics are connected to explore how to exploit existing topic synergies best or identify new connections to explore. The findings of this project help motivate further research on the automation of the testing process. Our analysis mapped keywords into dense clusters, from which emerge high-level research topics—themes that characterize each cluster—and made clear the connections between keywords and topics within and across clusters. We also characterized the periods in which low-level keywords and high-level topics have emerged—identifying emerging research areas, as well as those where research interest has decreased. This snapshot of important disciplinary trends can provide valuable insight into

the state of the field, suggest topics to explore, and identify connections (or lack thereof) between keywords and topics that may reveal new insights.

In this project, we made the following observations:

- Both the most common author-assigned keywords and the keywords that attract the most citations, on average, tend to relate to automation, test creation and assessment guidance, assessment of system quality, and cyber-physical systems.
- These keywords can be clustered into 16 topics: automated test generation, creation guidance, evolution and maintenance, machine learning and predictive modeling, model-based testing, GUI testing, processes and risk, random testing, reliability, requirements, system testing, test automation, test case types, test oracles, verification and program analysis, and web application testing. Below these lie 18 more subtopics.
- Creation guidance, automated test generation, evolution and maintenance, and test oracles are particularly multidisciplinary topics with dense connections to many other topics. Twenty keywords connect topics, reflecting multidisciplinary concepts, common test activities, and test creation information.
- Emerging research particularly relates to web and mobile applications, ML and AI—including autonomous vehicles—energy consumption, automated program repair, or fuzzing and search-based test generation. Web applications require targeted testing approaches and practices, leading to emerging connections to many topics. Test oracles are also a rapidly-evolving topic with many emerging connections. ML has emerging potential to support automation.
- Research related to random and requirements-based testing may be in decline.

These insights—and the rich underlying networks of keywords—can inspire both current and future researchers in the field of software testing.

Furthermore, as discussed in Chapter 4, we examined whether common fitness functions can produce effective test input for triggering and detecting real-world faults. We shed light on the factors contributing to automated input generation's success or failure. In this study, we used EvoSuite and eight of its white-box fitness functions (as well as the default multi-objective configuration and a combination of branch, exception, and method coverage) to generate test suites for the fifteen systems, and 593 of the faults, in the Defects4J database. In each case, we tried to understand *when and why* generated test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of current test generation techniques' strengths and limitations and inspire new approaches. Thus, in each case, we recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. We recorded a set of traditional *source code metrics*—sixty metrics related to cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics—for each class associated with a fault in the Defects4J dataset. By analyzing these generation factors and metrics, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of machine learning algorithms—the factors correlating with a high or low likelihood of fault detection.

We examined the fitness function's role in determining search-based test generators' ability to produce suites that detect complex, real faults. From the eight fitness functions and 593 faults studied, we can conclude:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak

mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.

- While EvoSuite's default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include a high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases, where a fault is consistently detected, is the satisfaction of the chosen criterion's test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and

manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault and ensures that it manifests in failure. Criteria such as exception, output, and weak mutation coverage are situationally useful and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations provide evidence for the anecdotal findings of other researchers [19–23] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. More research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites. In future work, we plan to further explore these topics.

Finally, as discussed in Chapter 5, we investigated the degree of coupling between mutants and real faults by executing developer-written test suites against both mutated and faulty versions of classes from multiple open-source Java projects, based on 144 case examples from the Defects4J fault database [32]. In particular, we focused on the *trigger tests*—the tests that detect the real fault. A mutant that is most strongly coupled to a real fault will be detected only by the trigger tests, and those tests will fail for the same reasons—i.e., the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons. We defined a scale rating as the strength of the coupling between a mutant and a corresponding

real fault based on the number of failing tests and reasons for failure. This scale, in turn, allows us to contrast 31 mutation operators—applied using the muJava++ framework—based on their tendency to produce mutants with a stronger semantic relationship to real faults.

