Investigating Faults Missed by Test Suites Achieving High Code Coverage

Amanda Schwartz, Daniel Puckett

University of South Carolina Upstate

aschwar2@uscupstate.edu,dpuckett@email.uscupstate.edu

Ying Meng, Gregory Gay

University of South Carolina

ymeng@email.sc.edu, greg@greggay.com

Abstract

Code coverage criteria are commonly used to determine the adequacy of a test suite. However, studies investigating code coverage and fault-finding capabilities have mixed results. Some studies have shown that creating test suites to satisfy coverage criteria has a positive effect on finding faults, while other studies do not. In order to improve the fault-finding capabilities of test suites, it is essential to understand what is causing these mixed results. In this study, we investigated one possible source of variation in the results observed: fault type. Specifically, we studied 45 different types of faults and evaluated how effectively human-created test suites with high coverage percentages were able to detect each type of fault. Our results showed, with statistical significance, there were specific types of faults found less frequently than others. However, improvements in the formulation and selection of test oracles could overcome these weaknesses. Based on our results and the types of faults that were missed, we suggest focusing on the strength of test oracles along with code coverage to improve the effectiveness of test suites.

Keywords: Code Coverage, Automated Testing, Software Testing, Test Suite Effectiveness

1. Introduction

In order to ensure software quality, it is essential that the software is tested thoroughly. However, what exactly constitutes *thorough* testing can be subjective. As developers lack knowledge of the faults that may reside in their systems, guidance and a means of judging test suite *adequacy* is required. Currently, one of the most popular ways to evaluate the adequacy of a test suite is through the use of code coverage criteria.

Code coverage criteria evaluate test suites by examining how well they cover structural elements such as functions, statements, branches, and/or conditions of a software system [1]. Each criterion establishes a set of test obligations over the class-under-test (CUT) that must be fulfilled in order to satisfy the criterion. Coverage criteria are commonly used in both academic research and industry, as they are easy to understand, establish clear guidelines and stopping conditions for testing, and are well-supported across a variety of programming languages [2]. Consequently, the confidence in code coverage being a proper method of evaluating test suites has become high in the software testing community. In fact, the confidence is so high that in some domains, such as avionics, evaluating test suites through the use of code coverage is legally required [3, 4]. Many research studies also validate proposed techniques by their ability to achieve some level of code coverage (e.g. [5, 6]).

Contrary to the widespread use and acceptance of code coverage being an adequate measure of test suite effectiveness, studies investigating the relationship between code coverage and fault-finding capabilities do not consistently support this. Some studies have shown that generating test suites to satisfy code coverage criteria has a positive effect on finding faults (e.g. [7, 8, 9]), while other studies do not (e.g. [10, 11]). To better evaluate whether code coverage is a proper method of evaluating test suites, it is important to understand why there are such differences in these findings.

In order to accept high code coverage as an indicator for a test suite's ability to find faults, there should be consistent evidence that test suites with high

code coverage are capable of finding more faults than test suites with lower code coverage. However, since the research does not always support this, it is important to investigate whether there are particular factors that affect the ability of a test suite achieving high code coverage to find faults. Unfortunately, very little work has been done in this area. Few studies were found [12, 4, 13] that identify factors that influence the relationship between code coverage and fault detection. These studies provide some insight—particularly around the influence of program structure—but cannot completely explain the different findings in the studies investigating code coverage and fault detection. More research needs to be conducted to investigate this important issue.

In our previous work [14], we began to investigate the impact particular *fault types* had on the relationship between code coverage and fault finding effectiveness, as modeled through the use of mutation testing—the seeding of synthetic faults into the CUT. Specifically, we were interested in whether there were particular fault types that went undetected more frequently than other fault types when programs are evaluated by test suites that achieve high code coverage. Our research showed that the rate of fault detection varied significantly according to fault type. We also noticed that there were certain types of faults consistently found less frequently than others. These were interesting findings that could inspire future research on why these particular fault types were found less frequently. However, this study was limited in two ways. First, only class-level mutation operators were considered. Second, there were many mutation operators that did not produce enough mutants to have enough data to perform a statistical analysis or form any solid conclusions. We address these limitations in this paper. Specifically, this paper makes the following contributions:

- The paper is extended to consider 19 Traditional Mutant Operators in addition to the original 26 Class Level Mutation Operators considered in our previous work [14].
- An additional 25,100 Class-Level mutants were created to supplement the 15,834 Class-Level mutants created in our previous work, for a total of

40,934 Class-Level mutants.

- A total of 122,985 Traditional mutants were created and analyzed.
- Statistical tests were performed and presented to determine whether there is a statistical significance to the faults that go undetected more frequently than other faults.
- A discussion of the fault types identified as outliers by the statistical tests is included.
- A suggestion, based on the nature of the faults identified as outliers, on how to improve test suites and test suite evaluation techniques is provided.

