AUTOMATED TECHNIQUES FOR IMPROVING THE QUALITY OF EXISTING TEST SUITES

by

Chen Huo

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Fall 2017

AUTOMATED TECHNIQUES FOR IMPROVING THE QUALITY OF EXISTING TEST SUITES

by

Chen Huo

Approved: _____

Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Lori Pollock, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

René Just, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stephen Siegel, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. James Clause for his support, advice and encouragement. Dr. Clause not only provides me with valueable advice and encouragement in research but also is an excellent role model on how to balance life and stress. I would also like to thank my committee members, Drs. Lori Pollock, Rene just, Stephen Siegel, for all of their assistance and support.

I appreciate the help by our department staff, including Teresa Louise, Samantha Fowle and Gregory Lynch. I feel very fortunate to study in a group of amazing colleagues, Benwen Zhang, Cagri Sahin, Irene Manotas and Xiaoran Wang, for academic discussions and fun moments in life.

I would also like to thank Dr. Paul Gray for encouraging me to pursue a doctorate degree, providing useful advice along the way.

Finally, I would like to thank my wife, Siyan, for her love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Testing is playing a crucial and fundamental role in modern software development. Although software tests are conceptually simple—they are composed of two primary parts: *inputs* that are used to execute the program under test and an *oracle* that is used to verify that the execution induced by the inputs produces the expected results—they are often difficult to write in practice. The software engineering research community provided many techniques that can help developers determine whether they have written effective and efficient tests, including various coverage metrics which have been widely adopted. While they have been proven successful in practice, many follow-up studies show that there is still a lot to improve for the test quality measurements, including false negatives in alarming the lack of tests and constructive suggestions for improvements.

This dissertation focuses on improving the quality of existing test suites based on interpretations on test inputs and test oracles. If a test oracle checks the values which the test developers did not or can not control, it would make the test brittle. I developed a novel technique based on dynamic tainting which can identify the values that can make tests brittle. An empirical study on real-world applications shows that the technique can reveal brittle assertions and the values that cause the brittleness within reasonable cost. I also developed two techniques that can identify insufficiently tested code by interpreting traditional coverage information. One is a new approach based on the concepts of direct coverage and indirect coverage. The other is a new approach to discover incidentally tested code. Both techniques have shown efficiency and effectiveness in the empirical studies on real-world applications.

Chapter 1

INTRODUCTION

Testing is one of the primary methods developers use to judge the correctness of software by attempting to expose failures and errors before they impact users. Testing also provides numerous other benefits such as assessing software quality, enabling large scale changes and serving as a form of documentation. Therefore, according to a recent study of practicing software developers [34], many developers have a strong desire for more tests in their projects. Despite decades of work devoted to developing techniques that attempt to help developers test software, testing remains an expensive and laborious activity. Some reports estimate that the costs associated with testing can account for more than 50 % of the total cost of developing software [6]. In the future, such high costs are likely to persist or even increase as the growing size and complexity of modern software exacerbates existing challenges. Therefore, additional techniques for improving the efficiency and effectiveness of the testing process will be extremely beneficial in reducing the overall cost of software development and, at the same time, improving software quality.

The software engineering research community has provided many techniques that can help developers write test cases, including various testing paradigms as well as automatic test generation techniques (e.g., [33, 35, 45, 82, 91, 92, 97, 100, 115, 124]). While such techniques can be successful in helping developers write tests, they only address part of the overall problem. In order to provide more help, it is necessary to not only provide developers with help creating new tests but also improving the quality of existing ones.

In this dissertation, I first present a novel dynamic analysis technique that addresses both test brittleness and unused inputs. I consider the inputs as controlled

if they are explicitly provided by the test itself (e.g., constants that appear in the test method) and all other inputs are considered uncontrolled. The technique is based on dynamic tainting and works by tracking the flow of controlled and uncontrolled inputs along data- and control-dependencies at runtime. When a test finishes execution, the technique uses the tracked information to generate reports that identify *brittle assertions*—assertions that check values that are derived from inputs that are not controlled by the test and *unused inputs*—inputs that are controlled by the test but are not checked by an assertion. These reports are then filtered to remove false positives and presented to testers.

The next two techniques to be introduced in this dissertation are based on different interpretations of coverage criteria. The research community provides many pragmatic coverage criteria to measure the effectiveness of test suites, including e.g., [9, 20, 58, 70, 79, 88, 93, 95, 112]. Coverage criteria are often used as coverage adequacy metrics in the software industry by measuring how much of a certain criterion has been met with a test suite [4]. In the context of adequacy metrics, coverage criteria would suggest which part of code to test by indicating which entities (e.g., methods, statements, branches, etc.) in a program are executed (covered) by a certain test suite and which are not. Presumably, uncovered entities indicate deficiencies in a test suite. However, covered entities might be insufficiently tested.

In this dissertation, I introduce a new approach, based on the concepts of direct coverage and indirect coverage, for interpreting coverage information. The goal of the approach is to help developers focus their limited testing resources on insufficiently tested code. At a high-level, the approach identifies methods that contain a high proportion of indirectly covered entities as being insufficiently tested. In addition to taking into account how entities are covered by a test suite, the approach also eliminates the need for testers to manually identify whether the code indicated by the approach can be executed. Because they are covered by the test suite, the methods identified by the approach are guaranteed to be feasible. This means that developers do not have to spend time investigating whether it is possible to execute the identified code.

The third technique in this dissertation is a novel technique that identifies whether covered code is just incidentally covered. In particular, the technique identifies incidentally covered code by whether the code has been covered by its designated test(s). Intuitively, incidentally covered code can give the false feeling of security about how well the application code has been tested. The goal of the research is to quantify the effect of incidentally covered code on test quality and show how to improve the tests by testing incidentally covered code purposely. I developed a definition for incidentally covered code and a technique to identify incidentally covered code.

The dissertation is organized as follows: Chapter 2 introduces the related work of this dissertation. Chapter 3 presents the technique that identifies brittle test oracles and unused test inputs. Chapter 4 presents the two techniques that interpret coverage information to identify insufficiently tested code. Chapter 5 summarizes the dissertation.

Chapter 2

RELATED WORK

Existing work related to my research in the dissertation will be discussed in this section. Since the OraclePolish technique searches for brittle assertions, it is related to the areas including: test oracles, test brittleness. The ICDC and Fostered Code techniques provide new interpretations on coverage information. So they are closely related to the areas including: coverage criteria and defect prediction. Finally, since the three techniques give suggestion about what to test and how to improve the tests, I will discuss automatic test generation techniques.

2.1 Test Oracles

Test oracles play a crucial role in testing. To date, there have only been a few studies of test oracles that help developers gain better understanding in test oracles. In general, these studies have compared the costs and effectivenesses of different oracle-creation strategies—rules that specify which subset of a program's state should be checked by an oracle.

Most recently, Li and Offutt evaluated the performance of 6 new oracle-creation strategies for model-based testing [74]. They found that checking the values of a program's variables is much more effective than simply checking for the presence or absence of exceptions. They also found that checking the value of a variable more than once was only marginally better than checking multiple times and that in some cases, weaker oracle creation strategies are as effective as strong oracle creation strategies.

Yu et al. performed a similar study where they studied 6 oracle creation strategies designed to generate oracles that check for specific types of faults in concurrent

programs [127]. They found that, for concurrent programs, only checking the program's output performs worse than checking at least some of the internal state.

Shrestha and Rutherford compared the performance of oracles that only check for exceptions against oracles that use pre- and post-conditions written in the Java Modeling Language (JML) [102]. They found that pre- and post-condition based oracles are more effective and should be used in preference to oracles that only check for exceptions.

Sprenkle et al. developed and studied a suite of 22 oracle creation strategies designed to check whether a web application produces correct output when given a test input [103]. Overall, they found that the best strategy depends on the characteristics of the web application and the fault that is revealed by the input.

Xie and Memon considered different oracle creation strategies for GUI applications [123]. They compared tradeoffs between what elements of the interfaces should be checked (e.g., individual widgets, single window, multiple windows, etc.) and how often the checks should be performed (e.g., after each input, at the end of the test, etc.). Based on their study, they found that weaker oracle creation strategies detect fewer faults and that a thorough check at the end of a test often provides the best balance between cost and effectiveness.

While these studies have provided interesting results, they all share a common limitation. They are proposing and evaluating new oracle creation strategies. While investigating fabricated strategies is important, it does have several drawbacks. First, it is unclear if the oracle creation strategies that they are evaluating are representative of the strategies actually used by testers. Testers may not use a similar process or even think about the problem in the same way. Second, they are ignoring the human aspects of the oracle creation process. While test generation tools can mechanically follow an oracle creation strategy, developers will often want to make judgement calls based on their intuition, domain specific information, and other external data that should be taken into account.

Rather than studying only newly proposed oracle creation strategies, I believe

that a complementary approach, such as OraclePolish, that investigates and studies the current state of practice is also necessary.

2.2 Test Oracle Improvement

As I mentioned previously in the introduction, there is a large amount of work on improve test oracles by choosing test inputs. As these approaches are complementary to my proposed work, I will only provide a high-level overview of the main areas. *Symbolic execution-based test generation* approaches, first proposed in the 70s (e.g., [27, 57, 66, 94]), solve accumulated constraints on the inputs of a program to explore targeted paths of execution. More recent approaches improve on classic symbolic execution by combining both symbolic and concrete execution (e.g., [15, 45, 100]). This insight, combined with techniques that can address the path explosion problem (e.g., [10, 16, 44, 46, 77]), have led to several practical tools (e.g., [5, 14, 17, 78, 108, 124]). *Model-based test data generation* approaches derive test suites from a model of the system under test rather than the system itself. Specific sub-approaches include: axiomatic approaches (e.g., [13, 26, 37, 43, 69, 72]), finite-state machines (e.g., [56, 61, 71, 89]), and labeled transition systems (e.g., [18, 47, 48, 110, 111]). *Combinatorial-based test data generation* approaches focus on sampling the input parameters to cover a desired subset of the elements to be tested (e.g., [30, 40, 73, 87, 113, 125]). *Adaptive random test data generation* approaches augment random generation approaches under the observation that failure-causing inputs tend to form contiguous failure regions (e.g., [21–23, 25, 75, 76, 101, 107]). Finally, *search-based test data generation* approaches uses optimization algorithms to achieve their goals (e.g., [3, 12, 31, 32, 36, 51–53, 80, 86, 109, 117, 119, 120, 126, 128]).

In addition to techniques that attempt to improve the quality of existing oracles, there are also several techniques that attempt to automatically create oracles. Some of these techniques use mutation testing to discover how successful an oracle is at detecting mutants. For example, Staats et al. use mutation testing to support the creation of oracles by identifying the program variables that are most successful at detecting

mutants and therefore should be checked by an assertion [104, 105]. Conversely, Fraser and Zeller use mutation testing to generate complete test cases, including oracles. Other techniques create oracles from observed invariants (e.g., [91, 92, 97, 106]) or generate oracles for specific domains (e.g., web pages [24, 81]).

2.3 Test Brittleness

State pollution[49] by Gyori et al. finds tests that modify some location on the heap shared across tests or on the file system. Tests would be brittle if those locations are not properly reset. The technique captures and compares heap-graphs and file-system states during test execution. While it focuses more on *writing* on the shared states, my research focuses on *reading* from them.

The brittle assertion problem is also closely related to the test dependency problem. A recent study [131] proposed an approach for detecting test dependencies in existing test suites. The authors described a *k-bounded dependence aware algorithm* which trims the search space for re-ordering the test methods which otherwise requires a full permutation over the test methods. The remaining sequences will be executed and checked to see if this certain ordering will alter the outcomes of some tests in the sequence. However, the search space is still so large that the authors had to limit the length of the sequence up to 2. Moreover, the detection of dependencies are limited to the ones which will unveil their presence by altering the test outcome in a certain order. My research presents a more precise data flow analysis that will further narrow down the search space and also be aware of the dependencies which not only cause changes of outcomes in a certain order but also lie deep in the test suite and application code so that they may fail the tests in the future.

2.4 Coverage Criteria

Coverage refers to the code entities (e.g., methods, branches, statements, etc.) that have been executed (covered) by a test suite. Traditionally, covered entities are always considered as sufficiently tested and the focus of what to test next is on the

attempts to cover the code entities have not yet been covered. Prior to the ICDC and the Fostered Code techniques, some researchers have noticed that covered entities may be insufficiently tested. They developed various techniques based on different concepts.

Schuler and Zeller propose checked coverage as an approach for assessing oracle quality [99]. The checked coverage of a test or test suite is the ratio of executed statements that compute values that are checked by the test to the total number of executed statements. A low checked coverage score suggests that a test is likely to be missing assertions. Unlike my technique, which uses dynamic tainting, checked coverage uses backward dynamic slicing to compute the set of statements that contribute to values checked by the test. While dynamic tainting and dynamic slicing are similar, dynamic tainting, due to its focus on values rather than statements, provides several benefits. For example, my technique precisely identifies unused inputs while checked coverage only identifies sets of statements. Fixing the identified issues starting from sets of statements rather than inputs increases the amount of manual work that testers must perform. In addition, checked coverage shares the common limitation of all coverage-based techniques: deciding how much coverage is sufficient. Obviously, a checked coverage score of 0 % is bad, but what about a score of 60 %?

State coverage, originally proposed by Koster and Kao [67, 68] and extended by Vanoverberghe et al. [114], is similar to checked coverage. The primary difference is that state coverage is the ratio of executed output defining statements (ODS)—statements that define a variable that can be checked by the test suite—to the total number of ODSs. Unfortunately, there have only been small case studies on the technique's effectiveness so it is not clear how it compares to checked coverage. However, because state coverage is also a coverage metric, it shares the same limitations as state coverage as compared to my technique.

Relative coverage, proposed by Bartolini et al., gives another way of interpreting the adequacy criteria [8]. It suggests, that instead of measuring the covered entities throughout the test suite, it is more appropriate to count the entities covered for a certain purpose. The target entities will be specified, and the coverage is the ratio of

covered targeted entities among the targeted entities, for example, the measurement for newly added functionalities in regression testing [84]. In this case, the targeted entities are the newly added operations.

2.5 Defect Prediction

This group of related work is defect prediction approaches that attempt to model various software features in order to predict where defects are located before the defects lead to failures. A wide range of prediction models based on various features have been proposed including size and complexity models (e.g., [2]), development quality data models (e.g., [38, 54, 62]), history defect models (e.g., [55, 65]), multi-linear regression models based on multiple metrics (e.g., [64, 85]). The underlying mechanism how ICDC and Fostered Code can identify insufficiently tested code can be explained by such models. The defect prediction approaches that are most closely related to my approach are the ones proposed by Miller et al. and Voas and Miller.

Miller et al. proposed an approach to estimate the probability of a fault even when testing reveals no failures [83]. The authors provided a probability density function to predict the true probability of failure. With this function, Voas and Miller proposed *testability* which can be measured statically even before testing has taken place to predict the presence or the absence of defects [116]. The authors stated that different software systems have different sensitivities towards testing due to their structures that can be modeled by two kinds of information loss: implicit information loss and explicit information loss. Implicit information loss occurs when two or more different inputs to a function produce the same output. The authors defined the domain/ range ratio (DRR), as the ratio between the cardinality of the domain of the specification and the cardinality of the range of the specification. For example, while a function that checks whether an integer is odd can accept any integer as input, it only has two possible outputs so its DRR is $\infty : 2$. Conversely, a function that negates a boolean value has a DRR of $2 : 2$. The assumption is that functions with higher DRR values are more likely to contain undetected failures.

2.6 Automatic Test Generation

As my research can give suggestions for what to test next, automated test generation approaches can be used to actually generate the missing test cases. The family contains different techniques focusing on different facets of automatic test generation, including (1) generating test inputs effectively and efficiently, (e.g., [115]), (2) generating test oracles effectively and efficiently, (e.g., [1]), (3) using mutation to expose the vulnerability of the system, (e.g., [33, 42]), and (4) generating tests directed by coverage information, (e.g., [45, 96, 122]). The techniques that are guided by coverage information may benefit most from ICDC and Fostered Code. Although I do not provide an automatic approach about how to generate new tests, e.g., what the test inputs and expected values will be, the information about direct coverage can be used to guide the generation process.

Chapter 3

IMPROVING ORACLE QUALITY BY DETECTING BRITTLE ASSERTIONS AND UNUSED INPUTS

Although software tests are conceptually simple—they are composed of two parts: *inputs* that are used to execute the program under test and an *oracle* that is used to verify that the execution induced by the inputs produces the expected results—tests are often difficult to write in practice. This is especially true for modern software which is typically large and complex. Together, these characteristics produce a situation where test writers have an imperfect understanding of not only what inputs a program may receive but also how the program should behave and what outputs it should produce. In short, when writing tests, selecting neither inputs nor oracles is straightforward.

In many testing frameworks, an oracle is encoded as a set of assertions that check whether a subset of a program's state (variables) has particular values. Considered in this way, choosing an oracle is analogous to choosing a point on the continuum from checking nothing to checking the entire state of the program. While neither extreme is appropriate—oracles that check nothing will never find bugs, and oracles that check everything will likely be difficult to maintain and enormous—there is a point somewhere between that represents the ideal oracle. Unfortunately, identifying this point is challenging. In practice, the oracles written by testers often miss the mark by either: (1) checking too little by failing to include assertions for relevant variables—which can result in tests that are unable to reveal failures (i.e., missed warnings), or (2) checking too much by including assertions about irrelevant variables—which can lead to brittle tests that fail when they should not (i.e., false warnings, see Section 3.1 for an example).

Existing work on assessing the quality of test oracles addresses only the first of these possibilites by detecting tests that are likely missing assertions (e.g., [99, 105, 114]).

In this part of my research, I present a novel dynamic analysis technique that addresses both possibilites. The technique, named OraclePolish, is based on dynamic tainting and works by tracking the flow of controlled and uncontrolled inputs along data- and control-dependencies at runtime. Intuitively, controlled inputs are inputs explicitly provided by the test itself (e.g., constants that appear in the test method) and all other inputs are considered uncontrolled. When a test finishes execution, the technique uses the tracked information to generate reports that identify *brittle assertions*—assertions that check values that are derived from inputs that are not controlled by the test and *unused inputs*—inputs that are controlled by the test but are not checked by an assertion. These reports are then filtered to remove false positives and presented to testers.

To evaluate the technique, I created a prototype implementation that analyzes Java applications and tests written using the JUnit testing framework.[1] I used the prototype tool to analyze over 4,000 tests from real, open source software projects and to answer several research questions about (1) the feasibility and effectiveness of the technique, and (2) the quality of existing test oracles.

This work[59] makes the following contributions:

- The definition of a new technique that can automatically analyze tests to detect both brittle assertions and unused inputs.
- A prototype implementation of the technique that implements the technique for Java applications with test cases written using JUnit.
- An extensive empirical study that demonstrates my tool's feasibility, accuracy, and usefulness.

```
1 public class EmployeeTest extends TestCase {
2  static String firstName, lastName, ssn;
3  static double baseSalary, commissionRate = 0.5, grossSales;
4  static Employee E = new Employee(firstName, lastName, ssn,
5                            baseSalary, commissionRate,
6                            grossSales);
7
8  public void testToString() {
9    E.setFirstName("John")
10   E.setGrossSales(200);
11   E.setBaseSalary(100);
12   String expected = "Employee: null\n" +
13                     "social security number: null\n" +
14                     "total salary: 200.00";
15   assertEquals(E.toString(), expected);
16 }
17
18 public void testAbbreviateLastName() {
19   E.setLastName("Moore-Towers");
20   String expected = "Moore";
21   assertEquals(E.abbreviatedLastName(), expected);
22 }
23 }
```

Figure 3.1: An example of a brittle test case.

3.1 Motivating Example

In this section, I provide an example that will be used in the remainder of the section to illustrate my technique. Figure 3.1 shows the example, which consists of several tests derived from the test suite for CommissionEmployee, a small application that is used to perform various computations necessary to calculate payroll information for sales employees. Consider `testToString`, which is checking whether `Employee`'s `toString` method produces the expected output. While this test fulfills its goal, it has several problems.