Ultimately, we observed that 9.92% of the mutants are strongly coupled to real faults, and 51.03% of the faults have at least one strongly coupled mutant. EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median level of coupling. ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors. JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the largest percentage of mutants that are not detected. SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the largest percentage of mutants that are only detected by non-trigger tests. These mutants *are* detected but lack a significant relationship with the corresponding real faults. We also found that using the median coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

Understanding the semantic relationship between mutation operators and faults could enable improvements in how mutation testing is applied, improved implementation of specific mutation operators, and inspiration for new mutation operators. We plan to explore all three avenues further in future work by (1) further analyzing the operators identified above in experiments on additional fault examples, (2) expanding the range of mutation operators considered and contrasting implementations of operators from different frameworks, (3) empirically evaluating cost savings and impact on mutation score from different filtering methods, and (4), exploring how semantic coupling could be used as the basis for the design and automated generation (via machine learning) of new mutation operators.

BIBLIOGRAPHY

[1] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[2] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[3] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[4] Mary Jean Harrold. Testing: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 61–72, New York, NY, USA, 2000. Association for Computing Machinery.

[5] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007.

[6] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[7] Gregory Gay, Tim Menzies, Omid Jalali, Gregory Mundy, Beau Gilkerson, Martin Feather, and James Kiper. Finding robust solutions in requirements models. *Automated Software Engineering*, 17(1):87–116, 2010.

[8] Clark S. Turner Nancy G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 1993.

[9] A Anand and A Uddin. Importance of software testing in the process of software development. *International Journal for Scientfic Research and Development*, 12(6), 2019.

[10] Matthew Heusser and Govind Kulkarni. *How to reduce the cost of software testing*. CRC Press, 2018.

[11] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[12] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education, 2009.

[13] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE, 2012.

[14] M. Harman and B.F. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[15] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[16] Edward Kit and Susannah Finzi. *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[17] William Perry. *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

[18] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder K Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.

[19] A. Arcuri, G. Fraser, and R. Just. Private api access and functional mocking in automated unit test generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, volume 00, pages 126–137, March 2017.

[20] Gregory Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing*, ICST 2017. IEEE, 2017.

[21] Gregory Gay. Challenges in using search-based test generation to identify real faults in mockito. In *Search Based Software Engineering: 8th International Symposium,*

SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings, pages 231–237, Cham, 2016. Springer International Publishing.

[22] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.

[23] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 79–90, New York, NY, USA, 2014. ACM.

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 654–665, Hong Kong, November 18–20, 2014.

[25] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2 edition, 2016.

[26] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608 –624, aug. 2006.

[27] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[28] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, pages 78–84, 2006.

[29] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 163–171, New York, NY, USA, 2018. Association for Computing Machinery.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[31] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.

[32] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[33] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[34] Alireza Salahirad, Gregory Gay, and Ehsan Mohammadi. Mapping the structure and evolution of software testing research over the past three decades. *Journal of Systems and Software*, 195:111518, 2023.

[35] Hussein Almulla, Alireza Salahirad, and Gregory Gay. Using search-based test generation to discover real faults in guava. In *International Symposium on Search Based Software Engineering*, pages 153–160. Springer, 2017.

[36] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019.

[37] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 30(7-8):e1758, 2020.

[38] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[39] Mordechai Ben-Ari. The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, 33(2):58–59, 2001.

[40] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, pages 1–11, 2012.

[41] James A Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.

[42] James A Whittaker. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education, 2009.

[43] Itir Karac and Burak Turhan. What do we (really) know about test-driven development? *IEEE Software*, 35(4):81–85, 2018.

[44] A. Turing. *Checking a Large Routine*, page 70–72. MIT Press, Cambridge, MA, USA, 1989.

[45] Santo Fortunato, Carl T. Bergstrom, Katy Börner, James A. Evans, Dirk Helbing, Staša Milojević, Alexander M. Petersen, Filippo Radicchi, Roberta Sinatra, Brian Uzzi, Alessandro Vespignani, Ludo Waltman, Dashun Wang, and Albert-László Barabási. Science of science. *Science*, 359(6379), 2018.

[46] Christine L. Borgman and Jonathan Furner. Scholarly communication and bibliometrics. *Annual Review of Information Science and Technology*, 36(1):2–72, 2002.

[47] Henk F Moed. *Citation analysis in research evaluation*, volume 9. Springer Science & Business Media, 2006.