Our results identified that two types of mutants were found disproportionately often—AORB (Arithmetic Operator Replacement) and ROR (Relational Operator Replacement). However, four types of mutants were found less often than expected: AOIS (Arithmetic Operator Insertion), PCI (Type Case Operator Insertion), EAM (Accessor Method Change), and AODU (Arithmetic Operator Deletion). Test oracles that more thoroughly inspect internal state would aid in revealing such faults. Ultimately, code coverage alone does not ensure that faults are triggered and detected, and the selection of input and oracle have a dual influence on the effectiveness of a test suite. More attention should be given to the thoroughness of the selected oracle, and to the variables that are monitored and checked by the oracle.

The rest of the paper is organized as follows. Section 2 presents background information and related work. Section 3 explains our experimental procedures. Section 4 presents the results of our experiment. A discussion of our results is presented in Section 5. And finally, conclusions are discussed in Section 6.

2. Background and Related Work

Software testing is extremely important to the development process as the means of ensuring that software has the correct functionality and is not defec-

tive. However, software testing can be very time consuming and costly. Therefore, a significant amount of time and effort has been spent to identify ways to reduce the cost of software testing. Much of this attention has been spent researching automated software testing procedures. As a result, many different testing platforms and methods have been proposed, developed, and evaluated (e.g.[15, 16, 17]) and a great deal of progress has been made on reducing the time necessary for software testing. By reducing time, the hope is it will also reduce cost. However, the cost will only be reduced if testing continues to be effective at finding faults. Missed faults are extremely costly, and any reduction in cost resulting from the decreased time would be lost with the increase of costs associated with missed faults. Therefore, automated testing methods need to be evaluated to be sure they are effective at finding faults.

The most commonly used metric to evaluate automated test suites is to use some form of code coverage criteria. Coverage criteria reports a percentage according to how much source code is executed by the test suites. The exact calculation depends on the coverage method used. For example, line coverage is a very simple coverage metric that simply reports the percentage of lines of code that are covered by the test suite. Another popular coverage metric, branch coverage, adds an additional requirement that at each conditional both the true path and the false path will be executed.

Coverage criteria has become widely accepted in the software testing community as an adequate measure of test suite effectiveness [2]. It is used to validate new automated testing methods (e.g. [18, 19, 20]), used to compare testing methods based on level of coverage (e.g. [21, 22, 23]), and is used in many domains to determine whether a test suite as adequate. In fact, in some safety cricital domains, such as avionics, code coverage is legally required to determine the adequacy of a test suite [3].

Since coverage criteria is frequently the standard for many in terms of evaluating test suites, it is important to make sure code coverage *actually* is a good indicator of test suite effectiveness. Some recent research has been conducted to evaluate whether increasing code coverage also increases a test suite's ability

to find faults. The results of this research has been mixed. Some studies show achieving high code coverage is a good indicator for fault-finding capabilities, while other studies do not.

Studies which provide support for coverage criteria being an adequate measure of test suite effectiveness show a correlation between code coverage and fault-finding capabilities. For example, early work by Frankl and Weiss [24] shows a correlation between test suite effectiveness and all-use and decision coverage criteria. Cai and Lyu [25] found a moderate correlation between fault-finding capabilities and four different code coverage criteria. Frate et. al [26] report a study which finds a higher correlation between test suite effectiveness and block coverage than there is between test suite effectiveness and test suite size. Gligoric et. al [27] studied a total of 26 programs and found that test suite effectiveness was correlated with coverage, and reported branch coverage as performing the best. A recent study by Kochhar et. al [28] evaluated two industrial programs and found a correlation between code coverage and faults detected. In our past work examining the factors that indicated a high likelihood of fault detection, we found that high levels of code coverage had a stronger correlation to the likelihood of fault detection than the majority of the other measured factors [9]. However, we also found that coverage alone was not enough to ensure fault detection.

Even though there are a number of studies that show a correlation between code coverage and fault-finding effectiveness, there are also many studies which do not. In previous work, we reported that satisfying code coverage alone was a poor indication of test suite effectiveness when suites are generated specifically to achieve coverage [10, 29]. We studied the fault-finding effectiveness of automatically generated test suites that satisfied five code coverage criteria (branch, decision, condition, MC/DC, and Observable MC/DC) and compared them to randomly generated test suites of the same size for five different production avionics systems. We found that, for most criteria, test suites automatically generated to achieve coverage performed *significantly worse* than random test suites of equal size which did not work to achieve high coverage. We did find

that coverage had some utility as a *stopping criterion*, however. Randomly-generated test suites that used coverage as a stopping criterion outperformed equally-sized suites generated purely randomly. The results of this study confirm that satisfying code coverage criteria has the potential to find faults, but there are factors (i.e. program structure, types of faults, and choice of variables monitored by the test oracle) that affect a test suite's ability to find them even when high code coverage is achieved.