First, `testToString` is brittle because it makes assertions about values derived from inputs that it does not control. More specifically, the test assumes that the values of the employee's last name, social security number, and commission rate are not changed between the time when the employee is created and the time when the result of `toString` is checked. Note that no such assumption is made about the employee's first name, gross sales, or base salary as these values are controlled by the test (i.e.,

[1] http://www.junit.org

they are explicitly set to "John", 200 and 100, respectively, during the execution of the test).

In practice, there are many ways that `testToString`'s assumption that the employee's last name, social security number and commission rate are not changed could be violated. For example, if `testAbbreviateLastName` was added to the test suite, `testToString` would fail intermittently, depending on the order in which the test cases are executed. If `testToString` is executed first, the assumption holds and both tests will pass. However, if `testAbbreviateLastName` is executed first, the assumption is violated and `testToString` will fail because the value of the employee's last name will no longer be null. To prevent the possibility of failures due to brittleness, a test should not check values derived from inputs that it does not control. For `testToString`, this can be accomplished by explicitly controlling the values of the employee's last name, social security number, and commission rate, as is done for first name, gross sales, and base salary. This can also be accomplished by creating a new instance of `Employee` with known values or by explicitly setting the employee's last name, social security number, and commission rate. Both of these options fix the test's brittleness by ensuring that the values checked by the its oracle are derived from controlled inputs.

The second problem with `testToString` is that one of the test's controlled inputs is unused. Although the test specifically sets the employee's first name to "John", there is not any assertions checking any values that are derived from this input. Unused inputs suggest that the test's author is unsure about the behavior of the application under test, possibly because modifications made to the application have not been reflected in the test or simply because the tester was unfamiliar with the code when the test was written. In the worst case, an unused input indicates that a test is missing an assertion which could lead to missed warnings—situations where the test should fail but does not. Even if they do not lead to missed warnings, unused inputs increase the costs of test maintenance by increasing the cognitive burden on the tester. To eliminate unused inputs, additional assertions could be added to the test (e.g., adding `assertEquals(E.getFirstName(), "John")` to `testToString`) or the

unused inputs could be removed (e.g., deleting Line 9 in `testToString`).

The overall goal of the proposed technique, OraclePolish, is to reduce the costs of testing by automatically identifying, and helping testers fix both brittle assertions and unused inputs. The basic intuition behind the approach is that dynamic tainting, due to its ability to mark and track inputs at runtime, can be successfully used to accomplish this goal.

3.2 Background

In this section, I provide background information on dynamic tainting. Note that the material in this section is paraphrased from previous work on dynamic tainting [28]. Intuitively, dynamic tainting consists of (1) marking some data values in a program with a piece of metadata called a taint mark, and (2) propagating taint marks according to how data flows in the program at runtime. In this way, dynamic tainting can track and check the flow of information through a program while it executes.

Information flows through a program in two ways: through data dependences and through control dependences. I illustrate these two kinds of flows using the code examples in Figure 3.2. First, consider the code in Figure 3.2a. Assume that variable a is tainted with taint mark t_a at Line 2 and variable b is tainted with taint mark t_b at Line 3. Given this assignment of taint marks, variables x, y, and z would be tainted, at the end of the execution, with sets of taint marks $\{t_a\}$, $\{t_b\}$, and $\{t_a, t_b\}$, respectively. Taint mark t_a would be associated with x because the value of a is used to calculate the value of x (`x = a + 2`), that is, x is data dependent on a. Analogously, y would be tainted with t_b because the value of b is used to calculate the value of y (`y = b * 4`). Finally, z would be tainted with both t_a and t_b because the values of both x and y are used to compute the value of z (`z = x + y`), that is, z is indirectly data dependent on both a and b.

Consider now the code in Figure 3.2b and assume that variable a is tainted with taint mark t_a at Line 2. Although a's value is not directly involved in the computation of x in this case, it nevertheless affects x's value: the outcome of the predicate at Line 3

```
1 int x, y, z;
2 int a = input();
3 int b = input();

4 x = a - 2;
5 y = b * 4;
6 z = x + y;
```

(a)

```
1 int x, y;
2 int a = input();

3 if(a > 0)
4    x = 0;
5 else
6    x = 1;
7 y = 2;
```

(b)

Figure 3.2: Code examples that illustrate information flow through data and control dependencies.

decides whether Line 4 or Line 6 will be executed, that is, the statements at Lines 4 and 6 are control dependent on the statement at Line 3. Therefore, the value of x at the end of the execution would be associated with taint mark t_a.

In general, the propagation of taint marks along data dependences is called *data-flow propagation*, and the propagation along control dependences is called *control-flow propagation*. The details can be found in Section 3.5.

3.3 Detecting Brittle Assertions and Unused Inputs

This section presents the technique, OraclePolish, for helping testers improve the quality of their test oracles. I first provide an intuitive description of the technique and then discuss its main characteristics in the following 4 steps.

3.3.1 Overview

The basic technique behind OraclePolish is dynamic tainting which can mark and track inputs at runtime. In this spirit, the OraclePolish technique works by (1) assigning taint marks to two types of inputs: inputs that are controlled by the test and inputs that are not controlled by the test, (2) tracking both types of inputs by suitably propagating the taint marks as a test executes, and (3) identifying, when an assertion is executed, which taint marks are associated with the values checked by the assertion. The taint marks discovered in the third step allow for identifying situations where a test checks too much (i.e., is brittle) and situations where a test checks too little (i.e., has unused inputs).

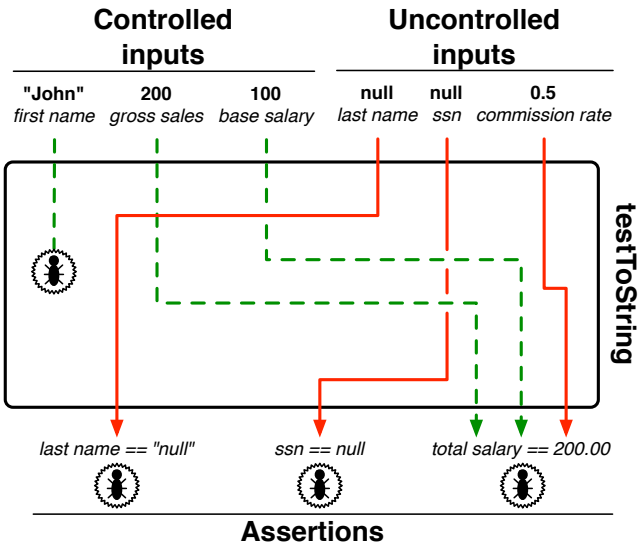


Figure 3.3: Intuitive view of the application of my technique to `testToString` from Figure 3.1.

Before presenting the details of the approach, I discuss how it works on `testToString` from Figure 3.1. This test is brittle because it contains assertions about values derived from uncontrolled inputs, and also has unused inputs because some controlled inputs, or values derived from them, are not checked by an assertion. Figure 3.3 provides an intuitive view of how OraclePolish can detect both problems.

The top of Figure 3.1 shows `testToString`'s inputs. Note that the inputs are divided into two categories: controlled inputs and uncontrolled inputs. Intuitively, controlled inputs are values that are provided as part of the test itself (in this case, "John", 200 and 100) and uncontrolled inputs are inputs that are not explicitly set during the execution of the test (in this case null, used to initialize `lastName` and `ssn`, and 0.5, used to initialize `commissionRate`). Section 3.4 provides a detailed discussion of how the technique identifies controlled and uncontrolled inputs. To make the example more clear, each input shows both the value (top) and a brief description of what the value represents (bottom). For example, the value "John" represents the first name of the employee, the value 200 represents the employee's gross sales value, etc.

The bottom of Figure 3.1 shows, conceptually, the test's oracle. The call to `assertEquals` at Line 15 implicitly checks whether the employee's first name is equal to "John", whether the employee's social security number is equal to null, and whether the employee's total salary is equal to 200.00.

The lines that traverse `testToString` illustrate, intuitively, how OraclePolish assigns a unique taint mark to each input. In the figure, each input is the source of a unique line. In addition, the color and style of the lines indicate the type of input: green, dashed lines indicate taint marks assigned to controlled inputs and solid, red lines indicate taint marks assigned to uncontrolled inputs. In the remainder of the section, I refer to taint marks assigned to controlled inputs as c-marks and taint marks assigned to uncontrolled inputs as u-marks. The lines in the figure also illustrate how my technique tracks inputs at runtime, by propagating the taint marks as the test executes. For example, the line connecting the input "null" to the assertion `last name == null` indicates that the value checked by the assertion (the employee's last name) is derived from the value null. Similarly, the lines that connect the inputs 200, 100, and 0.5 to the assertion `total salary == 200.00` indicate that the value checked by the assertion (the employee's total salary), is derived from the values 200, 100, and 0.5. The technique uses this information to detect brittle assertions and unused inputs. In Figure 3.3, both types of deficiencies are shown using a bug icon.

Brittle assertions are detected by (1) identifying, when an assertion occurs, the set of taint marks associated with the values checked by the assertion, and (2) checking whether the set of taint marks contains a u-mark. If the set does contain a u mark, the assertion is considered to be brittle. For example, as Figure 3.3 shows, there are three taint marks associated with the value checked by the assertion `total salary == 200`: the taint marks for 200, 100, and 0.5. Because the taint mark for 0.5 is a u-mark, the technique identifies this assertion as brittle. Similarly, the other two assertions are also identified as brittle because the values that they check are tainted with a u-mark.

Unused inputs are detected by (1) computing the union of all taint marks associated with every value checked by an assertion, and (2) checking whether the union

contains every c-mark assigned during the execution of the test. If a c-mark is not present in the union, its corresponding input is unused. For example, in Figure 3.3 the union of all taint marks associated with values checked by the assertions does not contain the c-marks for "John". Consequently, "John" is identified as an unused input.

In addition to detecting brittle assertions and unused inputs, the information provided by propagating taint marks is used to give testers additional data about the identified errors. Intuitively, the technique tracks backwards to identify the origins of the problems and outputs the source of each input (i.e., the locations where the controlled and uncontrolled inputs were assigned a taint mark). These locations can serve as a starting point to help testers fix the identified problems with their assertions.

3.4 Input Tainting

Input tainting is responsible for associating taint marks with a test's inputs. The technique intercepts the execution of the test at specific points and assigns either a c-mark, for controlled inputs, or a u-mark, for uncontrolled inputs.

3.4.1 Tainting Controlled Inputs

Currently, OraclePolish considers two types of values to be controlled inputs. First, the technique considers values (constants) that are (1) used during the execution of the test method itself, and (2) not directly passed to an assertion method to be controlled inputs. For example, in Figure 3.1, the constant "John" used at Line 9 is a controlled input, but the constant "Moore" at Line 20 is not because it is used as an argument to `assertEquals` on Line 21. The decision to only include constants in the test method itself is based on my experience, domain knowledge, and intuition about how testers write tests. Initially, the technique also considered constant values from the test's setup code to be controlled inputs. However, I found that many tests use the same setup code to construct a complex state. While, when considered individually, the tests appear to have unused inputs (i.e., parts of the common state that are not checked), when they are considered collectively, they check the entire state.

Second, the technique considers the return values of no-argument methods called in the test method itself and also implemented in the test class to be controlled inputs. I found that no-argument methods are frequently used to separate long sequences of initialization code from a test. Moving such initialization code to a separate method decreases the size of the test, which can improve its readability and understandability. Because such methods are conceptually part of the test, I consider their results to be controlled inputs.

To taint constants that are used as part of the test method, OraclePolish simply intercepts the loading of the constants and applies a unique c-mark to the loaded constant. Similarly, to taint the return value of no-argument methods implemented in test suite, the technique intercepts the test's execution immediately after the method returns and applies a unique c-mark to the return value. Note that if either type of controlled input is used repeatedly, as would be the case inside of a loop, the technique reuses the same c-mark for each iteration. Based on the results of the experiments (see Section 3.8), I found that this approach produces the most understandable reports. From the point of view of a tester, regardless of the number of times an input is used, it is still, conceptually, the same input.

As a concrete example of how my technique assigns c-marks to controlled inputs, consider `testToString` in Figure 3.1. The technique identifies three controlled inputs in this test: the literal value "John" used at Line 9, the literal value 200 used at Line 10, and the literal value 100 used at Line 11. When each of these constants is loaded, the technique assigns a unique c-mark to each value (e.g., c_1 is assigned to "John", c_2 is assigned to 200, and c_3 is assigned to 100).

3.4.2 Tainting Uncontrolled Inputs

Currently, OraclePolish considers one type of values to be uncontrolled inputs. The values of global variables (static, mutable fields) are considered to be uncontrolled inputs. The intuition behind this choice is that reading the value of a global variable is the most likely way for a test to unintentionally depend on a value that it does not

control. Because global variables maintain their state and can be written to at any time, a test has no way of knowing what values they contain. Similarly, tests can also unintentionally depend on the contents of the files, databases, network connections, etc. I chose not to consider values read from such sources as uncontrolled inputs because unit tests typically use mock objects instead of the real resources.

To assign u-marks to global variables, when each class, including each test class, is loaded, the technique assigns a unique u-mark to each non-final, static field. If the field is a reference value, OraclePolish will recursively assign taint marks to the fields of this reference value until the technique finds a primitive or an already solved case (e.g., an array). As a concrete example of how my technique assigns u-marks to uncontrolled inputs, consider `testToString` in Figure 3.1. `EmployeeTest` has six static fields, only one of which has a variable initializer (`commissionRate`). When an instance of `EmployeeTest` is initialized, all six fields are assigned a unique u-mark (e.g., `firstName` is assigned u_1, `lastName` is assigned u_2, `ssn` is assigned u_3, `baseSalary` is assigned u_4, `commissionRate` is assigned u_5, and `grossSales` is assigned u_6). During the class loading of `EmployeeTest`, when `commissionRate` is assigned the value 0.5, the taint mark assigned during object initialization is overwritten with a fresh taint mark (e.g., u_5 is overwritten by u_7). Later when `testToString` is executed, OraclePolish would assign a unique u-mark to each static, mutable field of every currently loaded class.

3.4.3 Recording Supplemental Information

Regardless of the type of taint mark, OraclePolish performs an additional action when assigning a taint mark t to an input i. To help testers debug the problems identified by OraclePolish, additional information is recorded. More specifically, OraclePolish logs a tuple $\langle t, loc, value \rangle$, where loc is the location in the execution where the taint mark was associated with the input and $value$ is the initial value of the input. The location is expressed differently depending on the type of taint mark. For c-marks, the location is the file name and line number corresponding to where the constant or

return value was used. For u-marks, the location is the fully qualified name (field name and declaring class name) of the field that contains the input. This information is used by OraclePolish when generating reports, as described in Section 3.6.

3.5 Taint Mark Propagation

A taint propagation policy specifies how taint marks are propagated during execution. Typically it is defined along two dimensions: how to combine taint marks and which types of dependences to consider.

My technique's policy for combining taint marks is fairly intuitive. In general, OraclePolish taints all values written by a statement with the union of all taint marks associated with the values read by that statement. For instance, after the execution of statement `x = y + z`, where `y` and `z` are tainted with taint marks t_1 and t_2, respectively, `x` would be associated with the set of taint marks $\{t_1, t_2\}$. The only type of statement where OraclePolish deviates from this general policy is the execution of native methods. Because native methods are executed by the Java Virtual Machine (JVM), it is often unclear which values are read by the native method. Rather than require a precise model of every native method, the technique conservatively assigns the union of all taint marks associated with the native method's arguments to its return value.

When choosing which dependences to consider, OraclePolish considers both data-flow and control-flow dependencies. Identifying data-flow dependencies is trivial as they are encoded as the semantics of the language. Identifying control-flow dependencies is more challenging.

A control-flow dependence arises when a conditional branch b decides whether a statement s is executed. In this case, the values that affect b's outcome indirectly affect the values of any data written by s. Therefore, to be conservative, the taint marks associated with the values read by b must be combined and associated with the values written by s. To achieve this, the technique uses an approach that in Clause et al.'s prior work [29]. In brief, the technique keeps track of relevant taint marks at runtime by leveraging statically-computed post-dominance information. When an execution

reaches a conditional branch b, the technique (1) computes T, the union of the taint marks associated with the values read by b, and (2) adds a pair $\langle b, T \rangle$ to CF, the set of active control flow marks. When execution reaches the immediate post-dominator of a conditional branch b, it removes from CF all pairs $\langle x, y \rangle$ such that x is equal to b. Note that CF will contain multiple pairs with x equal to b when b is executed as part of a loop. Each iteration will add a new pair $\langle b, T \rangle$ to CF, all of which must be removed when the immediate post-dominator is executed. When a statement s is executed and CF is not empty, the technique computes the union of all active control flow marks (i.e., the union of y, for each pair $\langle x, y \rangle$ in CF) and adds this set to the set of taint marks associated with the values written by s.

3.6 Checking Taint Marks

This third part of my technique, checking, is responsible for two tasks: (1) identifying brittle assertions and unused inputs, and (2) generating the error reports that will be presented to testers.

To identify brittle assertions, the technique intercepts the execution of comparison operations (e.g., greater than, less than, equals, etc.) that occur inside the execution of an assertion method (e.g., `assertEquals`). Intercepting the execution of comparison operations, rather than simply examining the actual argument of the assertion method, allows for a more precise identification of brittle assertions. For example if the actual parameter of the assertion method is an object, examining the taint marks associated with all of the object's fields is likely to be incorrect as not all of the fields are necessarily involved in checking whether the actual and expected arguments are equal. Rather than attempting to identify, a priori, which values are used to check for equality, the technique can simply monitor the comparison operations to achieve the same effect. In addition, testers often inadvertently swap the order of the actual and expected arguments which means that the actual parameter may not be in the correct location which would result in incorrect reports.

```
testToString appears to be brittle.  The assertions at the
following lines check values that are derived from
uncontrolled inputs:

assertEquals at Line 25 depends on:
EmployeeTest.lastName being null (object initialization)
EmployeeTest.ssn being null (object initialization)
EmployeeTest.commissionRate being 0.5 (object initialization)
```

(a) Report for brittle assertions.

```
testToString appears to be missing one more assertions. The
following values are provided as input but are not checked by
an assertion:

"John" (EmployeeTest.java, Line 9)
```

(b) Report for unused inputs.

Figure 3.4: Example reports output by my technique when run on `testToString` from Figure 3.1.

For each comparison operation inside of an assertion method, the technique identifies the taint marks associated with the values involved in the comparison and checks whether the set of identified taint marks contains a u-mark. If a u-mark is found, the technique detects a brittle assertion.

To identify unused inputs, the technique calculates the union of all c-marks that were encountered when checking for brittle assertions. The technique then subtracts this set from the set of all c-marks that were assigned to controlled inputs. If the resulting set is not empty, the inputs initially assigned with the remaining c-marks are marked as unused.

As a concrete example of how the checking part of the technique operates, consider again `testToString` from Figure 3.1. For this test, the technique would intercept the comparison operations that occur inside the call to `assertEquals`. Because the actual value is a string, the `String` class's `equals` method is used to perform the check. The `equals` method uses a series of equality comparisons (i.e., `==`) to check whether the same characters make up each string. Each time this equality is executed, the technique determines whether either character is tainted with a u-mark. Because the actual value is derived from three uncontrolled inputs, three u-marks are found and a brittle assertion is detected.

At the end of the test's execution, the technique calculates the union of all encountered c-marks. Because only values tainted with only two of three total c-marks were checked by the assertion, an unused input report for the remaining mark (the one initially assigned to "John") is created.

Figure 3.4 shows the error reports generated by the technique when run on `testToString`. As the figure shows, both the brittleness report (top) and the unused input report (bottom) include all of the information necessary to help testers fix the identified issues. The brittleness report includes the name and location of the brittle assertion (assertEquals on Line 25), the uncontrolled values that were used to compute the values checked by the assertion (null, null, and 0.5), the names of the fields where the uncontrolled values were stored (EmployeeTest.firstName, EmployeeTest.ssn, and EmployeeTest.commissionRate), and the locations where the uncontrolled values were stored into the fields (during object initialization). Note that to collect the location information, the technique traverses the test's call stack to find the name of the outermost enclosing assertion method invocation and the location where the assertion method was invoked. The unused input report includes the controlled inputs that were not checked by an assertion and the line number where the value was loaded.