[48] Ying Ding, Xiaozhong Liu, Chun Guo, and Blaise Cronin. The distribution of references across texts: Some implications for citation analysis. *Journal of Informetrics*, 7(3):583–592, 2013.

[49] Alan Pritchard et al. Statistical bibliography or bibliometrics. *Journal of documentation*, 25(4):348–349, 1969.

[50] Nicola De Bellis. *Bibliometrics and citation analysis: from the science citation index to cybermetrics*. scarecrow press, 2009.

[51] Naveen Donthu, Satish Kumar, and Debidutta Pattnaik. Forty-five years of journal of business research: a bibliometric analysis. *Journal of Business Research*, 109:1–14, 2020.

[52] Nees Jan Van Eck and Ludo Waltman. Visualizing bibliometric networks. In *Measuring scholarly impact*, pages 285–320. Springer, 2014.

[53] H.P.F. Peters and A.F.J. van Raan. Co-word-based science maps of chemical engineering. part i: Representations by direct multidimensional scaling. *Research Policy*, 22(1):23–45, 1993.

[54] Edmund Whittaker. *A History of the Theories of Aether and Electricity: Vol. I: The Classical Theories; Vol. II: The Modern Theories, 1900-1926*, volume 1. Courier Dover Publications, 1989.

[55] Hsin-Ning Su and Pei-Chun Lee. Mapping knowledge structure by keyword co-occurrence: a first look at journal papers in technology foresight. *Scientometrics*, 85(1):65–79, 2010.

[56] Luz M Romo-Fernández, Vicente P Guerrero-Bote, and Félix Moya-Anegón. Co-word based thematic analysis of renewable energy (1990–2010). *Scientometrics*, 97(3):743–765, 2013.

[57] Werner Marx, Robin Haunschild, and Lutz Bornmann. Global warming and tea production—the bibliometric view on a newly emerging research topic. *Climate*, 5(3), 2017.

[58] Ehsan Mohammadi. Knowledge mapping of the iranian nanoscience and technology: a text mining approach. *Scientometrics Scientometrics*, 92(3):593 – 608, 01 Sep. 2012.

[59] Yong Liu, Jorge Goncalves, Denzil Ferreira, Bei Xiao, Simo Hosio, and Vassilis Kostakos. Chi 1994-2013: Mapping two decades of intellectual progress through co-word analysis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 3553–3562, New York, NY, USA, 2014. Association for Computing Machinery.

[60] Huchang Liao, Ming Tang, Li Luo, Chunyang Li, Francisco Chiclana, and Xiao-Jun Zeng. A bibliometric analysis and visualization of medical big data research. *Sustainability*, 10(1), 2018.

[61] Ehsan Mohammadi and Amir Karami. Exploring research trends in big data across disciplines: A text mining analysis. *Journal of Information Science*, 0(0):0165551520932855, 2020.

[62] Vahid Garousi and Mika V. Mäntylä. Citations, research topics and active countries in software engineering: A bibliometrics study. *Computer Science Review*, 19:56–77, 2016.

[63] Vahid Garousi and João M. Fernandes. Quantity versus impact of software engineering papers: a quantitative study. *Scientometrics*, 112(2):963–1006, 2017.

[64] Dimitra Karanatsiou, Yihao Li, Elvira-Maria Arvanitou, Nikolaos Misirlis, and W. Eric Wong. A bibliometric assessment of software engineering scholars and institutions (2010–2017). *Journal of Systems and Software*, 147:246–261, 2019.

[65] W. Eric Wong, T.H. Tse, Robert L. Glass, Victor R. Basili, and T.Y. Chen. An assessment of systems and software engineering scholars and institutions (2001–2005). *Journal of Systems and Software*, 81(6):1059–1062, 2008. Agile Product Line Engineering.

[66] W. Eric Wong, T.H. Tse, Robert L. Glass, Victor R. Basili, and T.Y. Chen. An assessment of systems and software engineering scholars and institutions (2002–2006). *Journal of Systems and Software*, 82(8):1370–1373, 2009. SI: Architectural Decisions and Rationale.