Another group of studies have investigated an additional factor: the size of the test suite, and how it interacts with both code coverage and suite efficacy. Inozemtseva and Holmes [11] performed an empirical study investigating the relationship between coverage and fault-finding capabilities and observed the difference between the results when size was controlled and when it was not controlled. They noticed that when size was not controlled for a test suite there was a moderate to high correlation between coverage and fault-finding effectiveness. However, when size was controlled they saw the correlation drop significantly. They suggest that the effectiveness is not correlated with coverage, but instead with the size of the test suite. A similar study by Namin and Andres [8] studied the relationship between size, code coverage, and fault-finding effectiveness. They found that coverage is only *sometimes* correlated with effectiveness when the size of the test suite is controlled. They also noted that no linear relationship existed between the three variables. This directly contradicts the findings of Frate et. al [26] mentioned previously that found a higher correlation between test suite effectiveness and block coverage than between test suite effectiveness and test suite size.

These opposing results make this an interesting problem to solve. They indicate there are factors that are not yet discovered which prevent test suites from finding faults even when high code coverage is achieved. Unfortunately, very little work has been done to understand why these studies report such different findings. We [12, 4] previously investigated the effect program structure had on the ability of test suites satisfying the MC/DC coverage criterion to find faults. Specifically, we compared the difference in test obligations and

the resulting test suites when implementation is varied between a small number of complex, inlined expressions and a larger number of simple expressions. We found that the MC/DC criterion is highly sensitive to the different implementations, and reported that MC/DC satisfaction requires significantly more test cases—and generally, more complex test cases—on inlined implementations than it did on the non-inlined version of the same implementation. The use of an inlined implementation produces robust test suites that are significantly more adept at detecting faults. More recently, Zhang and Mesbah investigated the influence test assertions have on code coverage and test suite effectiveness [13]. They suggested *assertions* are the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness. The results of their study provide empirical evidence that assertion quantity and assertion coverage are strongly correlated with a test suite's effectiveness and the correlation between statement coverage and test suite effectiveness decreases dramatically when assertion coverage is controlled.

These studies identify two specific factors apart from code coverage that influence test suite effectiveness and provide empirical evidence that code coverage alone is not always a good indicator of test suite effectiveness. Additional factors which affect the effectiveness and/or additional test suite evaluation metrics should be proposed and studied. Perez et. al [30] proposed a test evaluation metric which incorporates not only *if* a component is covered by a test suite, but also *how* it is covered.

Our paper advances this area of research by investigating the impact fault type has on the relationship between code coverage and test suite effectiveness. Our study reveals specific fault types that frequently go undetected even though a test suite achieves high code coverage. Based on the nature of these faults we suggest an area of future research that could improve a test suite's ability to find these types of faults and increase the overall effectiveness of a test suite.

3. Study

In the preceding sections, the variability in studies investigating the fault-finding effectiveness of test suites meeting high levels of code coverage was discussed. In this work we conduct an empirical study to investigate whether fault type could be a contributing factor to the variability shown in the studies. Specifically, we answer the following research questions:

1. Are there particular fault types that go undetected more frequently by developer-written test suites achieving high code coverage?
2. Are there particular fault types that are detected with disproportionate frequency by developer-written test suites achieving high code coverage?

To answer these questions, we have performed the following steps:

1. Selected programs with large code bases and developer-written test suites achieving at least 80% code coverage (Section 3.1).
2. Generated mutants—synthetic faults—for all classes within each program (Section 3.2).
3. Executed the developer-written test suite against each mutant (Section 3.3).
4. Collected data on fault detection (Section 3.4).

In the following sections, we explain the procedures we followed for our experiment.

3.1. Selection of Object Programs

To begin the experiment, we needed to select our object programs. We selected these programs based on the following specific criteria:

1. A developer-written test suite must exist for the program.
2. This test suite must achieve at least 80% statement coverage over the program code.
3. The program must have at least 10,000 lines of code.

Table 1: Experiment Objects and Associated Data

	Commons Compress	Joda Time	Commons Lang	Commons Math	JSQL Parser
Thousand Lines of Code (KLOC)	11	14	13	49	10
Number of Test Cases	556	2157	3526	6248	315
Statement Coverage (%)	83	89	93	89	82
Branch Coverage (%)	81	81	90	85	76
Number of Mutation Faults	24,194	43,660	14,917	42,936	3,510
	Commons CLI	JSoup	Commons CSV	Commons Codec	Closure
Thousand Lines of Code (KLOC)	7	19	6	20	356
Number of Test Cases	408	648	303	887	13,341
Statement Coverage (%)	96	81	94	93	87
Branch Coverage (%)	94	77	89	90	80
Number of Mutation Faults	1,462	3,738	1,083	4,354	12,202

Table 1 lists the objects of analysis along with the following information for each program: the number of thousands of lines of code (KLOC), the number of test cases, line coverage percentage, branch coverage percentage, and number of mutation faults. Number of lines of code are counted without including commented and whitespace lines. We ran both Cobertura [1] and ECLEmma [2] coverage tools to obtain coverage information. For each of the programs, these tools reported within less than a percent of each other. The numbers reported in Table 1 are therefore an average of those numbers rounded to the nearest percent.