3.7 Removing False Positives

The purpose of the fourth part of the technique is to filter false positive error reports. Taint mark propagation is known to be imprecise, especially in the case of native methods. As a result, error reports generated by the third part of the technique may be false positives. More specifically, the technique may generate false positive reports if it under-propagates c-marks or over-propagates u-marks. As a result, controlled inputs may appear to be unused when in fact they are used and uncontrolled inputs may appear to be checked by an assertion when in fact they are not.

To eliminate such false positives, the technique uses an approach inspired by mutation testing [35, 50]. Essentially, the technique preemptively makes changes that may happen in the future and checks to see whether such changes alter the outcome

of the test. More specifically, for each input that is identified as the cause of a brittle assertion or as an unused input, the technique re-executes the test. As the test is being re-executed, at the point when the taint mark was assigned to the input in the original execution, the technique mutates the value of the input to a randomly chosen value of the same type. Note that data- and control- flow analysis is not needed to accomplish this. The technique then compares the outcomes of the re-executed and original executions.

In the case of uncontrolled inputs that cause brittle assertions, we would expect that changing the input's value would alter the outcome of the test. If a test checks values derived from an input, changing the value of the input should change the outcome of the test. If the outcome of the test does change, the report is a true positive (i.e., if the change were to be made, the test would fail) and is presented to the user. Conversely, if the outcome of the test does not change, the report is a false positive (i.e., the u-mark should have been over written but was not) and is discarded.

In the case of unused inputs, I would expect that changing the value of the input would not alter the outcome of the test. If an input is really unused, its value doesn't matter. If the outcome of the test does not change, the unused input report is a true positive and is presented to the user. Conversely, if the outcome of the test does change, the error report is a false positive (i.e., the c-mark should have propagated to an assertion but did not) and is discarded.

Note that this filtering strategy is precise—reports that are identified as true positives have an associated witness (a concrete change that will cause the test to fail)—but not safe—reports that are identified as false positives may actually be true positives. True positives can be identified as false positives when the randomly chosen value is indistinguishable from the original value. For example, consider an assertion that checks whether a value is positive. If the original value checked by the assertion is 1 and the randomly chosen replacement is 5, the outcome of the assertion will be the same for both values. To reduce the possibility of this occurring, multiple re-executions, each with a unique value, can be run or additional analysis could be performed to identify

values that are more likely to cause the outcome of the test to change.

In the case of `testToString`, both of the error reports generated by the third part of the technique are true positives. Changing the value of `commissionRate` causes the test to fail and changing the value of the employee's first name does not cause the test to fail.

3.8 Evaluation

To evaluate my technique, I created a prototype implementation called OraclePolish and analyzed over 4,000 tests for 12 real Java applications. Using the output of the tool, I investigated the following research questions:

RQ 1—Effectiveness. Can the technique detect both brittle assertions and unused inputs in real test suites?

RQ 2—Cost. What are the costs associated with using the technique and are they reasonable?

Note that RQ1 provides a quantitative assessment of the technique; it does not make any assumptions about whether the reported errors are likely to cause problems in the future. Conversely, RQ2 is a qualitative assessment that does take into account the users' perspective.

The remainder of this section describes (1) OraclePolish, the prototype implementation of my technique, (2) the experimental subjects I chose, (3) the experimental protocol I used and the data I generated, (4) the results of evaluation, and (5) threats to the validity of my results.

The prototype implementation of my technique, as well as the subjects I chose and the experimental data I generated, are available from: http://bitbucket.org/udse.

Table 3.1: Experimental subjects and data.

Subject	LoC	Test Suite		Brittle Assertions		Unused Inputs	
		# Tests	# Executable	# Reports	# TP	# Reports	# TP
CommissionEmployee	100	15	15	2	1	13	13
DataStructures	429	106	99	0	0	55	47
Employee	183	15	15	3	3	11	11
LoopFinder	49	13	13	0	0	13	5
Point	69	13	11	0	0	10	8
ReductionAndPriority	3,245	52	38	0	0	37	37
Sudoku	376	25	18	0	0	7	0
commons-beanutils-1.8.3	11,375	810	73	12	4	59	44
commons-cli-1.2	1,978	164	130	22	5	119	24
commons-collections-3.2.1	26,414	886	672	20	12	426	247
commons-io-2.4	26,614	824	236	1	0	154	78
commons-lang-3.1	23,070	2,024	1,547	128	114	1,133	549
commons-math-3.0	70,006	1,150	72	0	0	39	12
JDepend-2.9.1	2,531	39	0	0	0	0	0
Jfreechart-1.0.15	92,252	2,234	862	1	0	538	285
joda-convert-1.2	2,675	105	0	0	0	0	0
Joda-time-2.2	86,797	3,962	506	181	25	226	144
Jtopas-0.8	4,373	53	27	0	0	21	11
PMD-5.0.4	100,300	770	346	2	0	184	93
total		13,609	4,718	405	164	3,060	1,618

3.8.1 Prototype Tool

OraclePolish is a prototype implementation of my technique for applications written in the Java language using the JUnit testing framework. It consists of three separate components: the *analyzer*, the *runtime system*, and the *mutator*.

The primary task of the analyzer is to statically compute the information needed by the runtime system. More specifically, the analyzer computes the post-dominance information needed to perform control-flow propagation. The current implementation of the analyzer uses the T.J. Watson Libraries for Analysis (WALA) to perform the necessary analyses. I choose WALA because it (1) analyzes Java bytecode, which means that I do not need to obtain the source code for all parts of the application, (2) provides built-in dominator analyses, and (3) is extensible enough to allow us to easily implement the other necessary analyses.

The runtime system implements the input tainting, taint mark propagation, and checking parts of the technique described in Section 3.4, Section 3.5, Section 3.6,

respectively. The current implementation of the runtime system is an extension to Java PathFinder (JPF), an explicit state software model checker for Java software.[2] To assign taint marks to inputs, OraclePolish uses JPF's listener callbacks to intercept class and object initialization and to intercept the execution of instructions that load constants. To implement taint mark propagation, OraclePolish uses JPF's bytecode overloading facilities to replace each Java bytecode with a modified version that replicates the instruction's original semantics while also propagating taint marks. Finally, to implement checking, OraclePolish again uses JPF's listener callbacks to intercept the execution of comparisons instructions that occur inside of assertion methods.

The mutator implements the part of the technique that filters false positives (see Section 3.7). It is also implemented as a plugin to JPF. Similarly to the runtime system, the mutator uses JPF's listener callbacks to intercept class and object initialization and to intercept the execution of instructions that load constants. However, instead of assigning taint marks, the mutator randomly changes the values of the inputs. Currently, the mutator re-executes the test three times. As I demonstrate in Section 3.8.4, this number is sufficient to eliminate many false positives.

3.8.2 Subjects

The goal of the technique is to improve oracle quality by detecting brittle assertions and unused inputs. To suitably evaluate the technique with respect to this goal, I selected the test suites of 20 applications as my subjects. Table 3.1 describes the applications. In the table, the first column, *Subject* shows the name and version of each application, if available. The first eight applications (CommissionEmployee through Sudoku) are taken from the Proteja Test Suite Executor and Coverage Monitor repository.[3] The remaining subjects were obtained from various repositories including: (1) the Software-artifact Infrastructure Repository (SIR),[4] which provides a variety

[2] http://javapathfinder.sourceforge.net/

[3] https://code.google.com/p/proteja/

[4] http://sir.unl.edu

of open-source projects for empirical software engineering, (2) SourceForge,[5] a popular repository for open-source projects, and (3) Apache Commons,[6] a collection of reusable components. The second column, *LoC*, shows the number of non-blank, non-comment, lines of code that comprise the application and the third column, *# Tests* shows the number of tests in each application's test suite.

I chose the test suites of these applications as subjects for several reasons. First, the applications cover a variety of subject domains. For example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, Joda-Time is a library for handling dates and times, etc. Second, the applications vary in size. For example, Commons-math has over 70,000 lines of code, while Sudoku only has 376 lines of code. Finally, the test suites also vary in size. The test suites for some of the application contain more than 3,000 tests while others contain fewer than 20. Selecting test suites and applications of various sizes and subject domains improves the generalizability of my results.

After selecting my subjects, I performed an initial sanity check and removed any tests that can not be run using JPF. The number of remaining tests is shown in the fourth column, *# Executable*. For example, although Commons-beanutils-1.8.3's test suite contains 810 tests, 73 of which are executable using JPF. After filtering, I was left with 4,718 tests.

3.8.3 Experimental Protocol and Data

To generate the experimental data necessary for answering my research questions, I ran OraclePolish on each of my 4,718 tests and recorded its output. The experiments were all conducted on the same computer: a machine running Ubuntu 12.04 LTS 64-bit edition with a 3.40 GHz Intel Core i7-2600 processor and 8 GB of memory. Java version 1.7.0_03 was used and was configured with 2 GB of heap space (default).

---

[5] https://sourceforge.net

[6] http://commons.apache.org

Table 3.1 shows the experimental data that I generated. The last four columns in the table show the number of reports generated by the technique, *# Reports*, and the number of reports that are true positives, *# TP*, for both *Brittle Assertions* and *Unused Inputs*. The number of reports is the total number of reports generated by the checking part of the technique (see Section 3.6) and the number of true positives is the number of reports that remain after being filtered by the fourth part of the technique (see Section 3.7).

3.8.4 RQ1: Effectiveness

The purpose of my first research question is to determine the effectiveness of the technique at detecting brittle assertions and unused inputs in real tests. To answer this question, I first judged effectiveness quantitatively, by examining the number of true positive reports generated by OraclePolish.

As Table 3.1 shows, for the subjects I considered, OraclePolish was able to detect both brittle assertions and unused inputs. In total, it detected 164 tests that contain at least one brittle assertion and 1,618 tests that contain unused inputs. These results are encouraging and also a bit surprising. Because most of the tests that I considered are from the test suites of mature applications, I expected them to contain few errors.

It is interesting to note that OraclePolish detects far more unused inputs than brittle assertions. Intuitively, this makes sense as unused inputs are unlikely to cause any observable problems. While missing assertions may cause a test to pass when it should fail, there is no way to detect this occurrence. Similarly, there is not an easy way to measure the amount of additional effort needed to comprehend and maintain tests with unused inputs. As a result, unused inputs are more likely go undetected and unfixed than brittle assertions.

The second way I judged effectiveness was by qualitatively assessing the reports generated by OraclePolish. In the remainder of the section, I provide a more detailed discussion of two randomly chosen brittle tests and two randomly chosen tests with unused inputs.

```
    public class BugsTest extends TestCase {
      public void test13666() throws Exception {
229     Options options = new Options();
230     Option dir = OptionBuilder.withDescription( "dir" )
                                  .hasArg()
                                  .create( 'd' );
233     options.addOption(dir);

236     final PrintStream oldSystemOut = System.out;
237     try {
239       OutputStream bytes = new ByteArrayOutputStream();

247       System.setOut(new PrintStream(bytes));

249          HelpFormatter formatter = new HelpFormatter();
250          formatter.printHelp( "dir", options );

252          assertEquals("usage: dir"+eol+" -d <arg>   dir"
                          + eol,
                          bytes.toString());
        }
        finally {
256       System.setOut(oldSystemOut);
        }
      }
    }
```

Figure 3.5: Brittle assertions in `test13666`

Figure 3.6 shows an excerpt of `test13666` that is part of the test suite for Commons-cli. OraclePolish detects that the assertion at Line 252 is brittle because it depends on several of `OptionBuilder`'s static fields. Because `OptionBuilder` is a singleton, it is possible for other users of the class to leave it in an indeterminate state by starting to build an option but never calling `create`. Internally, create resets the state of the `OptionBuilder` so that it is safe to reuse. To prevent the possibility that `OptionBuilder` has already been partially configured, the test should call `OptionBuilder`'s `reset` method before building an option at Line 230.

Figure 3.6 shows an excerpt of `testPut` that is part of test suite for Commons-beanutils. OraclePolish detects that the assertion at Line 246 is brittle because it checks the value of `stringVal`. As the code shows, `stringVal` is a static field of the `DynaBeanMapDecoratorTestCase`. Because `testPut` does not control the value of `stringVal`, it is assuming that `stringVal` will not be modified between the time when the field is initialized and the time when the assertion is executed. To fix this error, the reference to the static field could be replaced with the expected constant. Alternatively,

```
   public class DynaBeanMapDecoratorTestCase extends TestCase {
43   private static final DynaProperty[] properties =
       new DynaProperty[] { ... };

47   private static String stringVal = "somevalue";

52   private Object[] values = new Object[] {stringVal, ...};

54   private BasicDynaBean dynaBean;

     public void setUp() throws Exception {
96     dynaBean = new BasicDynaBean(dynaClass);
97     for (int i = 0; i < properties.length; i++) {
98         dynaBean.set(properties[i].getName(), values[i]);
99     }

103    modifiableMap = new DynaBeanMapDecorator(dynaBean,
                                                false);
     }

     public void testPut() {
235    String newValue = "ABC";

246    assertEquals(stringVal,
                    modifiableMap.put(stringProp.getName(),
                                      newValue);
247    assertEquals(newValue,
                    dynaBean.get(stringProp.getName()));
248    assertEquals(newValue,
                    modifiableMap.get(stringProp.getName()));
     }
   }
```

Figure 3.6: Brittle assertions in `testPut`

if the field were to be made final it would be guaranteed to have the expected value. Because the majority of `DynaBeanMapDecoratorTestCase`'s other fields are final, this later option is likely to be the correct fix.

Figure 3.7 shows an excerpt of `testGetRowKey` from JFreeChart's test suite. OraclePolish detected two unused inputs in this test: the value "C1" at Line 266 and the value "C1" at Line 267. Although these values are used as arguments to the calls to `addValue` at Line 266 and Line 267, they are not checked by an assertion. Adding additional assertions (i.e., `assertEquals("C1", d.getColumnKey(0))` and `assertEquals("C1", d.getColumnKey(1))`) would ensure that not only are the correct row keys returned, but also that the column keys are not modified.

Figure 3.8 shows an excerpt of `test13` from Employee's test suite. OraclePolish detected three unused inputs in this test: "FN" at Line 160, "SN" at Line 161, and

```
    public class DefaultKeyedValues2DTests extends TestCase {
      public void testGetRowKey() {
257     DefaultKeyedValues2D d = new DefaultKeyedValues2D();

266     d.addValue(new Double(1.0), "R1", "C1");
267     d.addValue(new Double(1.0), "R2", "C1");
268     assertEquals("R1", d.getRowKey(0));
269     assertEquals("R2", d.getRowKey(1));
      }
    }
```

Figure 3.7: Unused inputs in `testGetRowKey`

```
    public class EmployeeTest extends TestCase {
  8   private String fn, ln, ssn;
  9   private double s;

 20   SalariedEmployee s1 = new SalariedEmployee(fn,ln,
                                                 ssn,s);

158   public void test13() {
159     s1.setWeeklySalary(10);
160     fn = "FN";
161     ln = "SN";
162     ssn = "ssn";
163     SalariedEmployee s2 = new SalariedEmployee(fn,ln,
                                                   ssn,s);
164     String actual = s1.toString();
165     String expected = "salaried employee: null null\n"
                        + "ssn: null\n"
                        + "weekly salary: $10.00";
166     assertEquals(actual, expected);
167   }
    }
```

Figure 3.8: Unused inputs in `test13`

"ssn" at Line 162. Although these values are used to construct `s2`, a new instance of `SalariedEmployee`, they are never checked by an assertion. Note that `s1` is used to construct the actual value passed to `assertEquals` at Line 166, not `s2`. In this case, it is not clear how to best fix the test. The unused inputs could be deleted or the actual value could be constructed using `s2` instead of `s1`.

3.8.5 RQ2: Cost

The purpose of my second research question is to investigate the costs of using OraclePolish and to determine if such costs are reasonable. Because my technique is fully automated, the primary cost is its runtime overhead.

To investigate the runtime overhead that OraclePolish imposes, I executed my subject tests twice, once using the JVM and once using OraclePolish (with the preceding static analysis), and compared the execution times of these runs. Based on these measurements, I found that running the tests using OraclePolish takes between 5 and 30 times longer than running the tests using the JVM. Although this cost is significant, I believe that it is reasonable. In my experience, developers will accept high overheads for tools that produce accurate results. This is especially true when, as is the case for OraclePolish, the tools do not require any developer interaction and can be run overnight, possibly as part of an automated build system whose results are inspected the next day. In addition, OraclePolish is an unoptimized prototype. I chose to implement it as a JPF plugin because JPF is a general platform that already implements many of the capabilities I needed (e.g., the ability to associate metadata with runtime values). However, JPF's generality comes at a cost. Based on my experience with taint-based techniques, I believe that a custom implementation of OraclePolish could reduce its overhead to less than 20 %, levels that are comparable to other recent tainting-based approaches (e.g., [11, 90]), by taking advantage of several optimizations (e.g., [19, 98]).

3.8.6 Threats to Validity

There are several threats to the validity of my evaluation. First, I considered a limited number of tests, all of which were written in Java and used the JUnit testing framework. In addition, I filtered out tests that could not be run using JPF. Consequently, my results may not generalize beyond the considered domains. However, the tests that I considered represent a wide range of application domains, sizes, and maturity levels. Therefore, I believe that my results are promising and motivate further research. Second, I qualitatively assessed the usefulness of the error reports generated by the technique, which may introduce bias. While I am planning to conduct a human study with developers to eliminate this threat in the future, I did not believe that such a study was justified at this stage of the research.

3.9 Summary

In this chapter I presented a new technique, OraclePolish, for automatically analyzing test oracles. OraclePolish is based on dynamic tainting and can detect both brittle assertions—assertions that depends on values that are derived from uncontrolled inputs—and unused inputs—inputs provided by the test that are not checked by an assertion. An implementation of the technique has been built to analyze tests that are written in Java and use the JUnit testing framework. Using the implementation, I conducted an empirical evaluation of the OraclePolish's performance on more than 4,000 tests from 12 real applications. The results of the evaluation demonstrate that OraclePolish is able to detect both brittle assertions and unused inputs in real tests at a reasonable cost.

For practical use of the OraclePolish technique, the developers need to determine what inputs shall be considered controlled and what shall be considered uncontrolled in various cases. For example, for JUnit test suites in Java, non-final non-static fields in test classes cannot be used to communicate between tests because a new test class instance is initialized for each test method. For some other testing frameworks, it might not be the case. The cost associated with using the OraclePolish technique mostly depends on the dynamic tainting framework. The prototype implementation is 5 to 30 times slower than running the application on a pure JVM. Since OraclePolish is an automatic tool, the developers can use the tool overnight and get the reports the next day. Upon evolution of the software project, if a test method is added, the additional cost is only associated with the test method for identifying brittle assertions and unused inputs in this test. The additional test will not affect the previous results. If the application has been changed, the whole test suite needs to be executed to determine the changes in brittle assertions. However, in case of changes in the application, the identified unused inputs stay the same.

Chapter 4

IDENTIFYING INSUFFICIENTLY TESTED CODE

Testing provides numerous benefits such as assessing software quality, enabling large scale changes and serving as a form of documentation. Because of these benefits, according to a recent study of practicing software developers [34], many developers have a strong desire for more tests in their projects. However, writing additional tests can be difficult and costly. As reported in the study, a significant portion of this cost is due to the difficulty of identifying which parts of the code to test. To help developers locate where to test, researchers have proposed numerous code coverage criteria (e.g., [9, 20, 58, 70, 79, 88, 93, 95, 112]). Coverage criteria are often used as coverage metrics in the software industry by measuring how much of a certain criterion has been met by a test suite [4]. In this context, coverage metrics offer essential clues about which parts of code to test by indicating which entities (e.g., methods, statements, branches, etc.) in a program are executed (covered) by a certain test suite and which are not. Obviously, uncovered entities are not tested. However, covered entities cannot be assumed to be adequately tested. For example, they may be covered by accident.