[67] W. Eric Wong, T.H. Tse, Robert L. Glass, Victor R. Basili, and T.Y. Chen. An assessment of systems and software engineering scholars and institutions (2003–2007 and 2004–2008). *Journal of Systems and Software*, 84(1):162–168, 2011. Information Networking and Software Services.

[68] Vahid Garousi and Tan Varma. A bibliometric assessment of canadian software engineering scholars and institutions (1996-2006). *Computer and Information Science*, 3(2):19, 2010.

[69] Vahid Garousi. A bibliometric analysis of the turkish software engineering research community. *Scientometrics*, 105(1):23–49, 2015.

[70] Roshanak Farhoodi, Vahid Garousi, Dietmar Pfahl, and Jonathan Sillito. Development of scientific software: A systematic mapping, a bibliometrics study, and a paper repository. *International Journal of Software Engineering and Knowledge Engineering*, 23(04):463–506, 2013.

[71] Fabrício Gomes de Freitas and Jerffeson Teixeira de Souza. Ten years of search based software engineering: A bibliometric analysis. In Myra B. Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering*, pages 18–32, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[72] Nees Jan Van Eck and Ludo Waltman. Software survey: Vosviewer, a computer program for bibliometric mapping. *scientometrics*, 84(2):523–538, 2010.

[73] Nees Jan van Eck and Ludo Waltman. *Visualizing Bibliometric Networks*, pages 285–320. Springer International Publishing, Cham, 2014.

[74] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[75] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[76] Mark Fewster and Dorothy Graham. *Software test automation*. Addison-Wesley Reading, 1999.

[77] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[78] J. Voas. Fault injection for the masses. *Computer*, 30(12):129–130, 1997.

[79] B. Kitchenham and S.L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE Software*, 13(1):12–21, 1996.

[80] W Herzner, R Schlick, A Le Guennec, and B Martin. Model-based simulation of distributed real-time applications. In *5th IEEE Int'l Conf. on Industrial Infomatics*, pages 989 – 994, June 2007.

[81] Aloysius K. Mok and Douglas Stuart. Simulation vs. verification: Getting the best of both worlds. In *Proceedings of the Eleventh Annual Conf. on Computer Assurance, COMPASS 96*, 1996.

[82] G. Gay, S. Rayadurgam, and M. P. E. Heimdahl. Automated steering of model-based test oracles to admit real program behaviors. *IEEE Transactions on Software Engineering*, 43(6):531–555, June 2017.

[83] Mark Crossley. *The Desk Reference of Statistical Quality Methods*. ASQ Quality Press, 2000.

[84] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[85] G. Gay, M. Staats, M. Whalen, and M.P.E. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.

[86] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2), February 2011.

[87] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[88] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019. e1701 stvr.1701.

[89] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[90] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*. CRC press, 2011.

[91] Toshio Fukuda. Theory and applications of neural networks for industrial control systems. *IEEE Transactions on Industrial Electronics*, pages 472–489, December 1992.

[92] A. van Lamsweerde. Engineering requirements for system reliability and security. *Software System Reliability and Security*, 9, 2007.

[93] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.

[94] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb 2002.

[95] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. on Software Engineering*, 17(7):703–711, 1991.

[96] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[97] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[98] Andrea Arcuri and Lionel C. Briand. Adaptive random testing: An illusion of effectiveness? In *ISSTA*, 2011.

[99] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[100] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[101] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

[102] Lee Copeland. *A practitioner's guide to software test design*. Artech House, 2004.

[103] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[104] Bestoun S Ahmed, Eduard Enoiu, Wasif Afzal, and Kamal Z Zamli. An evaluation of monte carlo-based hyper-heuristic for interaction testing of industrial embedded software applications. *Soft Computing*, 24(18):13929–13954, 2020.

[105] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, 2019.

[106] Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[107] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[108] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A survey on data-flow testing. *ACM Computing Surveys (CSUR)*, 50(1):1–35, 2017.

[109] RTCA/DO-178C. Software considerations in airborne systems and equipment certification.

[110] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 210–220, 2012.

[111] Dominique Douglas-Smith, Takuya Iwanaga, Barry FW Croke, and Anthony J Jakeman. Certain trends in uncertainty and sensitivity analysis: An overview of software tools and techniques. *Environmental Modelling & Software*, 124:104588, 2020.