The programs we chose all have existing JUnit test suites. The JUnit test suites are developer-written (as opposed to test suits that are generated by automated test generation tools), which reflects common practice in industry. This also means the tests are most closely tailored for each specific program. We are focused on developer-written suites, as automatically-generated suites typically are very different than developer-written suites [31, 32]. Automatically-generated suites often use shorter sequences of test steps [10], choose inputs that do not resemble those chosen by humans [31], follow sequences of events that differ from those chosen by humans [33], and may be verified with gen-

[1] cobertura.github.io/cobertura

[2] www.eclemma.org

erated test oracles that differ from those created by humans [33]. Therefore, focusing on automatically-generated test suites or mixing test creation methods would introduce a risk of conflated results. Instead, we focus in this study on developer-written tests and the typical strengths and weaknesses of such suites.

We wanted programs with existing test suites that satisfy high code coverage percentages. We required programs with high code coverage because many studies that found correlation between code coverage and fault-finding capabilities, only found that correlation when code coverage was high (e.g. [34, 24, 35, 36]). By choosing programs with high code coverage there is a better chance of finding the faults (as indicated by studies revealing there to be a correlation between high code coverage and fault detection). We wanted to have the highest chance of finding the fault so we could identify faults that were still going undetected. Faults going undetected in this situation could explain some variance in the findings of studies investigating this relationship. Based on the results of studies which found a correlation between coverage and fault detection at high coverage rates (e.g. [34, 35]), we chose 80% as the minimum target [3].

In order to provide an ample opportunity for a large number of faults to be inserted, all of our programs have over 6,000 non-commented, non-whitespace lines of code. This was necessary to collect enough fault data for statistical analysis. We chose ten medium-sized Java programs that met the criteria we were searching for. The programs range from 6,000 to 356,000 lines of code. Joda Time [4] is an opensource replacement for the date and time classes in Java versions previous to Java 8. The Apache Commons Math library [5] is an opensource Mathematics library containing Math and Statistics components.

[3] JSQL Parser and JSoup had statement coverage above 80%, but the branch coverage for these programs fell just below. Because the statement coverage met the threshhold and the branch coverage was very close, we decided to include these two programs as well. All other programs met or exceeded 80% for both statement and branch coverage.

[4] http://joda-time.sourceforge.net/

[5] http://commons.apache.org/proper/commons-math/

The Apache Commons Compress package [6] is an opensource API that provides Java with the ability to work with file compression. The Apache Commons Lang API [7] is an opensource library that provides helper utilities for the java.lang API. JSQLParser [8] is an opensource API that parses SQL statements into a hierarchy of Java classes. The Apache Commons CLI API[9] provides support for parsing command-line options passed to programs. The Jsoup HTML Parser allows Java programs to work with real-world HTML[10]. The Apache Commons CSV API reads and writes files in variations of the comma-separated value (CSV) format[11]. The Apache Commons Codec API provides implementations of common encoders and decoders[12]. Finally, the Closure Compiler is a JavaScript checker and optimizer, written in Java[13].

3.2. Mutant Creation

To investigate whether certain types of faults consistently go undetected when tested with test suites satisfying high code coverage criteria, we needed to have programs with faults. More specifically, we needed to know the location and nature of each fault in order to determine whether the test suite was able to catch it, and which types of faults were getting caught less frequently than others.

Mutation testing is used in software testing to take the original program and modify it in small ways. This can be helpful for many different reasons. One main use of mutation testing is to evaluate the quality of software tests. First, small variations of the original program are created by inserting a known fault into the program. Then, the test suite will be run on the variation to see if it is able to catch that fault. This process provides an indicator of how effective the

[6] http://commons.apache.org/proper/commons-compress/
[7] http://commons.apache.org/proper/commons-lang/
[8] http://jsqlparser.sourceforge.net/
[9] https://commons.apache.org/proper/commons-cli/
[10] https://jsoup.org/
[11] https://commons.apache.org/proper/commons-csv/
[12] https://commons.apache.org/proper/commons-codec/
[13] https://github.com/google/closure-compiler

test suite is at finding faults.

Mutation testing works by using *mutation operators* to create the small variations of the programs. Each mutation operator provides the mechanism to insert one specific type of fault. For example, the JSD (Static Modifier Deletion) operator changes class variables to instance variables by removing the static modifier. An example mutant is shown below.

Original:	Mutated:
`public static int x = 100;`	`public int x = 100;`

The previous example was just one example of a simple mutation operator. Many mutation operators have been proposed and studied in the literature (e.g. [37, 38]). Mutation operators are built with the goal of emulating common programming mistakes. Each small variation of the program created through the use of mutation operators are called *mutants*. Mutants can be created by making a single change to the program, or by making more than one change to the program. When mutants are created with multiple changes, the faults may interfere with each other and cause different results than if each fault was inserted individually in a mutant. For example, a test case which would fail with the insertion of one fault, may no longer fail when another fault is added. Similarly, a test case may not fail with a single fault, but fail with additional faults. This problem of fault interference and interaction has been studied and discussed in recent studies [39, 40]. To eliminate the threat of fault interference and to get accurate results according to each individual fault type, we created each mutant with a single fault.