The shortcoming is often due to the fact that code coverage metrics do not consider how an entity is covered, only whether it is executed by the test suite. It is common in practice that code is covered but not tested properly. The research community is aware of this problem and has developed techniques to address it. For example, Schuler and Zeller [99] proposed *checked coverage* such that covered entities are considered as checked only when there are assertions associated with them. My intuition for identifying insufficiently tested entities rises from (1) the caller-callee association

during the test execution, and (2) the association between application classes and test classes in various testing paradigms.

In this chapter, I will present two new approaches to identify insufficiently tested code. The first approach is based on the concepts of indirect coverage and direct coverage, ICDC, for interpreting coverage information [60]. At high level, ICDC identifies code entities that have never been directly invoked from a test as being insufficiently tested. The second approach, Fostered Code, is based on the association between the application and the tests in various testing paradigms. At high level, Fostered Code identifies code entities that have not been covered by their designated tests as insufficiently tested. In addition to taking into account how entities are covered by tests, the two approaches eliminate the need for testers to manually identify whether the code indicated by the approaches can be executed. Because the code entities are already covered by the test suite, the entities identified by the approach are guaranteed to be feasible. This means that developers do not have to spend time investigating whether it is possible to execute the identified code.

4.1 Indirect and Direct Coverage

In this section, I will provide the details of the ICDC technique which identifies insufficiently covered code using the concept of direct and indirect covereage. The rest of the section is organized as follows: Section 4.2.1 describes the details of ICDC for interpreting coverage information including formal definitions of direct coverage and indirect coverage, an algorithm to compute direct coverage and indirect coverage, and an example illustrating the approach. Section 4.2.2 presents an empirical study investigating various aspects of direct coverage and indirect coverage on 17 real-world software projects.

4.1.1 Direct and Indirect Coverage

This subsection describes background information necessary for understanding the remainder of the subsection. First, it defines the concepts of direct coverage and

indirect coverage. Second, it describes how the direct coverage and indirect coverage of a test suite can be calculated. Third, it explains how insufficiently tested methods can be identified. Finally, it provides a concrete example of the concepts in terms of statement coverage.

4.1.1.1 Definitions

The concepts of direct coverage and indirect coverage are relatively simple: rather than identifying only *whether an entity (e.g., a branch, a statement, etc.) is covered* by a test suite, direct and indirect coverage identify *how an entity is covered*. Formally, the concepts of coverage, direct coverage, and indirect coverage are defined as follows:

Definition 4.1.1 *An entity e is* covered *by a test suite T iff there exists a test $t \in T$ such that e is executed by t.*

Definition 4.1.2 *An entity e is* directly covered *by a test suite T iff (1) e is covered by T, and (2) there is at least one occurrence that e is covered because a test $t \in T$ directly invokes the method that contains e.*

Definition 4.1.3 *An entity e is* indirectly covered *by a test suite T iff e is (1) covered by T, (2) not directly covered by T, and (3) contained within a method that is publicly accessible.*

Note that in Definition 4.1.3, an entity has to be inside a method that is publicly accessible. If it is not feasible for the test to directly invoke a method, all the entities in such a method are always indirectly covered.

4.1.1.2 Calculating Direct Coverage and Indirect Coverage

Algorithm 1 shows the procedure for computing the direct coverage and indirect coverage of a test suite. As input, the algorithm requires five pieces of information. The first is $\mathcal{E}$, the entities covered in publicly accessible methods. The second is Coverage, a

Algorithm 1 Compute the direct coverage and indirect coverage for a test suite.

Input: AUT is the application under test. T is the test suite of AUT.
Input: $\mathcal{E}$, all covered entities in publicly accessible methods.
Input: Coverage, a mapping that associates each test $t \in T$ to the set of entities covered by t.
Input: Direct, a mapping that associates each test $t \in T$ to the methods that have a depth of 1 in t's dynamic call graph.
Input: MethodOf, a mapping that associates each entity e to the method that contains e.
Output: DC, a mapping that associates each method $m \in AUT$ to the set of entities in m that are directly covered.
Output: IC, a mapping that associates each method $m \in AUT$ to the set of entities in m that are indirectly covered.

```
1: function computeICDC
2:   E' ← E
3:   DC[m] ← {}
4:   IC[m] ← {}

5:   for t ∈ T do
6:     E_t ← Coverage[t]
7:     M_t ← E_t map {e ⇒ MethodOf[e]}
8:     M_d ← M_t filter {m ∈ Direct[t]}

9:     for m ∈ M_d do
10:      E_m ← E_t filter {e ⇒ m = MethodOf[e]}
11:      E ← E \ E_m
12:    end for
13:  end for

14:  IC ← E groupby {e => MethodOf[e]}
15:  DC ← (E' \ E) groupby {e => MethodOf[e]}
16:  return DC, IC
17: end function
```

mapping that associates each test in the test suite to the set of entities that are covered when the test is executed. The third is Direct, a mapping that associates each test in the test suite to the set of methods that are directly invoked by the test (i.e., the set of methods that have a depth of 1 in the test's dynamic call graph). And the fourth is MethodOf, a mapping that associates each entity in the application under test with

the method that contains the entity. The application under test and the associated test suite are provided by the tester while the mappings can be computed using various straightforward static and dynamic analyses.

As output, the algorithm produces two mappings, DC, which associates each method in the application under test to the set of entities contained in the method that are directly covered and, IC, which associates each method in the application under test to the set of entities contained in the method that are indirectly covered. Given DC and IC, it is straightforward to compute the set of entities directly covered or indirectly covered by the entire test suite.

Given the necessary inputs, the algorithm proceeds as follows. First a set $\mathcal{E}'$ memorizes the original $\mathcal{E}$. Then the output mappings, DC and IC, are initialized to empty maps (Lines 3–4). Then, the for loop from Line 5 to Line 13 iterates over each test contained in the application under test's test suite.

In the body of the loop, Line 6 retrieves E_t, the set of entities covered by the t from the Coverage mapping. Line 7 calculates M_t, the set of methods covered by test t, by transforming the set of entities covered by t to their containing method using MethodOf. Line 8 filter M_t based on whether a method is directly invoked by t. The result, M_d, contains the methods that are directly invoked.

The for loop from Line 9 to Line 12 iterates over the directly covered methods to exclude directly covered entities from $\mathcal{E}$. Line 10 computes E_m, the subset of entities covered by t that are contained in method m, by filtering the set of all entities covered by t. In practice, instead of filtering the set of entities for each directly covered method, an additional mapping $M_{m \to E} \subseteq M_d \times \mathcal{P}(E)$ could be computed for all methods in M before Lines 9. Line 10 updates the set $\mathcal{E}$ removing E_m, which is the set of entities directly covered by m. When the loop exits, $\mathcal{E}$ is left with indirectly covered entities. Line 14 computes IC by grouping the entities in $\mathcal{E}$ by the methods that the entities belong to. Analogously, line 15 computes DC by grouping the directly covered entities (i.e., the difference of $\mathcal{E}'$ and $\mathcal{E}$). Finally, DC and IC are returned.

Note that, while the concepts of direct coverage and indirect coverage are agnostic to the type of entity that is being covered, in the remainder of the section I will focus on statement coverage. I choose to focus on statement coverage because, due to its simplicity and availability of tool support, it is the most commonly used coverage metric in practice.

4.1.1.3 Identifying Insufficiently Tested Methods

Given the DC and IC mappings produced by Algorithm 1, it is possible to identify methods that are insufficiently tested by computing each method's direct and indirect coverage scores:

$$DirectCoverage(m) = \frac{|DC[m]|}{|DC[m]| + |IC[m]|}$$
$$IndirectCoverage(m) = \frac{|IC[m]|}{|DC[m]| + |IC[m]|}$$

Analogously to how traditional coverage scores can indicate insufficiently tested code by identifying areas where large numbers of entities are uncovered, direct and indirect coverage scores indicate insufficiently tested methods by identifying methods that have a small percentage of directly covered entities or a high percentage of indirectly covered entities.

4.1.1.4 Prototype implementation

The implementation of my technique consists of three components, a coverage profiler,[1] a call graph tracer[2], and a direct coverage calculator. The coverage profiler uses WALA, a static analysis framework developed by IBM[3], to find out the tests in applications. It then executes each test individually and uses the Jacoco framework to record the test's coverage. The output of running each test is used to define

[1] Available at: https://bitbucket.org/huoc/icdc

[2] Available at: https://bitbucket.org/huoc/icdctracer

[3] http://wala.sourceforge.net

```
 1. public class Example {
 2.
 3.   public int m1(int a, int b) {
 4.     return a + b + m4(a, b);
 5.   }
 6.
 7.   public int m2(int a, int b) {
 8.     return a + (2 / b);
 9.   }
10.
11.   public int m3(int a) {
12.     return m2(a, 2);
13.   }
14.
15.   public int m4(int a, int b) {
16.     return a - b;
17.   }
18. }
```

(a) Application under test.

```
public class ExampleTest {
  public void test1() {
    Example e = new Example();
    assertEquals(3, e.m1(1, 2));
  }

  public void test2() {
    Example e = new Example();
    assertEquals(3, e.m3(1) + e.m4(1, 2));
  }
}
```

(b) Corresponding test suite.

Figure 4.1: Example code for illustrating direct coverage and indirect coverage.

the Coverage mapping explained in Section 4.1.1.2. It then builds up the mapping MethodOf to relate the entities to the methods with the help of WALA. Finally, the Direct mapping is built using the JVMTM Tool Interface (JVMTI) to track the direct invocations from the tests. Every time a method is invoked, the tracer will check if the caller is a test or some auxiliary method in the test suite. If the caller is a test or some auxiliary method in the test suite, the callee is considered directly covered. The tracer stores the execution traces for direct invocations. Finally, I implemented the algorithm described in Section 4.1.1.2 to compute the direct coverage and indirect coverage of the test suite.

4.1.1.5 An Illustrative Example

As a concrete example of computing direct statement coverage and indirect statement coverage, consider Figure 4.1 which shows the four methods that constitute

an application under test (Figure 4.1a) and the application under test's corresponding test suite (Figure 4.1b).

The input mappings, Coverage, Direct, and MethodOf for this example are shown below:

$$\text{Coverage} = \begin{cases} \text{test1} & \rightarrow \{\text{s4}, \text{s16}\} \\ \text{test2} & \rightarrow \{\text{s8}, \text{s12}, \text{s16}\} \end{cases}$$

$$\text{Direct} = \begin{cases} \text{test1} & \rightarrow \{\text{m1}\} \\ \text{test2} & \rightarrow \{\text{m3}, \text{m4}\} \end{cases}$$

$$\text{MethodOf} = \begin{cases} \text{s4} & \rightarrow \text{m1} \\ \text{s8} & \rightarrow \text{m2} \\ \text{s12} & \rightarrow \text{m3} \\ \text{s16} & \rightarrow \text{m4} \end{cases}$$

The coverage mapping indicates that test1 covers Statements 4 and 16 while test2 covers Statements 8, 12, and 16; the direct invocation mapping indicates that test1 directly invokes method m1 while test2 directly invokes methods m3 and m4; and the containing method mapping indicates that Statement 4 is contained in method m1, Statement 8 is contained in method m2, Statement 12 is contained in method m3, and Statement 16 is contained in method m4.

The DC and IC mappings computed by Algorithm 1 are as follows:

$$DC = \begin{cases} \text{m1} & \rightarrow \{\text{s4}\} \\ \text{m2} & \rightarrow \emptyset \\ \text{m3} & \rightarrow \{\text{s12}\} \\ \text{m4} & \rightarrow \{\text{s16}\} \end{cases}$$

$$IC = \begin{cases} \text{m1} & \rightarrow \emptyset \\ \text{m2} & \rightarrow \{\text{s8}\} \\ \text{m3} & \rightarrow \emptyset \\ \text{m4} & \rightarrow \emptyset \end{cases}$$

For this example, the mappings show that Statements 4, 12, and 16 are directly covered by the test suite, while Statement 8 is indirectly covered. Statement 8 is the only indirectly covered entity because all of the other statements are directly covered by either test1 or test2. Given this output, the direct coverage scores of methods m1, m2, and m4, are 100 % while the direct coverage score of method m3 is 0 %. Although the test suite achieves 100 % statement coverage, the direct coverage scores suggest that method m2 may be insufficiently tested.

In this particular case, the fact that Statement 8 is never directly covered means that the potential division by zero error that it contains may not be detected. Because m2 is only called by m3, the argument to m2 is always 2. By pointing out that method m2 has low proportion of directly covered statements, the approach guides testers to the portions of their applications that can benefit from additional testing.

4.1.2 Empirical Study

I conducted an empirical study that applied my proposed approach on 17 real world applications. This subsection describes the design details of my empirical study, including the research questions and subject applications. This empirical study is designed to answer the following research questions:

RQ1—Presence Does indirect coverage exist in real world applications?

RQ2—Significance Is the possibility of finding a fault influenced by whether the code containing the fault is directly or indirectly covered?

RQ3—Distribution How are indirectly covered statements distributed throughout an application?

Table 4.1: Considered applications.

Subject	LoC	# Tests
Apache XML Security	20,396	52
Barbecue	4,129	172
Commons Beanutils	11,375	973
Commons CLI	1,978	187
Commons CLI2	11,231	470
Commons Collections	26,414	2,563
Commons IO	26,614	882
Commons Language	23,070	2,044
Crammer	20,034	185
Crystal VC	8,031	80
DecentXML	12,741	714
HTML Parser	31,216	764
HttpClient	48,994	894
JDom	16,154	1,257
JFreeChart	92,252	2,234
Joda-Time	86,797	3,962
Numerics4j	3,647	194

RQ4—Categorization What are the potential causes for indirectly covered methods?

4.1.2.1 Considered Applications

To investigate my research questions, I selected 17 Java applications with their associated developer-provided test suites as my research subjects. Table 4.1 lists the specific applications that I chose. The first column, *Subject*, shows the names of the selected projects. These projects were taken from a variety of open source repositories including: (1) SIR,[4] which provides a variety of open-source projects for empirical software engineering, (2) SourceForge,[5] a popular repository for open-source projects, and (3) Apache Commons,[6] a collection of reusable components. The second column, *LoC*, shows the number of source lines of code in the Java files of each subject. The

[4] http://sir.unl.edu

[5] https://sourceforge.net

[6] http://commons.apache.org

Table 4.2: Direct and indirect coverage in percentage.

Subject	Coverage	% DC	% IC
Apache XML Security	42.4	79.4	20.6
Barbecue	84.0	75.3	24.7
Commons Beanutils	74.1	57.9	42.1
Commons CLI	93.1	66.2	33.8
Commons CLI2	95.7	71.3	28.7
Commons Collections	84.7	65.9	34.1
Commons IO	80.0	83.4	16.6
Commons Language	91.8	90.6	9.4
Crammer	59.2	53.7	46.3
Crystal VC	40.9	63.5	36.5
DecentXML	72.8	40.5	59.5
HTML Parser	61.1	50.3	49.7
HttpClient	69.2	77.8	22.2
JDom	71.2	77.1	22.9
JFreeChart	69.0	67.0	33.0
Joda-Time	89.1	84.4	15.6
Numerics4j	94.2	77.0	23.0

third column, *# Tests*, shows the number of tests included in each application's test suite.

I chose these specific applications for several reasons. First, in general, they are popular and widely used. Second, the applications cover a variety of subject domains. For example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, Joda-Time is a library for handling dates and times, etc. Third, the applications vary in size. For example, JFreeChart has over 90,000 lines of code, while Commons CLI has approximately 2,000 lines of code. Finally, the test suites also vary in size. The test suites for some of the applications contain nearly 4,000 tests, while others contain fewer than 100. Selecting test suites and applications of various sizes and subject domains improves the generalizability of my results.

4.1.2.2 Presence

The purpose of my first research question is to determine how the statements in my subject applications are covered by the test suites. Because I am proposing to identify insufficiently tested code based on indirect coverage, it is important to understand how common indirect covered code is in real applications. To answer this question, I first computed the overall direct and indirect coverage scores for each of my subject applications. The results of this computation are shown in Table 4.2.

In Table 4.2, the first column, *Subject*, again shows the names of my experimental subjects. The second column, *Coverage*, shows the overall statement coverage achieved by the application's test suite. The third and fourth columns, *% DC* and *% IC*, show the application's direct coverage score and indirect coverage score, respectively. That is, of the covered statements, the percentage that are directly covered and the percentage that are indirectly covered. The data shows that the percentage of indirect coverage ranges roughly from 10 % to 60 %. This suggests that, even for real test suites, the proportion of indirectly covered code in an application can be significant.

To gain some additional insights into this data, I checked whether there is any correlation between an application's statement coverage score and the percentage of statements that are indirectly covered. To compute the correlation, I used R version 2.14.1's implementation of the Pearson correlation coefficient. The computed correlation coefficient is -0.35 which indicates a very weak negative correlation. That means that, in practice, it is not possible to infer the amounts of direct or indirect coverage by considering only the traditional coverage score. It is necessary to compute direct coverage and indirect coverage scores directly.

4.1.2.3 Significance

The purpose of my second research question is to determine whether how a statement is covered impacts the effectiveness of the test suite. More specifically, I am interested in knowing if faults located in indirectly covered statements are less likely

Table 4.3: Mutants covered and killed.

Subject	# Mutants		# Killed		Mutation Score			
	IC	DC	IC	DC	IC	DC	% Change	p value
Barbecue	81	668	35	362	43.2	54.2	11.0	7.7×10^{-9}
Commons Beanutils	567	1,130	292	678	51.5	60.0	7.5	5.6×10^{-9}
Commons CLI	185	453	135	357	73.0	78.8	5.8	2.5×10^{-3}
Commons CLI2	476	1,110	309	911	64.9	82.1	17.2	2.2×10^{-16}
Commons Collections	1,150	4,011	712	3,004	61.9	74.9	13.0	2.2×10^{-16}
Commons IO	519	3,570	394	3,000	75.9	84.0	8.1	2.2×10^{-16}
Commons Language	1,031	12,575	726	9,496	70.4	75.5	5.1	2.2×10^{-16}
HTMLParser	2,163	2,443	1,220	1,558	56.4	63.8	7.4	7.3×10^{-14}
JDom	491	1,059	298	703	60.7	66.4	5.7	7.4×10^{-5}
Joda Time	693	2,111	465	1,532	67.1	72.6	5.5	3.2×10^{-8}
JFreeChart	10,001	16,681	2,915	9,171	29.2	55.0	25.8	2.2×10^{-16}
Numerics4j	65	665	35	484	30.8	72.5	41.7	2.2×10^{-16}

to be detected by the application's test suite than faults located in directly covered statements.

Because it is difficult to identify a suitable number of real faults with the necessary uniform distribution among directly and indirectly covered code, I chose to consider injected faults. More specifically, I considered mutants. Although mutants are artificial, recent work has shown that they can be a valid substitute for real faults [7, 63].

To generate the necessary mutants for my subjects, I used the MAJOR framework.[7] While MAJOR is a state of the art mutation testing framework, it was unable to complete the mutation testing process for five of the applications.

As part of performing mutation testing, MAJOR collects several pieces of useful information for each mutant that it generates, including: the location of the mutant, in terms of the containing class and method names and the line number; the mutation operator used to generate the mutant; and whether the mutant was detected by the application's test suite (i.e., if the mutant was killed). Because I know the location of the mutant, I can identify whether each mutant is located in directly covered code or indirectly covered code.