[112] Mustafa Bozkurt, Mark Harman, Youssef Hassoun, et al. Testing web services: A survey. *Department of Computer Science, King's College London, Tech. Rep. TR-10-01*, 2010.

[113] Imran Ali Qureshi and Aamer Nadeem. Gui testing techniques: a survey. *International Journal of Future computer and communication*, 2(2):142, 2013.

[114] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[115] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, Aug 2017.

[116] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper. Machine learning to guide performance testing: An autonomous test framework. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 164–167, 2019.

[117] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of Int'l Symposium on Software Testing and Analysis*, pages 25–36. ACM, July 2006.

[118] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 2008.

[119] A.P. Sistla E. M. Clarke, E.A. Emerson. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages 244–263, April 1986.

[120] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In *Search-Based Software Engineering*, pages 3–45, Cham, 2018. Springer International Publishing.

[121] R. France and B. Rumpe. Model-driven development of complex systems: A research roadmap. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[122] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[123] Afonso Fontes and Gregory Gay. Using machine learning to generate test oracles: A systematic literature review. In *Proceedings of the 1st International Workshop on Test Oracles*, TORACLE 2021, page 1–10, New York, NY, USA, 2021. Association for Computing Machinery.

[124] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, 2013.

[125] Mike Thelwall and Pardeep Sud. Scopus 1900–2020: Growth in articles, abstracts, countries, fields, and journals. *Quantitative Science Studies*, pages 1–14, 02 2022.

[126] Antonio Cavacini. What is the best database for computer science journal articles? *Scientometrics*, 102(3):2059–2071, 2015.

[127] H.R. Jamali, C.C. Steel, and E. Mohammadi. Wine research and its relationship with wine production: a scientometric analysis of global trends. *Australian Journal of Grape and Wine Research*, 26(2):130–138, 2020.

[128] Huajiao Li, Haizhong An, Yue Wang, Jiachen Huang, and Xiangyun Gao. Evolutionary features of academic articles co-keyword network and keywords co-occurrence network: Based on two-mode affiliation network. *Physica A: Statistical Mechanics and its Applications*, 450:657–669, 2016.

[129] Ingwer Borg and Patrick JF Groenen. *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

[130] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[131] Ludo Waltman and Nees Jan van Eck. A smart local moving algorithm for large-scale modularity-based community detection. *The European Physical Journal B*, 86(11):471, 2013.

[132] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[133] Akbar Siami Namin and James H Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68, 2009.

[134] A. Mockus, N. Nagappan, and T.T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301, Oct 2009.

[135] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[136] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, Feb 2013.

[137] Gregory Gay. Generating effective test suites by combining coverage criteria. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.

[138] Leonora Bianchi, Marco Dorigo, LucaMaria Gambardella, and WalterJ. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[139] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[140] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[141] Robert Feldt and Simon Poulding. Broadening the search in search-based software testing: It need not be evolutionary. In *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, pages 1–7, May 2015.

[142] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

[143] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'14, pages 255–268, New York, NY, USA, 2014. ACM.

[144] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[145] Andrea Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[146] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA, pages 291–301, New York, NY, USA, 2013. ACM.

[147] Hussein Almulla, Alireza Salahirad, and Gregory Gay. Using search-based test generation to discover real faults in Guava. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.

[148] K. Molokken and M. Jorgensen. A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, pages 223–230, Sept 2003.

[149] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A.C. Rocha, Carmelo M. Torre, David Taniar, Bernady O. Apduhan, Elena Stankova, and Shangguang Wang, editors, *Computational Science and Its Applications – ICCSA 2016*, pages 625–638, Cham, 2016. Springer International Publishing.

[150] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[151] G. Gui and P. D. Scott. Coupling and cohesion measures for evaluation of component reusability. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 18–21, New York, NY, USA, 2006. ACM.

[152] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM.

[153] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[154] P. Edith Linda, V. Manju Bashini, and S. Gomathi. Metrics for component based measurement tools. In *International Journal of Scientific & Engineering Research Volume 2, Issue 5*, 2011.