In our experiment, we used mutation testing because of its ability to provide programs with a large number of known, categorized faults. Our experiment is concerned with the frequency at which faults are found, as well as the nature of these faults. Therefore, we used mutation testing software to create mutants with known fault location and type. Then, we could run the test suites over the

mutated programs and determine which ones were caught by the test suite and which ones were not. In recent research, there has been some concern whether conclusions drawn from studies using faults created through mutation operators can be generalized for real faults. In our study the concern would be whether the ability to find the seeded faults would be representative of a test suite's ability to find real faults. There is research to support that mutant detection is strongly correlated with real fault detection [41, 42]. Further, our study isn't focused solely on the exact mutant operator being missed, but instead on the nature of that type of fault. We are using the results of the study to be able to expand the knowledge base and answer questions such as: What problems do faults being missed cause? How can test suites be improved to find faults which cause that type of problem?

Our study uses Java programs, so we needed a mutation tool to be able to create mutants for Java programs. There are a few tools available to do this. The most popular mutation tools for Java include Pit [14], Major [43], and MuJava [44]. Of the three, MuJava provided the most complete set of mutation operators, providing both traditional and class-level mutation operators. Therefore, we used MuJava to generate the mutants in this study. MuJava provides 19 traditional mutation operators and 28 class-level mutation operators [45]. There were two class-level mutation operators (EOA and IHD) that MuJava was unable to generate in any of our programs selected for this study, therefore we only used the remaining 26 class-level mutation operators in this study. The number of mutants created for each program is included in the last row in Table 1. A description of the traditional operators is provided in Table 2 and a description of the class-level mutation operators is provided in Table 3.

There are some mutant operators that would appear to be very specific to Java programs, or at least to object-oriented languages. For example, the IOD (Overriding Method Deletion) operator deletes an entire declaration of an overriding method in a subclass, so that it will use the parent's version of

[14] http://pitest.org/

Table 2: Traditional Mutation Operators and Descriptions

Mutation Operators	Description
AORB (Arithmetic Operator Replacement)	Replace basic binary arithmetic operators with other binary arithmetic operators.
AORS (Arithmetic Operator Replacement)	Replace short-cut arithmetic operators with other unary arithmetic operators.
AOIU (Arithmetic Operator Insertion)	Insert basic unary arithmetic operators.
AOIS (Arithmetic Operator Insertion)	Insert short-cut arithmetic operators.
AODU (Arithmetic Operator Deletion)	Delete basic unary arithmetic operators.
AODS (Arithmetic Operator Deletion)	Delete short-cut arithmetic operators.
ROR (Relational Operator Replacement)	Replace relational operators with other relational operators, and replace the entire predicate with true and false.
COR (Conditional Operator Replacement)	Replace binary conditional operators with other binary conditional operators.
COD (Conditional Operator Deletion)	Delete unary conditional operators.
COI (Conditional Operator Insertion)	Insert unary conditional operators.
SOR (Shift Operator Replacement)	Replace shift operators with other shift operators.
LOR (Logical Operator Replacement)	Replace binary logical operators with other binary logical operators.
LOI (Logical Operator Insertion)	Insert unary logical operator.
LOD (Logical Operator Delete)	Delete unary logical operator.
ASRS (Short-Cut Assignment Operator Replacement)	Replace short-cut assignment operators with other short-cut operators of the same kind.
SDL (Statement Deletion)	SDL deletes each executable statement by commenting them out. It does not delete declarations.
VDL (Variable Deletion)	All occurrences of variable references are deleted from every expression. When needed to preserve compilation, operators are also deleted.
CDL (Constant Deletion)	All occurrences of constant references are deleted from every expression. When needed to preserve compilation, operators are also deleted.
ODL (Operator Deletion)	Each arithmetic, relational, logical, bitwise, and shift operator is deleted from expressions and assignment operators. When removed from assignment operators, a plain assignment is left.

the method. However, there is still useful information that can be gained by understanding a test suite's ability to catch this type of error. This type of error would be similar to just calling the wrong method. It would be considered a control flow error, as it is executing the wrong area of code. This type of problem can be caused in languages that are not object-oriented as well. For example, in functional languages it is quite common to put a function call as a parameter in another function call (and possibly nested a few times). The function calls could be accidentally transposed, calling the wrong method. This problem would be similar to the type of problem caused by the IOD operator. This example illustrates how—even though our experiment uses Java programs and mutant operators built for Java—the knowledge gained from our study could be useful in other circumstances as well.