[7] http://mutation-testing.org

To determine whether there is a significant difference in the ability of a test suite to detect a fault depending on how the fault is covered, I used the binomial test. As an example of how to compute the test, let N_d be the number of mutants located in directly covered code, N_i be the number of mutants located in indirectly covered code, K_d be the number of mutants located in directly covered code that are killed by the test suite and K_i be the number of mutants located in indirectly covered code that are killed by the test suite. Then the binomial distribution $B(N_d, K_i/N_i)$ is used to calculate the probability of K_d or more kills in a sample of size of N_d, given the assumption that the probability of killing a mutant is K_i/N_i. Informally, my null hypothesis is that the location of the mutant does not affect the likelihood that it is killed. I used R version 2.14.1's implementation of the test (i.e., `binom.test`) with the one-sided option (i.e., `alternative="greater"`).

Table 4.3 shows the twelve subjects for which MAJOR was able to generate mutants. The first column, *subject*, shows the name of each subject. The second and third columns show the number of mutants generated by MAJOR that are located in indirectly covered code (IC) and the number that are located in directly covered code (DC). The fourth and fifth columns show the number of mutants, located in either directly covered code (DC) or indirectly covered code (IC) that were detected by the application's test suite. The sixth and seventh columns show the mutation scores for indirectly covered mutants (IC) and directly covered mutants (DC), that is the ratio of killed mutants to total mutants. The eighth column, *% Change* shows the percentage change in mutation score when comparing the mutation score for mutants located in indirectly covered code to the mutation score for mutants located in directly covered code. Finally, the last column, *p value*, shows the p value computed by the binomial test for each subject.[8]

As the data shows, for all twelve subjects, there is a statistically significant difference in the likelihood that a mutant is killed depending on how the mutant is

[8] 2.2×10^{-16} is the minimum value can be shown by R in the binomial test.

covered by the test suite. Moreover, the percentage change in mutation score ranges from 5.5 % to 41.7 %. This means that not only is the effect of how a mutant is covered significant, the size of the effect can be large as well. These results support my assumption that indirectly covered code is less effectively tested than directly covered code and should be brought to the attention of testers.

An entity is identified as directly covered once a test directly covers [9] the entity. So for a killed mutant in direct coverage, it is possible that the mutant is killed only by the tests that indirectly cover it. If it was the case for most killed mutants in direct coverage, it might not be plausible to encourage developers to directly cover what was indirectly covered. For this reason, I conducted another experiment that provides the details how the mutants in direct coverage are killed. The first column in Table 4.4 shows a subset of 7 subjects used in this experiment since it is much more expensive to specify which individual test methods killed the mutant. The second column, *Total*, shows the total numbers of killed mutants in direct coverage for each subject. The numbers are slightly different from the numbers from Table 4.3 since there have been slight changes in the mutation analysis framework. The next three columns show the numbers of the mutants that have been killed only by the tests that indirectly cover the mutants, only by the tests that directly cover the mutants, and both. The last three columns show the percentage of the mutants that killed in each manner. The *Indirectly Only* percentages show that only a minor portion of mutants in direct coverage are killed only by the tests that indirectly cover the mutants. The *Both* percentages show that the majority of killed mutants in direct coverage are killed by both tests that directly and indirectly cover the mutants. Moreover, the *Direct Only* percentages are more or less greater than the percentages of *Indirect Only*. The result provides evidence that the developers can improve the test suite by directly covering the indirect coverage.

[9] For a mutant in direct coverage, if *a test* is said to directly cover a mutant, that means the test covers the mutant's location by directly invoking the method that contains the mutant.

Table 4.4: Mutants killed in direct coverage.

Subject	# Killed Mutants in Direct Coverage				# Percentage		
	Total	Indirectly Only	Both	Directly Only	% Indirectly Only	% Both	% Directly Only
Barbecue	172	27	97	48	15.7	56.4	27.9
Commons CLI	330	112	90	128	33.9	27.3	38.8
Commons CLI2	797	261	260	276	32.7	32.6	34.7
Commons Collections	2,591	331	1,859	401	12.8	71.7	15.5
Commons IO	2,814	402	1,935	477	14.3	68.8	16.9
Commons Language	8,208	497	7,007	704	6.1	85.3	8.6
Numerics4j	452	65	287	35	15.7	63.5	20.8

4.1.2.4 Distribution

The purpose of my third research question is to learn about the distribution of indirectly covered statements in each of the applications. By learning how indirectly covered statements are distributed, I can determine the appropriate level at which to report insufficiently covered code to developers. If indirectly covered statements are evenly distributed, then reporting them individually is the only option. However, if indirectly covered statements are clustered, reporting indirectly covered code at a higher granularity can be more useful.

To investigate how indirectly covered statements are distributed, we computed the indirect coverage scores of each method in each subject application. The results of this calculation are shown in Figure 4.2. This figure shows several plots, one for each subject, and one, *(all)*, for all applications together. Each individual plot presents a histogram that shows the distribution of indirectly covered statements in the application. The y-axis shows the percentage of indirect statement coverage grouped into three bins: 0 % indirect coverage, 1 % and 99 % indirect coverage, and 100 % indirect coverage. The y-axis shows the percentage of methods whose indirect coverage score falls within the corresponding bin. For example, for apache-xml-security, approximately 20 % of its methods are completely directly covered, approximately 3 % of its methods have indirect coverage scores between 1 % and 99 %, and approximately 77 % of its methods are completely indirectly covered.

As the data shows, for all the applications, the distribution of the methods is primarily binary. In general, methods are either completely directly covered or

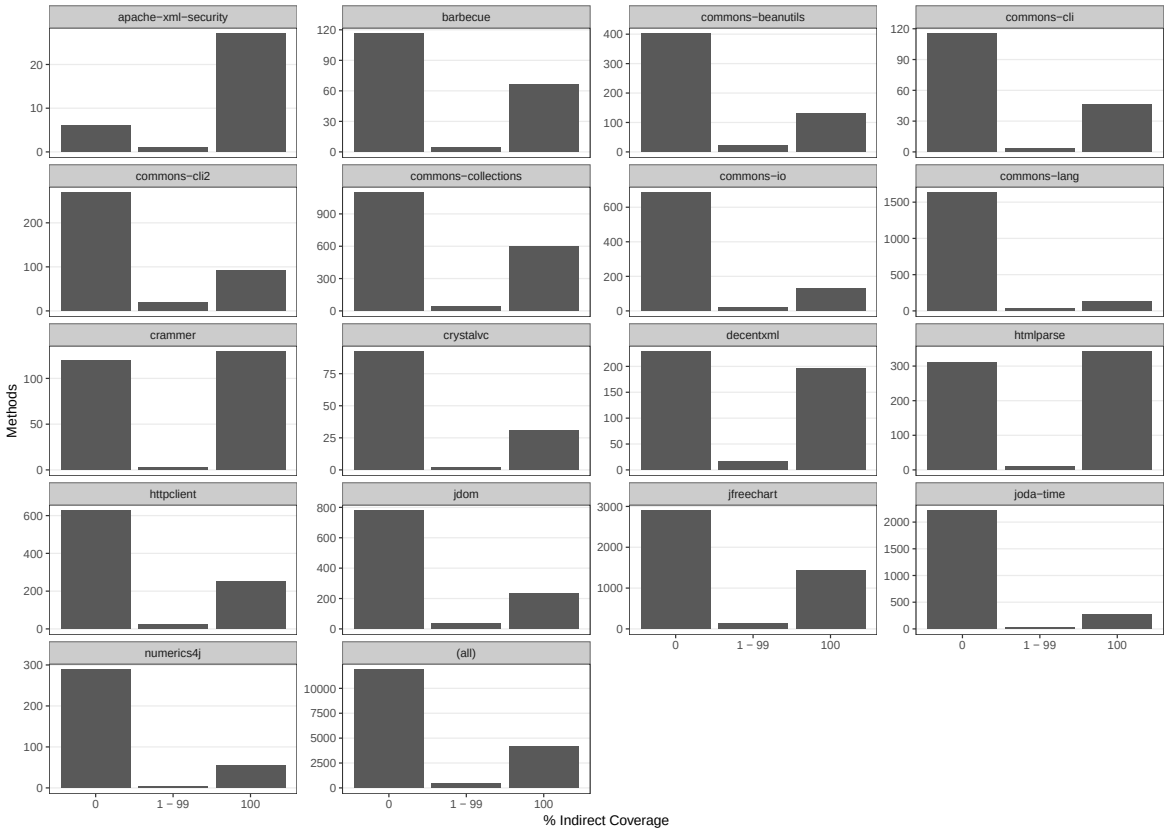


Figure 4.2: The distribution of indirectly covered statements of each application.

completely indirectly covered. Across all applications, less than 3 % of the methods have indirect coverage scores between 1 % and 99 %. This suggests that reporting indirectly covered statements at a method level will be effective at helping guide testers to indirectly covered code.

4.1.2.5 Categorization

The purpose of my fourth research question is to determine the potential causes of indirectly covered code. Understanding why indirectly covered code occurs in applications can help testers preemptively address why the code is indirectly covered and improve their test suites.

To determine the causes of the indirectly covered code, I manually investigated

Table 4.5: Methods with partial indirect covereage.

		Overloading		Inheritance		
Subject	Total	# Methods	# Unique	# Methods	# Unique	Other
Apache XML Security	2	0	0	0	0	2
Barbecue	5	1	1	1	1	3
Commons Beanutils	23	14	12	3	3	6
Commons CLI	4	2	2	0	0	2
Commons CLI2	20	5	5	4	4	11
Commons Collections	43	11	9	7	7	25
Commons IO	22	16	15	1	1	5
Commons Language	37	30	26	6	5	1
Crammer	3	2	2	0	0	1
Crystal VC	2	0	0	0	0	2
Decent XML	18	9	9	2	2	7
HTML Parser	9	4	4	0	0	5
HttpClient	25	3	3	4	4	18
JDom	36	14	10	1	1	21
JFreeChart	126	32	30	31	15	63
Joda Time	24	9	8	2	2	13
Numerics4j	4	1	1	0	0	3
Total	403	153	137	62	45	188

all of the methods in my subjects with indirect coverage. As a result of this investigation, I classified the methods into three groups: Overloading, Inheritance, and Other. The Overloading group contains methods that appear to be indirectly covered due to overloading among methods in the same class. For example, if a class contains more than one method with the same name and return type and at least one of the overloaded methods is directly covered, I consider the cause of any indirectly covered overloaded methods to be Overloading. Similarly, the Inheritance group contains methods that appear to be indirectly covered due to inheritance among subclasses and super classes. For example, if super class contains a method that is indirectly covered, while a subclass's implementation of the method is directly covered, I consider the cause of the indirectly covered method to be Inheritance. Finally, the last group Other contains methods that are indirectly covered for another reason (e.g., a tester may simply have forgotten to directly test the method).

Table 4.6: Methods with full indirect covereage.

Subject	Total	Overloading		Inheritance		Other
		# Methods	# Unique	# Methods	# Unique	
Apache XML Security	27	0	0	2	1	25
Barbecue	116	12	8	33	11	71
Commons Beanutils	132	80	47	22	14	30
Commons CLI	46	16	10	6	2	24
Commons CLI2	93	27	17	10	8	56
Commons Collections	597	128	79	101	53	368
Commons IO	133	73	47	5	4	55
Commons Language	133	46	20	6	5	81
Crammer	129	24	18	6	2	99
Crystal VC	32	2	1	1	1	29
Decent XML	228	56	47	16	6	156
HTML Parser	313	79	50	54	35	180
HttpClient	252	62	43	21	15	169
JDom	223	79	32	15	10	129
JFreeChart	1,443	347	254	139	80	957
Joda Time	263	93	51	55	28	115
Numerics4j	56	40	39	3	3	13
Total	4,216	1,164	756	495	276	2,557

The results of this classification are shown in Tables 4.5 and 4.6. Table 4.5 shows the results for methods whose percentage of indirectly covered statements is between 1 % and 99 % while Table 4.6 shows the results for methods that are completely indirectly covered. In each table, the first column, *Subject* shows the name of each subject. The second column, *Total*, shows the number of indirectly covered methods. The third and fourth columns, *Overloading*, show the total number of methods in the Overloading category (*# Methods*) and the total number of unique method names (*# Unique*). For example, for Commons Beanutils, there are 14 methods in the Overloading category, but two of them have the same name as another indirectly covered method. The fifth and sixth columns, *Inheritance* show the same information for the methods in the Inheritance group. Finally, the last column, *Other*, shows the number of methods in the Other category.

As the data shows, for methods with partial indirect coverage, 53 % are caused

by either overloading or inheritance and for methods with 100 % indirect coverage, 40 % are caused by either overloading or inheritance. This suggests that both overloading and inheritance are common causes of indirect coverage and should be given special attention in the testing process.

In the remainder of this subsection, I will provide specific examples of methods in the overloading and inheritance categories and explain why they are likely to be insufficiently tested.

```
// 0% Indirect coverage
public static FileOutputStream openOutputStream(File
file) throws IOException {
  return openOutputStream(file, false);
}

// 100% Indirect coverage
public static FileOutputStream openOutputStream(File
file, boolean append) throws IOException {
  return new FileOutputStream(file, append);
}
```

Figure 4.3: An example for an overloading group where one method is completely indirectly covered.

Figure 4.3 shows an excerpt of an overloading group where one method is directly covered and another is completely indirectly covered. The method `openOutputStream` has two variants, one that accepts a File and a boolean as input (`openOutputStream(File, boolean)`) and one that accepts only a File as input (`openOutputStream(File)`). The single argument variant delegates to the multiple argument variant by supplying a default boolean parameter. While the single argument variant is directly covered by the associated test suite, the two argument variant is never directly covered. Its indirect coverage score is 100 %. In this case, the tester cannot access the boolean parameter, and any failures that depend on the parameter being true may not be found by the test suite.

Figure 4.4 shows an excerpt of an overloading group where one method is directly covered and the other is partially indirectly covered. The method `toInt` has two variants, one that accepts a String and an int as parameters and one that accepts only a String and delegates to the two argument variant by supplying a default int value.

```java
// 0% Indirect coverage
public static int toInt(String str) {
  return toInt(str, 0);
}

// 20% Indirect coverage
public static int toInt(String str, int defaultValue) {
  if(str == null) {
    return defaultValue;
  }
  try {
    return Integer.parseInt(str);
  } catch (NumberFormatException nfe) {
    return defaultValue;
  }
}
```

(a) Application under test.

```java
public void testToIntString() {
  assertTrue(NumberUtils.toInt("12345") == 12345);
  assertTrue(NumberUtils.toInt("abc") == 0);
  assertTrue(NumberUtils.toInt("") == 0);
  assertTrue(NumberUtils.toInt(null) == 0);
}

public void testToIntStringI() {
  assertTrue(NumberUtils.toInt("12345", 5) == 12345);
  assertTrue(NumberUtils.toInt("1234.5", 5) == 5);
}
```

(b) Corresponding test suite.

Figure 4.4: An example for an overloading group where one method is partially indirectly covered.

Unlike the previous example, the application's test suite does directly cover some of the statements in the two argument variant by calling it directly in `testToIntStringI`. However, because the calls to `toInt` in this test never pass in a null value for the String parameter, the method never returns the provided default value. Again, any failures related to this code path can not be detected by the test suite.

Figure 4.5 shows an excerpt of an inheritance group where one method is directly covered and the other is completely indirectly covered. The implementation of `widthInBars()` declared in the Module class, a concrete and public class, is never tested directly by the test suite while the implementation declared in the `CompositeModule` subclass is directly covered.

Figure 4.6 shows an excerpt of an inheritance group with partial indirect coverage. The class `ParentImpl` is the base of many subclasses whose method `canProcess`

```
// Module
// 100% Indirect coverage
public int widthInBars() {
  int sum = 0;
  for (int i = 0; i < bars.length; i++) {
    sum += bars[i];
  }
  return sum;
}

// CompositeModule extends Module
// 0% Indirect coverage
public int widthInBars() {
  int width = 0;
  for (Iterator iterator = modules.iterator();
       iterator.hasNext();) {
    Module module = (Module) iterator.next();
    width += module.widthInBars();
  }
  return width;
}
```

(a) Application under test.

```
protected Module getPreAmble() {
  CompositeModule module = new CompositeModule();
  if(drawingQuietSection) {
    module.add(QUIET_SECTION);
  }
  module.add(START[mode]);
  return module;
}

public void testQuietZoneWidthIsAtLeast10BarWidths() {
  assertTrue(barcode.getPreAmble().widthInBars() > 10);
}
```

(b) Corresponding test suite.

Figure 4.5: An example for an inheritance group where one method is completely indirectly covered.

has many variants. The `ParentImpl` has not been explicitly constructed, however, `ParentImpl.canProcess` has partial direct coverage. Unlike `Module` in the previous example, there are some subclasses that do not override `canProcess`. So when these subclasses are tested in the test suite, part of the method gets directly covered.

The later two examples for inheritance groups show that it is difficult for test programers to understand which siblings/ancestors/offsprings have not yet been used as the test inputs because of the various builders and dynamic binding for the big family of classes.

```
//class ParentImpl
public boolean canProcess(WriteableCommandLine cl,
                          String arg) {
  final Set triggers = getTriggers();

  if (argument != null) {
    char separator = argument.getInitialSeparator();
    // if there is a valid separator character
    if (separator != NUL) {
      final int initialIndex = arg.indexOf(separator);
      // if there is a separator present
      if (initialIndex > 0) {
        return triggers.contains(arg.substring(0,
                                 initialIndex));
      }
    }
  }
  return triggers.contains(arg);
}

// DefaultOption extends ParentImpl
public boolean canProcess(WriteableCommandLine cl,
                          String argument) {
  return (argument != null) &&
         (super.canProcess(cl, argument) ||
         ((argument.length() >= burstLength) &&
         burstAliases.contains(argument.substring(0,
                               burstLength))));
}
```

Figure 4.6: An example for an inheritance group where one method is partially indirectly covered.

4.1.3 Summary

In this section, I presented ICDC for interpreting coverage information to identify insufficiently tested methods. ICDC is a technique based on partitioning the set of covered entities into entities that are directly covered and entities that are indirectly covered. I also presented the results of an empirical study of 17 applications that demonstrates: (1) real test suites indirectly cover large portions of their corresponding applications, (2) faults located in code that is indirectly covered are significantly less likely to be detected than faults that are located in code that is directly covered, (3) the majority of methods are either completely directly covered or completely indirectly covered, and (4) a significant portion of indirectly covered methods are likely

due to testers improperly considering inheritance or method overloading relations. As a result, I believe that identifying indirectly covered methods can be an effective approach for helping testers improve the quality of their test suites by directing them to insufficiently tested code. In the next section, I will present a similar technique that identifies insufficiently tested code based on the association between the components of an application and the tests of the application.

4.2 Fostered Code: Identifying Incidentally Covered Code

Code entities are incidentally covered when these code entities are covered but no test is written intentionally for covering these code entities. I identify incidentally covered code as insufficiently tested because the test developers have not yet purposely tested the properties of the code. It is very likely that such incidentally tested code is only used as an auxiliary for tests with other purposes. My intuition for identifying incidentally covered entities rises from the association between application classes and test classes in various testing paradigms. In this section, I will present a novel technique, Fostered Code, that identifies whether covered code is just incidentally covered by exploring such associations. The rest of the section is organized as follows: Section 4.2.1 describes the details of fostered code including the formal definition and an algorithm to compute fostered code. Section 4.2.2 presents an empirical study investigating various aspects of fostered code on 7 real-world software projects. Section 4.2.3 presents case studies from the empirical study and demonstrates how to leverage fostered code information.

4.2.1 Fostered Coverage

This subsection describes background information necessary for understanding the remainder of the section. First, it demonstrates how to find fostered code using a motivating example. Second, it defines the concept of class-under-test with regard to a test class. It also provides an implementation for identifying the class-under-class for a given test class, which will be used in my empirical study. Third, it gives the formal

```
1.  public class TA extends Person {
2.    public TA(int loggedHours, int id){
3.      this.loggedHours = loggedHours;
4.      this.id = id;
5.    }
6.    public int getStudentId(){ return id; }
7.    @override public int getBase() {return 3000;}
8.    @override public int getIncome() {
9.      return this.loggedHours * HOURLY_RATE + getBase();
10.   }
11. }

12. public class Utils {
13.   public bool underPovertyLine(Person p) {
14.     return p.getIncome() < 11700;
15.   }
16. }

17. public class TATest {
18.   public void testConstructor() {}
19. }

20. public class UtilsTest {
21.   public void testUnderPovertyLine() {
22.     TA ta = new TA(loggedHours=400,id=123);
23.     assertTrue(Utils.underPovertyLine(ta));
24.   }
25. }
```

Figure 4.7: A motivating example.

definition of fostered code. Finally, it describes how the fostered code of a test suite can be calculated.