[155] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, Nov 1983.

[156] SourceMeter. Sourcemeter java documentation. https://www.sourcemeter.com/resources/java/, 2014.

[157] Nadia Alshahwan and Mark Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.

[158] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[159] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In Márcio Barros and Yvan Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.

[160] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, 25(3):25:1–25:34, July 2016.

[161] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[162] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012.

[163] Tim Menzies and Ying Hu. Data mining for very busy people. *Computer*, 36(11):22–29, November 2003.

[164] Gregory Gay, Tim Menzies, Misty Davies, and Karen Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engineering*, 17(4):439–468, December 2010.

[165] W.W. Daniel. *Applied Nonparametric Statistics*. Duxbury advanced series in statistics and decision sciences. PWS-KENT, 1990.

[166] M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM, May 2013.

[167] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST '05, pages 1–7, New York, NY, USA, 2005. ACM.

[168] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 154–164, New York, NY, USA, 1991. ACM.

[169] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, Aug 1993.

[170] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 153–162, New York, NY, USA, 1998. ACM.

[171] Phil McMinn, Mark Harman, Gordon Fraser, and Gregory M. Kapfhammer. Automated search for good coverage criteria: Moving from code coverage to fault coverage through search-based software engineering. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 43–44, New York, NY, USA, 2016. ACM.

[172] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *CoRR*, abs/1703.00198, 2017.

[173] Gregory Gay. To call, or not to call: Contrasting direct and indirect branch coverage in test generation. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, SBST 2018, New York, NY, USA, 2018. ACM.

[174] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, New York, NY, USA, 2007. ACM.

[175] Shin Yoo and Mark Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689 – 701, 2010.

[176] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, Feb 2007.

[177] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 2018.

[178] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Incremental control dependency frontier exploration for many-criteria test case generation. In Thelma Elita Colanzi and Phil McMinn, editors, *Search-Based Software Engineering*, pages 309–324, Cham, 2018. Springer International Publishing.

[179] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[180] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Empirical validation of

object-oriented metrics for predicting fault proneness models. *Software quality journal*, 18(1):3, 2010.

[181] Usman Mansoor, Marouane Kessentini, Bruce R Maxim, and Kalyanmoy Deb. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, 25(2):529–552, 2017.

[182] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 49–58. ACM, 2012.

[183] Ashish Tripathi and Kapil Sharma. Improving software quality based on relationship among the change proneness and object oriented metrics. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 1633–1636. IEEE, 2015.

[184] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A.C. Rocha, Carmelo M. Torre, David Taniar, Bernady O. Apduhan, Elena Stankova, and Shangguang Wang, editors, *Computational Science and Its Applications – ICCSA 2016*, pages 625–638, Cham, 2016. Springer International Publishing.

[185] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *arXiv preprint arXiv:1801.06393*, 2018.

[186] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[187] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 537–548, New York, NY, USA, 2018. Association for Computing Machinery.

[188] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pages 402–411, 2005.

[189] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[190] Richard A. De Millo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[191] Brian Marick. The weak mutation hypothesis. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 190–199, New York, NY, USA, 1991. ACM.

[192] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[193] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.

[194] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery.

[195] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: a mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, pages 827–830, 2006.

[196] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[197] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. Mudelta: Delta-oriented mutation testing at commit time. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[198] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes*, 21(3):158–171, 1996.

[199] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[200] Matthieu Jimenez, Thiery Titcheu Checkam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural? a study on how "naturalness" helps mutant selection. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[201] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings. 27th International Conference on Software Engineering*, pages 402–411, 2005.

[202] Mingwan Kim, Neunghoe Kim, and Hoh Peter In. Investigating the relationship between mutants and real faults with respect to mutated code. *International Journal of Software Engineering and Knowledge Engineering*, 30(08):1119–1137, 2020.

[203] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, 2018.

[204] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522, 2017.

[205] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2019.

[206] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. Deepmutation: a neural mutation tool. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 29–33. IEEE, 2020.

[207] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 906–918, 2021.

[208] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 268–277. IEEE, 2021.

[209] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies. *arXiv preprint arXiv:2112.14508*, 2021.