3.3. Execute Test Suites Over Mutants

After the programs were chosen and mutants were created for them, we executed the entire test suite for each program against every mutant. Although

Table 3: Class Level Mutation Operators and Descriptions

Mutation Operators	Description
IHD (Hiding Variable Deletion)	The IHD operator will delete a variable in a subclass that has the same name and type as a variable in the parent class.
IHI (Hiding Variable Insertion)	The IHI operator inserts a variable of the same name as a variable in the parent scope, so as to "hide" the variable in the parent scope.
IOD (Overriding Method Deletion)	The IOD operator deletes an entire declaration of an overriding method in a subclass so it will use the parent's version.
IOP (Overriding method calling position change)	When a child class makes a call to a method it overrides in the parent class, the IOP operator will move that call to a different location in the method.
IOR (Overridden method rename)	The IOR operator renames the parent's version of an overridden method.
ISI (super Keyword Insertion)	The ISI operator inserts the super keyword so that a reference to a variable or method in a child class uses the parent variable or method.
ISD (super Keyword Deletion)	The ISD operator deletes occurrences of the super keyword so that a reference to a variable or method is no longer to the parent class' variable or method.
IPC (Explicit Call of a Parent's Constructor Deletion)	The IPC operator deletes super constructor calls, causing the possibility for a child object to not be initialized correctly.
PNC (New Method Call with Child Class Type)	The PNC operator causes an object reference to refer to an object of a different compatible type.
PMD (Member Variable Declaration with Parent Class Type)	The PMD operator changes the declared type of an object reference to the parent of the original declared type.
PPD (Parameter Variable Declaration with Child Class Type)	The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables.
PCI (Type Cast Operator Insertion)	The PCI operator changes the actual type of an object reference to the parent or child of the original declared type.
PCC (Cast Type Change)	The PCC operator changes the type that a variable is cast into.
PCD (Type Cast Operator Deletion)	The PCD operator deletes a type casting operator.
PRV (Reference Assignment with Other Compatible Types)	The PRV operator changes operands of a reference assignment to be assigned to objects of subclasses.
OMR (Overloading Method Contents Change)	The OMR operator replaces the body of a method with the body of another method that has the same name.
OMD (Overloading Method Deletion)	The OMD operator deletes overloading method declarations.
JTI (this Keyword Insertion)	The JTI operator inserts the keyword this.
JTD (this Keyword Deletion)	The JTD operator deletes uses of the keyword this.
JSI (static Modifier Insertion)	The JSI operator adds the static modifier to change instance variables to class variables.
JSD (static Modifier Deletion)	The JSD operator removes the static modifier to change class variables to instance variables.
JID (Member Variable Initialization Deletion)	The JID operator removes the initialization of member variables in the variable declaration.
JDC (Java-supported Default Constructor Deletion)	The JDC operator forces Java to create a default constructor by deleting the implemented default constructor.
EOA (Reference Assignment and Content Assignment Replacement)	The EOA operator replaces an assignment of an object reference with a copy of the object, using the Java clone() method.
EOC (Reference Comparison and Content Comparison Replacement)	The EOC operator changes an object reference check to object content comparison check through the use of Java's equal() method.
EAM (Accessor Method Change)	The EAM operator changes an accessor method name for other compatible accessor method names (where compatible means they have the same signature).
EMM (Modifier Method Change)	The EMM does the same as EAM except it works with modifier methods instead of accessor methods.
OAN (Argument Number Change)	The OAN operator changes the number of arguments in the method invocations, but only if there is an overloading method that can accept the new argument list.

recent studies [11, 8, 10] have reported size to be a factor in the effectiveness of test suites at finding faults, we did not try to limit the size of the test suite. We ran all tests across all mutants. If the test case found the seeded fault in the mutant, it is said to have *killed the mutant*. By running all tests (instead of a subset of tests that met the same coverage percentage), the chances of killing the mutant were higher. The goal of our study was to gain knowledge about particular fault types that go undetected most frequently. By revealing the faults that went undetected when all tests were executed, we successfully identify the most troublesome fault types.

3.4. Collect and Analyze Data

Finally, after running the test suites across all mutants, we completed the data collection. To be able to answer our research question, we needed to collect the following data: the number of mutants created *for each mutant operator* and the number of these mutants killed by the test suite. Because of the large number of mutants investigated in this study, in order to collect this data, we wrote some custom scripts to inspect the log files generated from running the JUnit test suites. Using these scripts we were able to determine, for every mutant, whether the bug was found (aka mutant killed) or not found. Then we could use this information and quickly tally totals for each mutation operator for each program.

3.5. Threats to Validity

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

3.5.1. Construct Validity:

In our study, we measure fault-finding capabilities over seeded faults, rather than real faults. It is possible real faults could produce different results, even though recent studies [41, 42] show that the detection of mutation faults is strongly correlated with the detection of real faults. Motivationally, however,

we believe there is benefit in using mutations even if mutations do not perfectly map to real faults. Test suites that are effective at detecting a wide range of mutation types may be likely to be better at detecting real faults as well, simply because more effort went into creating a robuse set of test cases. Guarding against mutations is guarding against additional potential shortcomings. If we can understand where developer-written suites tend to fall short, we can advise programmers on their practices and the creators of automated generation tools on where they can improve their work.

Further, the faults used in this study were limited to what MuJava was able to generate. Not all programs have mutants for every available mutant operator, and some mutant operators (EOA and IHD) were not generated for any of the studied programs. These operators were excluded from our results.