4.2.1.1 Motivating Example

In Figure 4.7, there are two application classes and their test classes. The `TA` class which represents instances of teaching assistants extends `Person` and implements `getBase()` and `getIncome()`. A teaching assistant is paid hourly plus a base salary. The `IncomeUtils` class provides utilities on a person's income, for example, determining if someone's income is under the poverty line. There is a fault in `getBase()` where the base for a teaching assistant should be 2,500 dollars instead of 3,000 dollars. `TATest` and `UtilsTest` are the two test classes for the two application classes. `TATest` contains one test method, `testConstructor()`. The bodies of the two tests are omitted in the

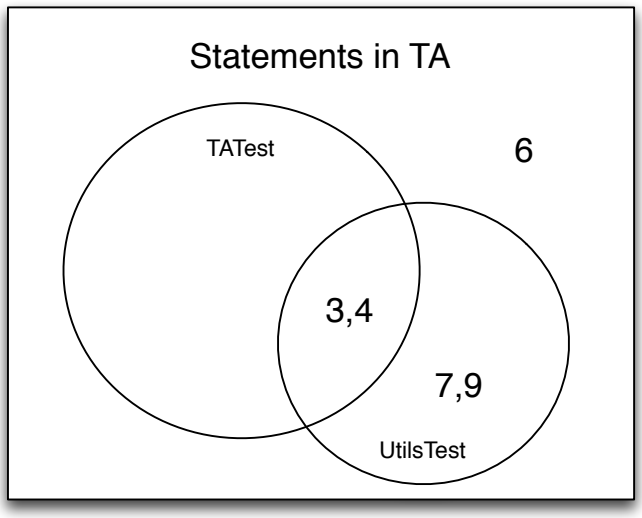


Figure 4.8: Test coverage on `TA`.

figure. Inside `testConstructor()`, a `TA` object is constructed and tested.

Figure 4.8 depicts the coverage information of class `TA` with respect to the two test classes. The rectangle represents all the statements from class `TA`. The circle labeled `TATest` represents the statements from `TA` covered by `TATest` and the circle `UtilsTest` represents the statements from class `Utils`. The intersection represents the entities covered by both test classes. Since there are two test classes in the test suite, the statements outside of the two circles are the ones that have not been covered. In this case, Line 6 is not covered because `getStudentId()` is never executed. The intersection of the two circles contains the statements of the constructor since it has been invoked by both test classes, as shown in the tests, Line 18 and Line 23. Line 7 and Line 9, `getIncome()` and `getBase()`, are covered by `UtilsTest` but not `TATest`. Although `getIncome()` and `getBase()` from class `TA` are *covered*, they are not covered by the test class designated for class `TA` itself.

The way that `getIncome()` and `getBase()` are tested is not sensitive enough to find potential problems in the methods. For example, if the hourly rate is 10 dollars, the `loggedHours` has to be between 870 hours and 919 hours to get an unexpected actual

outcome which is much greater than a reasonable amount of hours for a teaching assistant in an academic year. The root cause of the problem is the lack of corresponding tests in `TATest` so that `getIncome()` has never been intentionally and directly tested by the developers. Even if the fault happens to be detected by `testUnderPovertyLine` with an input within the narrow window (a `loggedHours` between 870 – 919 hours), it will be excessively difficult to trace back to the origin of the fault.

4.2.1.2 Class Under Test

The practice of unit-testing usually encourages a one-to-one or sometimes one-to-many correspondence between application classes and test classes. When an application class is large in size, it may have several test classes, each responsible for a particular purpose. Developers usually use texts, such as test names, to express the purpose of the tests and the association with the application classes (e.g., [118, 121, 129, 130]). Ideally, an application class `A` has a test class, usually named `TestA` or `ATest` and belongs to the same (Java) package. Each method in class `A`, e.g., `foo` and `bar`, will have their counterparts as `testFoo` and `testBar` in the test class. Such naming conventions preserve the correspondence between the application class and the test class(es) which reveals the intention of the developers.

In this research, I provide an implementation that utilizes the name association between the application class and the test class. The association will provide a mapping from a test class to an application class. Note that it allows multiple test classes for an application class. Ideally, an application class with name "A" will have an associated test class with name "TestA" or "ATest" – a postfix or prefix of "Test". While this postfix-prefix resolution works for many test-testee associations, there are a few exceptions. Some of them can still be resolved by name association with additional rules. For example, if the name of class `A` can be broken into three English words B, C, D, as `BCD`, the corresponding test class may have the name `CBDTest`. Or the corresponding test class does not belong to the same package with the class-under-test. Usually, a software project will have consistent naming convention for test classes

so people either find most or little of the associations.

4.2.1.3 Fostered Code

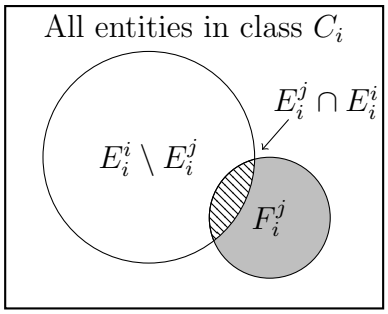
A set of code entities (e.g., branches, statements, etc.) is *fostered* when the entities are covered only by test(s) that are not designated for testing the entities. A set of code entities is *hosted* if the entities have been intentionally covered, the opposite of *fostered*. In this dissertation, I use *fostered coverage* as a metric to measure the proportion of entities that are fostered.

Let $\mathcal{P}(S)$ be the powerset of set S. Let $\mathcal{C}$ be the set of all the application classes $C_1, ..., C_n$. Let E_i be the set of entities in application C_i. Let $\mathcal{E}$ be all the entities in the application classes. Let $\mathcal{T}$ be the set of all test classes $T_1, ..., T_m$. A coverage mapping is defined on the powerset of all test classes to the powerset of all entities. $Cov : \mathcal{P}(\mathcal{T}) \rightarrow \mathcal{P}(\mathcal{E})$ is a mapping from a set of test classes to a set of entities that have been covered by the test classes. $classUnderTest : \mathcal{P}(\mathcal{T}) \rightarrow \mathcal{P}(\mathcal{C})$ is a mapping from a set of test classes to a set of application classes such that every application class has the tester-testee relation with at least one of the test classes. Fostered code is defined as the following:

Definition 4.2.1 *Let $E_i^j = Cov(T_j) \cap E_i$ be the set of the entities in C_i that have been executed by T_j. Let $H_i = \bigcup E_i^k, \forall k$ such that $C_i \in classUnderTest(T_k)$. F_i is the set of fostered entities of class C_i where $F_i = \bigcup_1^m E_i^k \setminus H_i$. The union of fostered code of each class, $\bigcup_1^n F_k$, is the fostered code of the application.*

Suppose there are two application class, C_i and C_j, and two test classes, T_i and T_j. T_i is the designated test class for C_i and T_j is the designated test class for C_j. Figure 4.9 illustrates the categorization of the entities in C_i. The larger circle on the left represents the entities covered by T_i and the smaller circle on the bottom right represents the entities covered by T_j. The gray area, F_i^j, is the fostered code of C_i. The line shaded area represents the entities in C_i covered by both test classes and can be written as $E_i^j \cap E_i^i$. The white area represents the entities covered only by $testC_i$, $E_i^i \setminus E_i^j$.

Figure 4.9: Code coverage description using the definitions.



Algorithm 2 shows the procedure for computing the fostered code of an application with a test suite. As input, the algorithm requires five pieces of information. The first is AUT, the application under test. The second is $\mathcal{T}$, the test suite of the application under test. The third is Coverage, a mapping that associates each test class in the test suite to a set of entities that are covered when the test class is executed. The fourth is $isClassUnderTestOf(C_i, C_j)$, a predicate that returns true when C_j is a test class and C_i is the class-under-test of C_j. The last is enclosingClass which is a mapping from an entity to the class that contains the entity.

Given the necessary inputs, the algorithm proceeds as follows. First, the output FC, a collection of entities, is initialized to all the entities covered by the test suite at Line 2. Then the loop from Line 3 to Line 7 iterates over each test class T_i and the entities that T_i covered. It then finds the covered entities from the class(es) under test. It updates FC by removing these entities from FC. When all the test class coverage information has been processed, FC is left with the entities that have never been tested by the associated test classes, i.e., fostered code.

While the concepts of fostered coverage are agnostic to the type of the entity (e.g., a branch, a statement, etc.) that is being covered, in the remainder of the section, I will focus on statement coverage. I choose to focus on statement coverage because, due to its simplicity and availability of tool support, it is the most commonly used coverage metric in practice.

Algorithm 2 Compute the fostered coverage.

Input: AUT, the application under test.
Input: $\mathcal{T}$, the test suite of the application under test.
Input: Coverage, a mapping that associates each test class $T \in \mathcal{T}$ to the set of entities covered by T.
Input: isClassUnderTestOf, a mapping that associates a test class C to the application under test. The mapping will return *null* if there is no class-under-test identified.
Input: enclosingClass, a mapping that associates an entity e to the application class that it belongs to.
Output: FC, a collection of entities that are fostered.

```
1: function computeFosteredCoverage
2:   FC ← {}

3:   for T_i ∈ 𝒯 do
4:     E_i ← Coverage[T_i]
5:     NotFosteredEntities ←
         E_i.filter{e ⇒
           C_j ← enclosingClass[e]
           isClassUnderTestOf(C_j, T_i)
         }
6:     FC ← FC \ NotFosteredEntities
7:   end for

8:   return FC
9: end function
```

4.2.2 Empirical Study

I conducted an empirical study that applies the fostered code identification technique on real world applications to validate the usefulness of my technique. This subsection describes the design details of my empirical study, including the research questions and subject applications.

4.2.2.1 Research Questions

The primary task of this work is to show the usefulness of the Fostered Code technique. The first step of the task is to show fostered code exists in real world applications and composes a reasonable amount of covered entities that is large enough be

concerned with. Once I can confirm the presence of fostered code, I would like to know if fostered code is insufficiently tested compared to hosted code. I will first compare the difference in the ability of a test suite to detect a fault, based on whether the fault is fostered. I expect that if a fault is fostered, in general, the fault is significantly harder to detect. Finally, if I can confirm fostered code is likely insufficiently tested, I then examine the causes. The above task can be condensed to the following three research questions:

RQ1—Presence. How much code is fostered in real world applications?

RQ2—Significance. Is it more difficult for a fault to be detected in fostered code?

RQ3—Root Cause. What is the root cause that makes fostered code different from hosted code?

4.2.2.2 Considered Applications

I selected 7 Java applications with their associated developer-provided test suites as my research subjects. Table 4.7 lists the specific applications. The first column, *Subject*, shows the names of the selected projects. The second column, *LoC*, shows the number of lines of source code in the Java files of each subject. The third column, # *Tests*, shows the number of tests included in each application's test suite.

I chose these applications for several reasons. First, in general, they are popular and widely used. Second, the applications cover a variety of subject domains. For example, Commons CLI2 is a library for processing command-line options, Commons IO is a library for performing various input/output operations, Joda-Time is a library for handling dates and times, etc. Third, the applications vary in size. For example, JFreeChart has over 90,000 lines of code, while Commons CLI2 has approximately 11,000 lines of code. The test suites also vary in size. The test suites for some of the applications contain nearly 4,000 tests while others contain fewer than 500. The applications also have application-test class associations that can be identified by the

Table 4.7: Considered applications.

Subject	LoC	# Tests
Commons CLI2	11,231	470
Commons IO	26,614	882
Commons Beanutils	11,375	973
Commons Collections	26,414	2,563
Commons Language	23,070	2,044
JFreeChart	92,252	2,234
Joda-Time	86,797	3,962

name association. Selecting test suites and applications of various sizes and subject domains improves the generalizability of my results.

4.2.2.3 RQ1:Presence

The purpose of my first research question is to determine how the statements in my subject applications are covered by the test suite with regard to fostered coverage. It is important to understand how common fostered code is in real applications. My implementation, with a few special cases, discovers all the associations between test classes and their class under test if there is one. For example, in subject `commons-cli2`, I found application class `HelpFormatter` being the class-under-test for test class `HelpFormatterTest`; there is no class under test for `Bug123Test` that appears to be an integration test to ensure a certain bug has been eliminated. The name association is a preliminary and static check that provides necessary information. It is possible that `HelpFormatterTest` does not execute any part of `HelpFormatter` and it is usually the case that `HelpFormatterTest` does not execute all the code entities from `HelpFormatter`. An implementation of Algorithm 2 uses the covered entities, obtained dynamically, to precisely identify fostered code.

Table 4.8 shows the fostered code results for these 7 subjects. The first column shows the names of the subjects. The second column, *Coverage*, shows the overall statement coverage achieved by the application's test suite. The third and fourth columns, $\% H$ and $\% F$, respectively, show the application's hosted and fostered coverage, which

Table 4.8: Hosted and fostered coverage in percentage.

Subject	Coverage	% H	% F
Commons CLI2	95.7	69.8	30.2
Commons IO	80.0	81.5	18.5
Commons Beanutils	74.1	41.3	58.7
Commons Collections	84.7	59.9	40.1
Commons Language	91.8	91.2	8.8
Joda-Time	89.1	55.3	44.7
JFreeChart	69.0	79.8	20.2

are the percentage of hosted and fostered code among all covered code. Recall that *hosted* code is code that is not fostered. The data shows that the percentage of fostered coverage ranges roughly from 10 % to 60 %. From this data, I can conclude that fostered code exists and is common in real-world applications.

4.2.2.4 Significance

The purpose of my second research question is to determine whether fostered code impacts the effectiveness of the test suite. I expect that if a fault is incidentally covered (fostered), in general, the fault is significantly more difficult to be detected. Because it is difficult to identify a suitable number of real faults with uniform distribution throughout an application, I chose to consider injected faults, more specifically, mutants. Recent work has shown that mutants, although being artificial, can be a valid substitute for real faults [7, 63].

Similar to the mutation analysis conducted in Section 4.1.2 for the ICDC technique, I used the MAJOR framework[10] to generate the mutants and perform mutant analysis. MAJOR collects several pieces of useful information for each mutant that it generates, including: the location of the mutant, in terms of the containing class and method names and the line number; the mutation operator used to generate the mutant; whether the mutant has been covered; and whether the mutant was detected

[10] http://mutation-testing.org

by the application's test suite (i.e., if the mutant was killed). With the location information of the mutant, I can identify whether each mutant is fostered with regard to the test suite.

Similar to the statistical significance test conducted in Section 4.1.2, I used the binomial test to determine whether it is significantly more difficult to detect a fault in fostered code. As an example of how to compute the test, let N_f be the number of mutants located in fostered code, N_h be the number of mutants located in hosted code, K_f be the number of mutants located in fostered code that are killed by the test suite and K_h be the number of mutants located in hosted code that are killed. The binomial distribution $B(N_h, K_f/N_f)$ is used to calculate the probability of K_h or more kills in a sample of size of N_h, given the assumption that the probability of killing a mutant is K_f/N_f. Informally, my null hypothesis is that the location of the mutant does not affect the likelihood that it is killed. I used R version 2.14.1's implementation of the test (i.e., `binom.test`) with the one-sided option (i.e., `alternative="greater"`).

Table 4.9 shows the results of mutation analysis on the seven subjects. The first column, *subject*, shows the name of each subject. The second and third columns show the number of mutants generated by MAJOR that are located in fostered coverage, with column name "F", and the number that are located in fostered coverage, with column name "H". The fourth and fifth columns show the number of mutants, located in either fostered coverage (F) or hosted coverage (H) that were detected by the application's test suite. The sixth and seventh columns show the mutation scores within the fostered coverage (F) and non-fostered coverage (H), which is the ratio of killed mutants to total mutants. The eighth column, *% Change* shows the percentage change in mutation score when comparing the mutation score for mutants within fostered coverage to the mutation score of the counterpart. Finally, the last column, *p value*, shows the p-value computed by the binomial test for each subject. [11] Since the significance level α is usually set at 0.05, the much lower p-values supported the rejection of the null

[11] 2.2×10^{-16} is the minimum value can be shown by R in the binomial test.

Table 4.9: Mutants covered and killed.

Subject	# Mutants		# Killed		Mutation Score			
	F	H	F	H	F	H	% Change	p value
Commons CLI2	265	876	142	710	53.6	81.1	27.5	2.2×10^{-16}
Commons IO	431	4,487	317	3,725	73.5	83.0	9.5	2.2×10^{-16}
Commons Beanutils	1,080	1,456	707	988	65.5	67.9	2.4	2.9×10^{-3}
Commons Collections	1,376	3,891	901	2,817	65.5	72.4	6.9	2.2×10^{-16}
Commons Language	866	15,499	594	11,564	68.6	74.6	6.0	2.2×10^{-16}
Joda Time	8,654	6,999	5,859	5,393	67.7	77.1	9.4	3.2×10^{-8}
JFreeChart	6,311	24,758	1,919	12,543	30.4	50.7	20.3	2.2×10^{-16}

hypothesis. Thus I concluded that fostered code is insufficiently tested.

4.2.2.5 Root Cause of The Difference

I hypothesize that fostered code lacks *direct* checks due to two possible reasons. One possibility is that the code is covered but not involved in assertions as pinpointed by Schuler and Zeller[99]. The other is that code is covered and involved in assertions, as depicted in the motivating example in Section 4.2.1.1, but much of the information associated with the code has been lost on the way through lossy computations towards the assertion. Both possibilities imply that fostered code lacks immediate checks.

I will provide evidence that fostered code lacks direct checks by placing immediate checks on values that are related to fostered code. Specifically, for each fault (mutant) inside the fostered code that has not been detected, I collect the *return value* from the method enclosing the code of interest immediately after the method returns. I also collect values of the *fields* from the instances of the enclosing class whenever they are accessed. I use the Java Debugger Interface (JDI) to construct a framework that taps the method returns and field accesses. I collect values from these events in temporal order as two sequences, one from the original version and one from the mutated version. If the sequence of the mutated version is different from the one of its counterpart, then this mutant is successfully detected. A mutant may alter the sequence in two different ways: (1) a change of data value, e.g., the fourth invocation

of a certain method returns a different value, or (2) an alternative control flow, e.g., a branch is skipped, which yields to a difference execution sequence.

Such additional checks in my empirical study are referred to as additional direct assertions (ADA) as this process simulates putting actual assertions on values related to fostered code except that the values are stored in a sequence and compared off-line. The expected values of the assertions are the values from the original version of the application and the actual values of the assertions are the values from the mutated version. ADA also refers to the process of applying additional direct assertions in the rest of the section. I anticipate gathering the following evidence from the empirical study: (1) ADA on fostered code kills a significant amount of previously not-detected mutants., (2) ADA on hosted code does not kill a significant amount of previously not-detected mutants., and (3) ADA will help fostered code achieve a similar level of mutation score with its hosted counterpart.

There are several advantages to answer the third research question in this manner. First, this approach provides not only evidence but also a solution which could be shown effective by observing the improved mutation score. Second, it is easier to reason about and write assertions for a certain purpose (e.g., put an assertion on the return value of a method) but rather difficult to reason about the purpose of an given assertion (e.g., determining related values in tests [59, 99]). Finally, I could the same technology on the mutants in the hosted code that haven't been detected. If the additional assertions cannot improve the likelihood for an error to be detected in code, the study provides evidence that fostered code lacks immediate checks.

Table 4.10 shows how ADA performs on previously not-detected mutants individually in both fostered and hosted code. The columns in Table 4.10 are identical to those in Table 4.9 except that the target mutants are the ones that survive the original test (so the column names for the mutants are prefixed with *L*s which indicate the mutants are live for the original test suites). The mutation score is calculated as the amount of the additionally killed mutants divided by the amount of the live mutants. The 7th column, *% Change*, shows the difference in percentage between

Table 4.10: Mutants detected additionally.