Mutation testing may produce equivalent mutants—mutants that always return the correct result, and are indistinguishable from normal programs. It is possible that some of the undetected faults are equivalent mutants, and detecting equivalency is often an undecidable problem. Therefore, we cannot state with certainty which of the undetected mutants were equivalent. However, in the majority of cases, the mutation types that were detected less often were not types that seem to be prone to producing equivalent mutants.

There is also the risk that the mutants generated could have been generated in the areas of code that are not covered by any tests in the test suite. These mutants would be recorded as not being killed, but potentially could have been killed if they had been in an area covered by the test suite. Although this has the potential to skew the results, we believe it is unlikely because we have chosen such a high coverage level and have encluded enough mutants from enough different classes to overcome any potential clustering of mutants in uncovered code.

3.5.2. Internal Validity:

The accuracy of the results of our study are dependent upon the accuracy of the tools used. We used tools to seed faults and measure coverage percentages.

We did use multiple tools to measure coverage and found the results to be very similar (varying less than one percent) so we are confident these results are accurate. Also, when tallying the total results, we wrote scripts to sift through the log files of each mutant in order to determine whether the fault was found or not. Although we did manually verify many of the logs for each program, there is still a possibility for a very small margin of error. Still, sifting manually through over 163,919 log files would have likely created a much larger margin of error as that would introduce more human error.

3.5.3. External Validity:

There are three main threats to external validity in our study. First, the programs we considered were all Java programs. Our results may not extend to programs written in other languages. Even though we explained in Section 3.2 that information *may* be gained from mutant operators built for Java in other programs as well, we do not have empirical evidence to quantify the extent to which this is true. Second, although the programs we chose were considerably larger than many of the previous studies mentioned in the Section 2, they are not large compared to industry systems. Our results may not generalize to these size systems. Last, the mutants created in our study were created by inserting a single fault into each mutant. We did this to eliminate the problem of fault interference that can occur with multiple faults [39, 40]. Because of fault interaction and interference, the results could be much different if multiple faults were inserted into each mutant.

4. Data and Analysis

In this section we present the results of our experiment. First, we will discuss the results for the traditional mutant operators, then we will discuss the results for the class-level mutant operators. Finally, we present the overall results and analyze our statistical findings.

Table 4: Traditional Mutant Results by Program

	Commons Compress		**Joda Time**		**Commons Lang**		**Commons Math**		**JSQL Parser**	
Mutant Operators	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
AORB	487	54.21%	1183	41.67%	789	80.99%	7621	88.05%	25	68.00%
AORS	22	40.91%	134	8.21%	35	82.86%	347	16.43%	2	50.00%
AOIU	477	36.90%	1565	45.69%	619	77.54%	4094	69.32%	24	70.83%
AOIS	2370	34.51%	7852	22.82%	1865	66.49%	7626	72.21%	169	66.86%
AODU	8	37.50%	2241	2.05%	32	81.25%	382	64.14%	0	-
AODS	6	100.00%	1	100%	5	100.00%	68	41.18%	0	-
ROR	1830	43.50%	5333	27.68%	3479	73.93%	6269	71.93%	290	76.21%
COR	170	41.76%	231	60.17%	396	68.94%	382	85.34%	27	85.19%
COD	30	43.33%	18	77.78%	40	87.50%	163	66.87%	15	93.33%
COI	511	68.49%	1807	33.76%	942	93.31%	1492	70.84%	263	94.30%
SOR	26	100.00%	6	100.00%	0	-	64	18.75%	0	-
LOR	99	57.58%	24	25.00%	9	88.89%	267	31.84%	0	-
LOI	890	43.03%	2749	37.94%	1104	84.33%	3215	33.65%	48	75.00%
LOD	3	100.00%	4	0.00%	0	-	2	0.00%	0	-
ASRS	255	64.31%	144	18.75%	337	70.33%	784	84.44%	4	0.00%
SDL	1808	46.52%	6841	21.41%	2447	81.94%	4418	75.46%	1127	78.26%
VDL	262	45.04%	570	29.65%	256	74.22%	2880	82.92%	74	91.89%
CDL	219	31.96%	557	15.26%	161	62.11%	770	77.27%	104	87.50%
ODL	1036	35.91%	3993	13.00%	1161	68.99%	4803	88.24%	398	86.93%
Total	10509	43.20%	35253	24.44%	13677	76.41%	45647	74.01%	2570	80.82%
	Commons CLI		**JSoup**		**Commons CSV**		**Commons Codec**		**Closure**	
Mutant Operators	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
AORB	26	65.38%	109	99.08%	30	73.33%	198	87.88%	118	33.90%
AORS	1	100.00%	22	95.45%	6	83.33%	5	80.00%	13	46.15%
AOIU	38	73.68%	201	98.01%	62	88.71%	183	72.13%	139	43.88%
AOIS	92	68.48%	772	97.02%	154	91.56%	501	83.43%	499	26.85%
AODU	0	-	4	100.00%	0	-	0	-	7	0.00%
AODS	0	-	3	100.00%	0	-	5	100.00%	4	50%
ROR	278	71.58%	855	98.71%	142	80.99%	1071	79.55%	896	38.17%
COR	56	58.93%	134	94.78%	43	81.40%	135	82.22%	384	62.50%
COD	8	75.00%	22	100.00%	7	100%	24	87.5%	57	77.19%
COI	128	86.72%	199	100.00%	81	93.83%	299	92.64%	546	74.36%
SOR	0	-	0	-	0	-	4	100.00%	0	-
LOR	0	-	0	-	0	-	24	91.67%	14	64.29%
LOI	49	71.43%	295	98.31%	103	91.26%	311	78.78%	188	44.15%
LOD	0	-	0	-	0	-	0	-	0	-
ASRS	0	-	22	100.00%	4	0.00%	29	72.41%	9	11.11%
SDL	368	73.37%	554	95.49%	231	89.18%	898	76.39%	1645	58.78%
VDL	20	45.00%	47	100.00%	21	61.90%	82	71.95%	71	43.66%
CDL	31	19.35%	20	100.00%	18	38.89%	68	63.24%	74	35.14%
ODL	127	43.31%	268	97.76%	101	65.35%	375	70.67%	701	49.93%
Total	1222	68.17%	3527	97.65%	1003	83.95%	4212	79.27%	5365	51.11%