Subject	# Mutants		# Killed		Mutation Score			
	LF	LH	LF	LH	LF	LH	% Change	p value
Commons CLI2	124	166	52	19	41.9	11.4	30.5	2.2×10^{-16}
Commons IO	114	762	45	110	39.5	14.4	25.0	5.3×10^{-11}
Commons Beanutils	373	468	70	145	18.8	31.0	−12.2	NA
Commons Collections	475	1,074	94	110	19.8	10.2	9.5	4.4×10^{-10}
Commons Language	272	4,886	88	400	32.4	8.2	24.2	2.2×10^{-16}
Joda Time	2,795	1,606	635	293	22.7	18.2	4.5	3.2×10^{-8}
JFreeChart	4,392	12,215	2,126	2,485	48.4	20.3	28.1	2.2×10^{-16}

Table 4.11: Mutation scores after applying additional direct assertions.

Subject	Fostered			Hosted		
	before	improvement	after	before	improvement	after
Commons CLI2	53.6	19.6	73.2	81.1	2.2	83.2
Commons IO	73.5	10.5	84.0	83.0	2.5	85.5
Commons Beanutils	65.5	6.4	71.9	67.9	10.0	77.9
Commons Collections	65.5	6.8	72.3	72.4	2.8	75.2
Commons Language	68.6	10.2	78.8	74.6	2.6	77.2
Joda Time	67.7	7.3	75.0	77.1	4.1	81.2
JFreeChart	30.4	33.7	64.1	50.7	10.7	61.4

the additional kills. If the improvement in foster coverage is greater, the difference in percentage in the 7th column is positive. As shown in the table, six out of the seven subjects have positive values which means ADA results in better mutation score improvement in the fostered code than in the hosted code. `Commons-beanutils` is an outlier that the improvement is greater in the hosted code. Similar to Section 4.2.2.4, the binomial test is used to quantify the difference for those that have positive values. The p-value represents the significance of the difference between fostered code and its counterparts based on the following null hypothesis: *There is no difference in the likelihood of previously not-detected errors being found by ADA whether the faults are in fostered coverage or not for each subject.* This null hypothesis is rejected by the p-values calculated as the p-values are below the usual significance level (0.05), shown in the last column in Table 4.10. A similar binomial test is conducted on the fact that

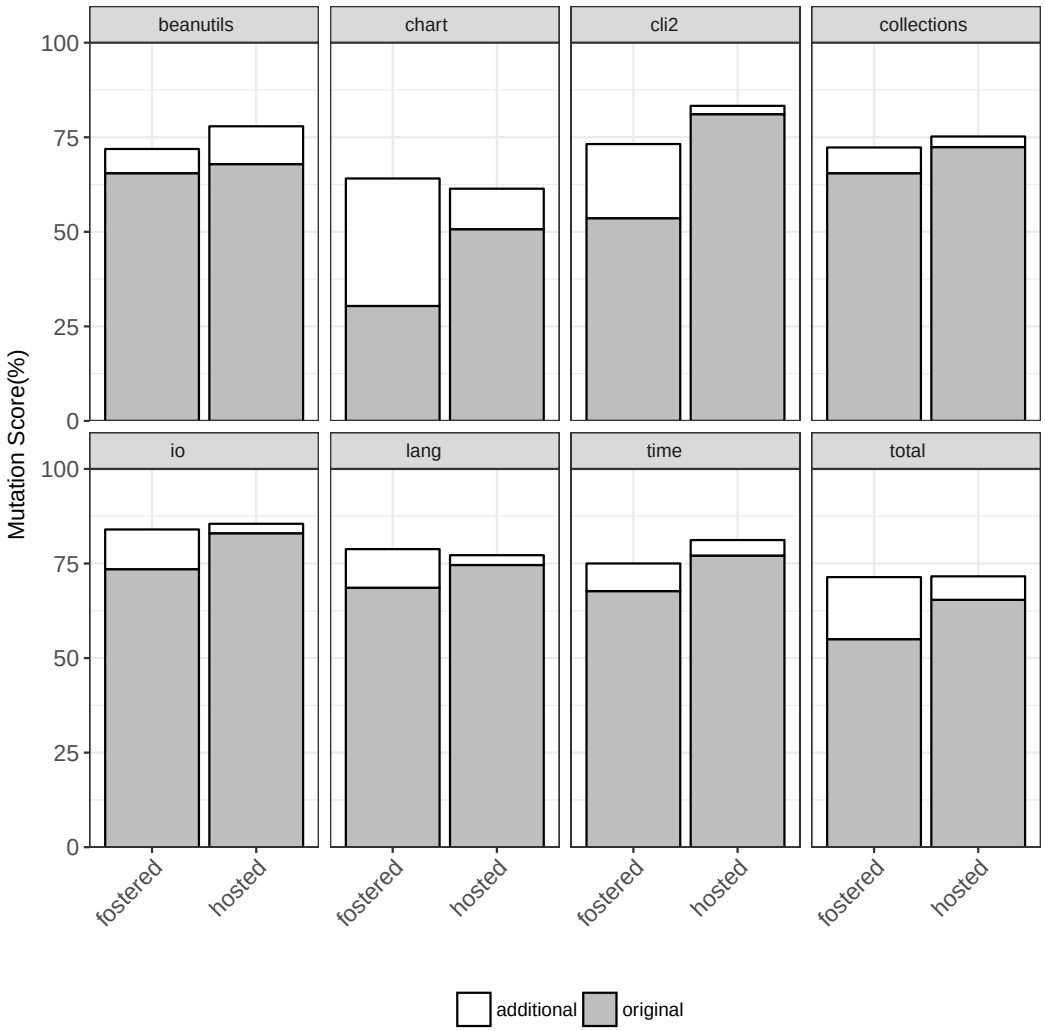


Figure 4.10: ADA improves mutation score better in fostered code and levels the mutation scores of fostered code and hosted code.

six out of seven subjects have better improvement in fostered code. The binomial distribution, `binom(7,0.5)`, gives a 0.0625 p-value for six positives. Although the p-value is slightly greater than the usual 0.05 significance level threshold, the 0.0625 p-value shows a noticeable difference. A weighted binomial test on the subjects, which tests on the total number of mutants, would favor the rejection of the null hypothesis because `JFreeChart` dominates in number.

Table 4.11 shows how ADA improves the overall mutation score respectively for the fostered and hosted coverage for each application, before and after ADA. Columns

2 to 4 show the mutation score before ADA, the mutation score improvement by ADA (based on total number of mutants) and the new mutation score after ADA, for fostered code. The same information for hosted code is shown from Column 5 to Column 7. By comparing Column 3 with Column 6, the respective improvement for fostered code and hosted code, the reader can see there are better improvements in fostered code for most applications. Moreover, with less improvement in hosted code, the difference in mutation scores between fostered and hosted code is narrowed after ADA for each application. Figure 4.10 illustrates the effects of ADA. Each facet in Figure 4.10 represents a subject except that the bottom-right facet represents the corresponding information of all the mutants from the subjects. For each facet, the bar on the left represents the information of fostered code and the bar on the right represents the information of hosted code. The gray area of a bar depicts the mutation score before ADA, and the white area stacking on the gray area represents the improvement in mutation score by ADA. The total height of a bar is the mutation score achieved after ADA in fostered code or hosted code for each subject.

4.2.3 Case Study

In this subsection, I present two mutants from two different applications that are fostered. I will show how the two mutants failed to be detected by the original test suite for the lack of direct checks and how the two mutants get detected with additional direct checks provided by ADA.

4.2.3.1 Case Study: Commons-CLI2

Subject `commons-cli2` provides utilities for command-line processing. The method `usage` in `HelpLineImpl` returns the usage as a string, as shown in Figure 4.11. The `cachedUsage` stores the result once the usage has been computed. In Figure 4.11, if there is no cached answer with the input settings (Line 2 to Line 4), the usage will be computed in the body from Line 5 to Line 12. Otherwise, the method returns the cached answer.

The `HelpLineImpl` class implements the interface `HelpLine` which represents a line of help information for a particular command-line option. The `HelpLine` interface is primarily used by `HelpFormatter`, which is an application class that formats the help output. In the test suite, there is no designated test class for `HelpLineImpl`. There is a test class `HelpFormatterTest` for the application class `HelpFormatter` which is the most closely related test class for `HelpLineImpl` and covers code of `HelpLineImpl`. In addition, there are four other test classes that execute code of `HelpLineImpl`. They are `CpTest`, `ParserTest`, `BugCLI18Test` and `NestedGroupTest`.

```
1. public String usage(final Set helpSettings, final Comparator comparator){
2.   if (cachedUsage == null
3.       || cachedHelpSettings != helpSettings
4.       || cachedComparator != comparator) {

5.       cachedHelpSettings = helpSettings;
6.       cachedComparator = comparator;

7.       final StringBuffer buffer = new StringBuffer();
8.       for (int i = 0; i < indent; ++i) {
9.         buffer.append("  ");
10.      }
11.      option.appendUsage(buffer, helpSettings, comparator);

12.      cachedUsage = buffer.toString();
13.  }
14.  return cachedUsage;
14.}
```

Figure 4.11: The `usage` method in `HelpLineImpl`.

Had there been a fault that turned any of the three conditions in Line 2 to Line 4 to a `true` value, the return value of the `usage()` method could not change. There will be difference in speed and space since in Line 11 in Figure 4.11, the `option` field cumulates the usage ever computed for the same `HelpLineImpl` object if the caching mechanism is not working correctly. Since the tests that cover this mutant are not designated for the `usage()` method, the checks in the tests could not be expected to (and did not) ensure that the caching mechanism works properly. However, since option is a field enclosed in class `HelpLineImpl` and modified by the `usage()` which encloses the mutant, ADA found the discrepancy between the values of the `option` field of the original version and the mutated version.

4.2.3.2 Case Study: Commons-IO

Subject `commons-io` has an abstract class `AbstractFileFilter` that contains 20 sub-classes, including `AndFileFilter`, `FalseFileFilter`, `OrFileFilter` and `WildCardFileFilter`. This inheritance relation is depicted in Figure 4.12 where a solid line indicates the destination is a sub-class of the origin. `FileFilterTestCase`, `AbstractFileFilter`'s corresponding test class, provides one test method for each sub-class of `AbstractFileFilter` as depicted in Figure 4.12. For example, method `testWildcard()` initializes `WildcardFileFilter` instances and puts fundamental checks on these instances, part of which is shown in Figure 4.14c. There are also methods like `testAnd()` and `testNot()`. Such a relation is shown in dotted lines in Figure 4.12. In other words, for each concrete sub-class of `AbstractFileFilter`, `AbstractFileFilter` provides one test method for the entire application class. Intuitively, `FileFilterTestCase` only provides integration tests because: (1) there is only one method for each application class, and (2) the tests only check the file filters' functionality but not their unit components, e.g., the constructors and different filtering functions.

Only three out of twenty sub-classes of `AbstractFileFilter` have their own designated test classes, e.g., `AndFileFilterTestCase` for `AndFileFilter`. Figure 4.12 shows two of the three test classes. However, they are not descendants of `FileFilterTestCase`. Instead, they are sub-classes of `IOFileFilterAbstractTestCase`. In Figure 4.12, such class-under-test relations are shown by connecting the test class and its class-under-test with a dashed line. In the following description, I demonstrate how faults in fostered code are less likely to be detected, by using two examples, one from a class that is only tested by a method from `FileFilterTestCase` and one from a class that is tested both by a method from `FileFilterTestCase` and its designated test class.

`WildCardFileFilter` is only tested by a test method `testWildCard()` in `FileFilterTestCase`. It is not hard to imagine that the single test method is not sufficient for the five constructors and three filtering related methods implemented in `WildCardFileFilter`. Figure 4.14 describes a mutant in one of the constructors and the

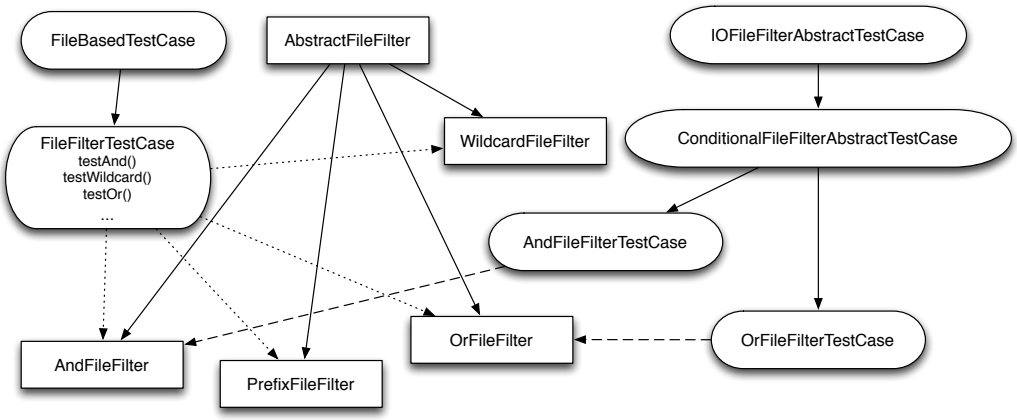


Figure 4.12: Solid lines connect classes with direct inheritance relations. Dashed lines connect classes between the test class and the class-under-test. Dotted lines connect classes an application class has a test method in `FileFilterTestCase`.

corresponding parts in the test method. Figure 4.14a shows the original code snippet of the constructor that takes in a string array and an `IOCase` instance. Figure 4.14b shows the mutated version which always assigns the second argument to the `caseSensitivity` field in Line 7. The check if the input `caseSensitivity` is `null` in the original version strongly implies the `caseSensitivity` field of a `WildCardFileFilter` should not be `null`. Figure 4.14c shows the corresponding part for the mutated constructor from the test method.

If the second argument is not `null`, there will be no difference between the original version and the mutated version. Otherwise, the `this.caseSensitivity` field will be `IOCase.SENSITIVE` for the original version and `null` for the mutated version. A `null` value is used as the second argument for constructing the second filter instance in the test method. Because of the fault introduced by the mutant, the `caseSensitivity` field of the new filter instance is `null` after the execution of a constructor, which is an erroneous state of the program. Unfortunately, due to the fact that `testWildCard()` serves more like an integration test, the defect is hidden and the test passes. As depicted in Figure 4.13, the mutated constructor will set `this.sensitivity` to `null`, which results in an erroneous state of the filter object. This erroneous state cannot reach the final and observable state in the test oracle because the auxiliary filtering method will

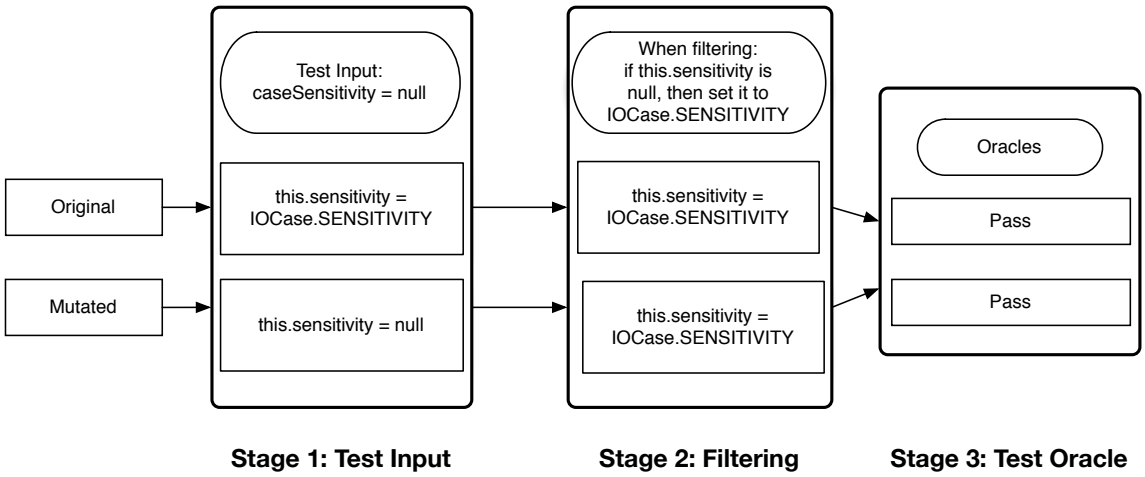


Figure 4.13: An illustration how the fault was not detected. A name filtering method will set a `null` sensitivity to a default value, thus hiding the fault in the filter constructor.

replace the `null this.sensitivity` field with the default value, `IOCase.SENSITIVE`. In other words, this auxiliary method reverts the system back to the correct state. Had there been a designated test for the constructor, which checks the outcome of the constructor before the application of the auxiliary method, the erroneous state could be detected. This is how ADA detected the fault.

Along with the test method `testAnd()` from `FileFilterTestCase`, application class `AndFileFilter` also has its designated test class `AndFileFilterTestCase`. Despite that the designated class provides detailed tests for individual methods in the application class, there is no test for the `toString()` method. `toString()` is (only) covered by one single line of `testAnd()`, as shown in Figure 4.15c. The assertion in the test method only checks whether `toString()` returns `null`. Even some developer left a note for better tests. No wonder the mutant that prepends an extra comma has been covered but not killed. The original `toString()` is shown in Figure 4.15a, and the mutated `toString()` is shown in Figure 4.15b where the mutant changes the if-condition on Line 8. A possible explanation for the lack of proper checks on `toString()` is that some of the sub-classes do not have `toString()` implemented. It would confuse test developers from the (higher) `FileFilterTestCase` perspective.

```
1. public WildcardFileFilter(String[] wildcards, IOCase caseSensitivity) {
2.   if (wildcards == null) {
3.     throw new IllegalArgumentException("The wildcard array must not be null");
4.   }
5.   this.wildcards = new String[wildcards.length];
6.   System.arraycopy(wildcard, 0, this.wildcards, 0, wildcards.length);
7.   this.caseSensitivity = caseSensitivity == null ?
8.                                 IOCase.SENSITIVITY : caseSensitivity;
9. }
```

(a) Original constructor.

```
1. public WildcardFileFilter(String[] wildcards, IOCase caseSensitivity) {
2.   if (wildcards == null) {
3.     throw new IllegalArgumentException("The wildcard array must not be null");
4.   }
5.   this.wildcards = new String[wildcards.length];
6.   System.arraycopy(wildcard, 0, this.wildcards, 0, wildcards.length);
7.   this.caseSensitivity = false ? IOCase.SENSITIVITY : caseSensitivity;
8. }
```

(b) Mutated constructor.

```
public void testWildcard() throws Exception {

  filter = new WildcardFileFilter(new String[] {"*.java", "*.class"});
  assertFiltering(filter, new File("Test.java"), true);
  assertFiltering(filter, new File("Test.class"), true);
  assertFiltering(filter, new File("Test.jsp"), false);

  filter = new WildcardFileFilter(new String[] {"*.java", "*.class"}, (IOCase) null);
  assertFiltering(filter, new File("Test.java"), true);
  assertFiltering(filter, new File("Test.JAVA"), false);
}
```

(c) Corresponding tests.

Figure 4.14: WildcardFileFilter Example.

```
1. @Override
2. public String toString() {
3.   StringBuilder buffer = new StringBuilder();
4.   buffer.append(super.toString());
5.   buffer.append("(");
6.   if (fileFilters != null) {
7.     for (int i = 0; i < fileFilters.size(); i++) {
8.       if (i > 0) {
9.         buffer.append(",");
10.      }
11.      Object filter = fileFilters.get(i);
12.      buffer.append(filter == null ? "null" : filter.toString());
13.    }
14.  }
15.  buffer.append(")");
16.  return buffer.toString();
17.}
```

(a) Original toString.

```
1. @Override
2. public String toString() {
3.   StringBuilder buffer = new StringBuilder();
4.   buffer.append(super.toString());
5.   buffer.append("(");
6.   if (fileFilters != null) {
7.     for (int i = 0; i < fileFilters.size(); i++) {
8.       if (i > -1) {
9.         buffer.append(",");
10.      }
11.      Object filter = fileFilters.get(i);
12.      buffer.append(filter == null ? "null" : filter.toString());
13.    }
14.  }
15.  buffer.append(")");
16.  return buffer.toString();
17.}
```

(b) Mutated toString.

```
assertNotNull(f.toString()); //TODO better tests
```

(c) Corresponding tests.

Figure 4.15: AndFileFilter Example.

The two examples show how promiscuous the test class arrangement can be when the application classes have complicated relations. If the test design does not start with a proper test-testee correspondence, it is almost inevitable that the developers overlook important aspects for testing the project.

4.2.4 Summary

In this section, I presented Fostered Code which is a technique that identifies insufficiently covered code by whether the code has been covered by its designated test(s). I conducted an empirical study on 7 real applications. The empirical study shows that fostered code is common in real applications and faults in fostered code are more difficult to be detected. The empirical study also shows that faults in fostered code are more difficult to be detected because fostered code lacks of direct checks.

This section makes the following contributions:

(1) A definition of *fostered code* which is a proxy for insufficiently tested code

(2) An efficient implementation of fostered code identification

(3) An empirical study to show that fostered code is significantly under-tested

(4) An empirical study that shows fostered code lacks of direct checks

For practical use of the Fostered Code technique, the developers need to determine the association between the application classes and the test classes. The feasibility of identifying the association depends on the testing paradigm which the developers use for a certain software project. If the developers decide to use the Fostered Code technique on their own projects, the developers should be able to identify their own association. The additional cost of the Fostered Code technique is mostly associated with identifying the class(es)-under-test. If the static name association is used, the cost is little comparing to running the tests. For each test class, the cost associated with the test class is to determine the class-under-test and the coverage information of the test class and then update the existing fostered coverage. Upon evolution of the

software project, if a test method is added to a test class, the additional cost is only associated with the test method for determining the additional code it covers. If the application has been changed, the whole test suite needs to be executed to determine the change in coverage.

4.3 Comparison Between ICDC and Fostered Code

The ICDC technique and Foster Code technique share common characteristics. First, both of them identify insufficiently tested code entities that have already been covered with additional information obtained within reasonable cost. Second, indirectly covered code might also be fostered code and vice versa. Third, the methodology of the two techniques is similar. The similarities and differences of the two techniques are discussed in the following sections.

4.3.1 Comparing the Usage

Despite the similarity, they each have their pros and cons. The ICDC technique has the advantage that it does not need the assumption that the class-under-test can be identified. The technique can always be applied to any application with a test suite, if method invocations are observable. Usually such method invocations can be observed using corresponding debuggers for different languages. The Fostered Code technique, however, is more restricted as it depends on the technique that is used to identify the class-under-test. If the name association is used for the identification, as in the implementation in Section 4.2, it is not guaranteed that there is a consistent association between application classes and test classes, especially when using the technique on third-party applications. However, if a development team conforms to their coding conventions, the team should be able to specify the association themselves and identify fostered code. On the other hand, if the name association can be identified, the fostered code technique requires even less effort to obtain the necessary information because it only involves reading the class names and matching them.

Table 4.12: *Phi* coefficient between ICDC and Fostered Code.

Subject	Both	Indirect Only	Fostered Only	Neither	ϕ
Commons CLI2	272	194	194	960	0.42
Commons IO	164	382	506	1,851	0.09
Commons Beanutils	1,028	180	287	817	0.66
Commons Collections	181	640	326	3,376	0.21
Commons Language	551	159	1,252	1,452	0.52
Joda-Time	2,011	562	1,510	4,100	0.62
JFreeChart	3,396	5,284	1,104	15,059	0.62

4.3.2 Comparing the Code Entities Identified

In addition to their differences and similarities in usage listed above, it is necessary to quantify how much overlap the two techniques have. That is, how often is a code entity fostered and also indirectly covered? I compared the fostered code and the indirectly covered code in the 7 subjects in Table 4.7. Since the ICDC technique only identifies code entities from public methods, for this comparison, I excluded code entities from non-public methods from the Fostered Code technique. A *phi* coefficient, usually denoted by ϕ, is computed for the two binary variables using *R* of version 3.2.3. The *phi* coefficient is a measure of association for two binary variables. It ranges from -1 to 1 where 0 indicates no relationship and ± 1 indicates perfect agreement or disagreement. The first column of Table 4.12 shows the 7 subjects used in this comparison. The four columns from the second through the fifth column contain the data necessary for computing the *phi* coefficient. Specifically, the second column named *Both* shows the numbers of code entities (statements) that are both identified as fostered and indirectly covered for each subject. The third column shows the numbers of code entities that are indirectly covered but not fostered. The fourth column shows the numbers of code entities that are fostered but not indirectly covered. The fifth column shows the numbers of code entities that are neither fostered nor indirectly covered. The last column gives the *phi* coefficient for each subject. The *phi* values provided in the table have been scaled with respect to the maximum possible ϕ value[39].

The *phi* values in every subject show that fostered code and indirectly covered

code are usually positively related. That is, when a code entity is fostered, it is likely also indirectly covered, and vice versa. In Table 4.12, 5 out of the 7 subjects have *phi* values close to or greater than 0.5 which indicates relatively strong agreement between the two techniques. It is intuitive because if one incidentally covered (fostered) code entity from a test class is not designated for these entities, it is very likely the entities are covered indirectly, and vice versa. However, none of the *phi* values is close to 1 which would indicate a nearly perfect agreement. This indicates that there are still considerable numbers of code entities that are, for example, covered in their designated test class only because the caller method has been used in tests or covered directly in a test which does not belong to the designated test class for these code entities.

4.3.3 Summary of Comparison

For usability, the two techniques vary in the following aspects: (1) The ICDC technique is more applicable in real-world applications than the Fostered Code technique, especially when evaluating a third-party application. This is because the relation between caller-callee is available among most software projects. However, the association between the application classes and the test classes vary in different designs and testing paradigms., and (2) The Fostered Code technique is more efficient by using an efficient method for identifying class-under-test, such as the name association. For the comparison of identifying insufficiently tested code entities, I find that: (1) The two techniques have positive correlations, which is consistent with intuition., (2) The positive correlation is relatively strong. That indicates, in the real world, if test developers forget or do not want to test a certain set of code entities in their designated test class, given the fact that the entities have been covered, they are probably covered indirectly only., and (3) The two techniques have neither perfect agreement nor subsumption relations. One of the common situations is that the test developers cover a few methods from another class to construct a test input of the class-under-test.

Chapter 5

CONCLUSIONS

5.1 Summary of Contributions

In this dissertation, I provide three techniques that can improve the quality of existing test suites. In summary, they are:

(1) a definition and an implementation of a new technique that can automatically analyze tests to detect both brittle assertions and unused inputs

(2) a definition and an implementation of a new technique to interpret coverage information based on the concepts of direct coverage and indirect coverage

(3) a definition and an implementation of a new technique to interpret coverage information based on the relation between the test class and the class-under-test

.

To be specific, *OraclePolish* is a new technique for automatically analyzing test oracles. The technique is based on dynamic tainting and can detect both brittle assertions—assertions that depend on values that are derived from uncontrolled inputs—and unused inputs—inputs provided by the test that are not checked by an assertion. I also present OraclePolish, an implementation of the technique that can analyze tests that are written in Java based on JUnit testing framework. Using OraclePolish, I conducted an empirical evaluation of the tool's performance on more than 4,000 tests from 19 real applications. The results of the evaluation demonstrate that OraclePolish is able to detect both brittle assertions and unused inputs in real tests. The evaluation identified 164 brittle assertions and 1,618 unused inputs among 4,718 tests in the applications. The evaluation also showed that the cost of the technique is

5 to 30 times than the cost for running the application on pure JVM, which can be considered reasonable for an automatic tool.

ICDC is a new approach for interpreting coverage information to identify insufficiently tested methods. The technique is based on partitioning the set of covered entities into entities that are directly covered and entities that are indirectly covered. I also presented the results of an empirical study of 17 applications that demonstrates: (1) real test suites indirectly cover large portions (10%–60%) of their corresponding applications, (2) faults located in code that is indirectly covered are significantly less likely to be detected than faults that are located in code that is directly covered, (3) the majority of methods are either completely directly covered or completely indirectly covered, and (4) a significant portion of indirectly covered methods are likely due to testers improperly considering inheritance or method overloading relations.

Fostered Code is a new approach for identifying insufficiently tested code entities. The technique is based on partitioning covered entities into entities that are intentionally covered and entities that are incidentally covered. I also presented the results of an empirical study of 7 applications that demonstrates: (1) real test suites foster a large portion (10%–60%) of code from their corresponding applications, (2) faults located in fostered code are significantly less likely to be detected, (3) there are less assertions on return values and field values related to fostered coverage, and (4) fostered code can be salvaged by adding assertions on related return values and field values. As a result, I believe that identifying fostered code can be an effective approach for helping testers improve the quality of their test suites by directing them to insufficiently tested code.

I anticipate that my research can improve the quality of test suites and thus help developers to reduce the cost of testing, debugging, and maintenance.

5.2 Future Work

The future work of this dissertation can also be divided with respect to each technique:

- For *OraclePolish*, I will implement the automated generation of recommendations to fix the reported oracle problems. The possible fixes follow very regular and specific patterns so that templates will be provided for the developers. I will investigate the possibility of extending the technique to analyze entire test suites rather than individual tests. This will allow the technique to more precisely handle certain situations, such as when logically connected assertions are split among multiple test cases (e.g., the one assertion per test style). I am also planning on conducting additional evaluations of the technique. In particular, I am interested in conducting human studies with testers to qualitatively assess the technique more fully, such as the importance that developers would give to such reported issues, and increase the number and type of subjects that I consider. Finally, I will investigate how my technique could be integrated with existing test generation approaches to improve the quality of the generated tests.

- For *ICDC*, I plan to investigate the insufficiently tested methods that my tool identified in more detail in order to expand the categorization of these methods. In addition, I will implement an automated tool for generating recommendations for possibly missing test inputs or oracles of the identified methods. Finally, I will extend my empirical evaluation to consider additional coverage metrics (e.g., branch coverage).

- For *Fostered Code*, I plan to investigate the following topics: (1) techniques that identify class-under-test efficiently other than name association, (2) the effectiveness of fostered coverage approach on other popular coverage criteria (e.g., branch coverage), (3) the focus of field values assertion when code is intentionally tested and narrow down the assertions in the generated test templates when removing fostered coverage, and (4) a generic design requirement to impose the criterion that code needs to be intentionally tested.

Bibliography

[1] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 29 (3):1–6, 2004.

[2] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[3] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12, 2011.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

[5] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[6] 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.

[7] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[8] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening SOA testing. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 161–170, 2009.

[9] Brady Benware, Chris Schuermyer, Sreenevasan Ranganathan, Robert Madge, Prabhu Krishnamurthy, Nagesh Tamarapalli, Kun-Han Tsai, and Janusz Rajski. Impact of multiple-detect test patterns on product quality. In *null*, page 1031. IEEE, 2003.

[10] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.

[11] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, pages 1–20, 2011.

[12] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, 2002.

[13] Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013.

[14] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[15] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, pages 2–23, 2005.

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[17] 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*, pages 209–224, 2008.

[18] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Testing concurrent object-oriented systems with Spec Explorer. In *Proceedings of the 2005 International Conference on Formal Methods*, pages 542–547, 2005.

[19] M. Chabbi. Efficient taint analysis using multicore machines. Master's thesis, University of Arizona, 2007.

[20] Sreejit Chakravarty, YiShing Chang, Hiep Hoang, Sridhar Jayaraman, Silvio Picano, Cheryl Prunty, Eric W Savage, Rehan Sheikh, Eric N Tran, and Khen Wee. Experimental evaluation of bridge patterns for a high performance microprocessor. In *null*, pages 337–342. IEEE, 2005.

[21] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing: Adaptive random testing by exclusion. *International Journal of Software Engineering and Knowledge Engineering*, 16(04):553–584, 2006.

[22] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[23] T.Y. Chen, R. Merkel, P. K. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Proceedings of the Fourth International Conference on Quality Software*, pages 79–86, 2004.

[24] S.R. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[25] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, pages 71–80, 2008.

[26] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[27] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[28] James Clause and Alessandro Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 249–260, 2009.

[29] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[30] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[31] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, 2003.

[32] Thelma Elita Colanzi, Wesley Klewerton Guez Assunção, Silvia Regina Vergilio, and Aurora Pozo. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In *Proceedings of the Third International Conference on Search Based Software Engineering*, pages 188–203, 2011.

[33] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[34] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *Proceedings of the 2014 International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[35] 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.

[36] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Automated unique input output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[37] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, 1993.

[38] Michael Dyer. *The cleanroom approach to quality software development.* John Wiley & Sons, Inc., 1992.

[39] Jr. Ernest C. Davenport and Nader A. El-Sanhurry. Phi/phimax: Review and synthesis. *Educational and Psychological Measurement*, 51(4):821–828, 1991. doi: 10.1177/001316449105100403.

[40] Sandro Fouché, Myra B. Cohen, and Adam Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 177–188, 2009.

[41] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 147–158, 2010.

[42] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, 2012.

[43] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, 1995.

[44] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, 2007.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, number 6, pages 213–223, 2005.

[46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security*, 2008.

[47] W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: A framework for encoding and composing partial behaviors. *International Journal of Software Engineering and Knowledge Engineering*, 16(05):705–726, 2006.

[48] Wolfgang Grieskamp and Nicolas Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 59–66, 2006.

[49] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 223–233. ACM, 2015.

[50] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[51] M. Harman. Software engineering meets evolutionary computation. *IEEE Computer*, 44(10):31–39, 2011.

[52] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36(2):226–247, 2010.

[53] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[54] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, 2009.

[55] Ahmed E Hassan and Richard C Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, 2005.

[56] Robert M. Hierons. Reaching and distinguishing states of distributed systems. *SIAM Journal of Computing*, 39(8):3480–3500, 2010.

[57] W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977.

[58] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, September 1975.

[59] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 621–631, 2014.

[60] Chen Huo and James Clause. Interpreting coverage information using direct and indirect coverage. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–244, 2016.

[61] Jiale Huo and Alexandre Petrenko. Covering transitions of concurrent systems through queues. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 335–345, 2005.

[62] Capers Jones. The pragmatics of software process improvements. *Software Process Newsletter*, 3(5), 1996.

[63] 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 *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[64] Taghi M Khoshgoftaar and John C Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[65] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007.

[66] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[67] Ken Koster. A state coverage tool for JUnit. In *Companion of the 30th International Conference on Software Engineering*, pages 965–966, 2008.

[68] Kenneth Koster and David C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 541–544, 2007.

[69] V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov. The unitesk approach to designing test suites. *Programming and Computer Software*, 29(6):310–322, 2003.

[70] Janusz W Laski and Bogdan Korel. A data flow oriented program testing strategy. *Software Engineering, IEEE Transactions on*, (3):347–354, 1983.

[71] D. Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[72] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 21–40, 2002.

[73] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG-IPOG-D: Efficient test generation for multi-way combinatorial testing. *Software Testing, Verification & Reliability*, 18(3):125–148, 2008.

[74] Nan Li and Jeff Offutt. An empirical analysis of test oracle strategies for model-based testing. In *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 363–372, 2014.

[75] Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. A divergence-oriented approach to adaptive random testing of Java programs. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 221–232, 2009.

[76] Huai Liu, Xiaodong Xie, Jing Yang, Yansheng Lu, and Tsong Yueh Chen. Adaptive random testing through test profiles. *Software: Practice and Experience*, 41 (10):1131–1154, 2011.

[77] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 416–426, 2007.

[78] Rupak Majumdar and Ru-Gang Xu. Reducing test inputs using information partitions. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 555–569, 2009.

[79] Elisabet J Mccluskey and Chao-Wen Tseng. Stuck-fault tests vs. actual defects. In *Test Conference, 2000. Proceedings. International*, pages 336–342. IEEE, 2000.

[80] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38(2):453–477, 2012.

[81] Ali Mesbah and Mukul R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570, 2011.

[82] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[83] Keith W Miller, Larry J Morell, Robert E Noonan, Stephen K Park, David M Nicol, Branson W Murrill, and Jeffrey M Voas. Estimating the probability of failure when testing reveals no failures. *Software Engineering, IEEE Transactions on*, 18(1):33–43, 1992.

[84] Breno Miranda. A proposal for revisiting coverage testing metrics. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 899–902, 2014.

[85] John C Munson and Taghi M Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[86] Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pages 521–528, 2009.

[87] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.

[88] Simeon C Ntafos. On required element testing. *Software Engineering, IEEE Transactions on*, (6):795–803, 1984.

[89] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, pages 416–429, 1999.

[90] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, pages 37:1–37:11, 2011.

[91] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[92] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[93] Wangqi Qiu, Jing Wang, DMH Walker, Divya Reddy, Xiang Lu, Zhuo Li, Weiping Shi, and Hari Balachandran. K longest paths per gate (klpg) test generation for

scan-based sequential circuits. In *Test Conference, 2004. Proceedings. ITC 2004. International*, pages 223–231. IEEE, 2004.

[94] C.V. Ramamoorthy, Siu-Bun F Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2 (4):293–300, 1976.

[95] Sandra Rapps and Elaine J Weyuker. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, (4):367–375, 1985.

[96] Sanjai Rayadurgam and Mats PE Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[97] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 23–32, 2011.

[98] P. Saxena, R Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 74–83, 2008.

[99] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

[100] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[101] A. Shahbazi, A.F. Tappenden, and J. Miller. Centroidal voronoi tessellations—a new approach to random testing. *IEEE Transactions on Software Engineering*, 39(2):163–183, 2013.

[102] K. Shrestha and M.J. Rutherford. An empirical evaluation of assertions as oracles. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation*, pages 110–119, 2011.

[103] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, pages 117–126, 2007.

[104] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Better testing through oracle selection (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 892–895, 2011.

[105] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 870–880, 2012.

[106] K. Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410, 2008.

[107] A.F. Tappenden and J. Miller. A novel evolutionary approach for adaptive random testing. *IEEE Transactions on Reliability*, 58(4):619–633, 2009.

[108] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, pages 134–153, 2008.

[109] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[110] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[111] Jan Tretmans. Formal methods and testing. chapter Model Based Testing with Labelled Transition Systems, pages 1–38. 2008.

[112] Chao-Wen Tseng, Subhasish Mitra, Scott Davidson, and Edward J McCluskey. An evaluation of pseudo random testing for detecting real defects. In *VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001*, pages 404–409. IEEE, 2001.

[113] Yu-Wen Tung and W.S. Aldiwan. Automating test case generation for the new generation mission software system. In *Proceedings of the IEEE Aerospace Conference*, pages 431–437, 2000.

[114] Dries Vanoverberghe, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, pages 542–553, 2012.

[115] Willem Visser, Corina S Pâsâreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[116] Jeffrey M Voas and Keith W Miller. Software testability: The new verification. *IEEE software*, (3):17–28, 1995.

[117] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time-aware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[118] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatically generating natural language descriptions for object-related statement sequences. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 205–216, Feb 2017.

[119] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation*, volume 3103 of *Lecture Notes in Computer Science*, pages 1400–1412. 2004.

[120] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3): 275–298, 1998.

[121] Joep Weijers. Extending project lombok to improve junit tests. Master's Thesis, 2012.

[122] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing-EDCC 5*, pages 281–292. 2005.

[123] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1), February 2007.

[124] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.

[125] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of the ACM*

SIGSOFT International Symposium on Software Testing and Analysis, pages 45–54, 2004.

[126] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 201–212, 2009.

[127] Tingting Yu, W. Srisa-an, and G. Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering*, pages 11–20, 2013.

[128] Yuan Zhan and John A. Clark. Search-based mutation testing for Simulink models. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1061–1068.

[129] Benwen Zhang, Emily Hill, and James Clause. Automatically generating test templates from test names (n). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 506–511, 2015.

[130] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636, 2016.

[131] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanc Muslu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 14th International Symposium on Software Testing and Analysis*, pages 385–396, 2014.