4.1. Traditional Mutants

MuJava provides 19 traditional mutant operators. A list of the 19 operators and a description of each is provided in Table 2. A total of 122,985 traditional mutants were created across the ten object programs. Table 4 provides a breakdown of the number of each individual traditional mutant that were created for each object program, as well as the percentage of those mutants that were killed by the test suite for that program.

The results show there is a wide variance in the kill rate between each mutant type for the traditional operators. Figure 1 provides a bar chart of the overall kill percentages (using the totals for all ten programs) for each traditional operator. There are two mutant operators with a kill rate of less than 25%: AODU (12%) and AORS (25%).

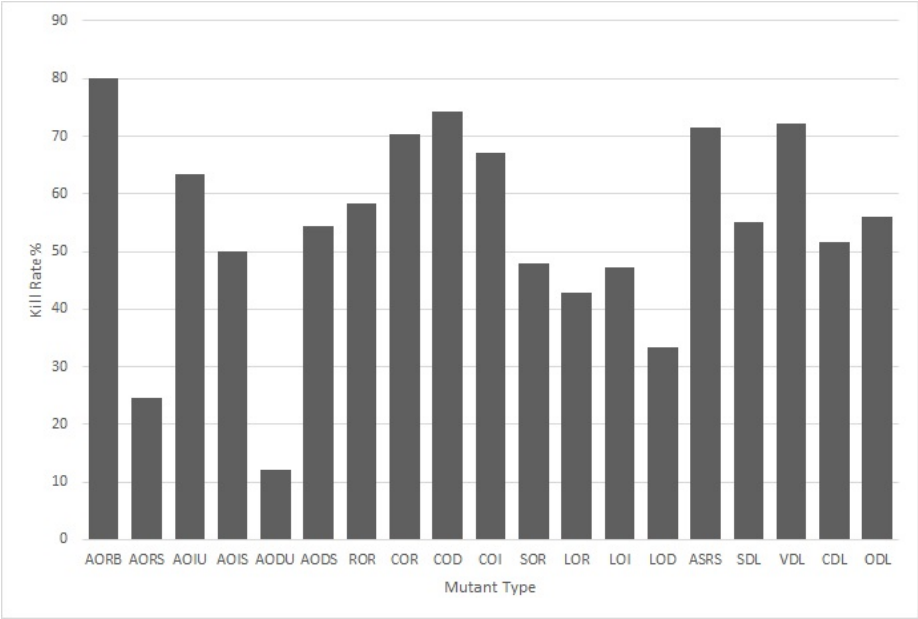

Figure 1: Traditional Mutants Kill Rates

4.2. Class-level Mutants

There are 26 class-level mutant operators that we considered in our experiment. A list of the operators and descriptions is provided in Table 3. A total of 40,934 class-level mutants were created across the ten object programs. Table 5 provides the number of mutants created for each operator for each program and the percentage of the mutants killed by the test suite for the program.

Like the traditional mutants, the kill rate for the class-level mutants vary greatly by mutant type. To quickly see which mutant types had the lowest kill rate overall, a bar graph is provided in Figure 2. The bar graph shows seven mutant operators with a kill rate of less than 20%: ISI (15%), JSD (10.09 %), PPD (0 %), IOR (6.63%), PMD (0 %), JID (17.41 %), and PNC (13.16 %).

4.3. Overall results

The bar graphs and tables provided in the previous sections show that there are some types of mutants found less frequently than others. The goal of this section is to see whether we can say *statistically* that these mutants are found

Table 5: Class Level Mutant Results by Program

	Commons Compress		**Joda Time**		**Commons Lang**		**Commons Math**		**JSQL Parser**	
Mutant Operators	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
EOC	3	0.00%	2	100%	6	33.33%	0	-	0	-
EMM	541	53.05%	28	85.71%	130	69.23%	65	27.69%	9	66.67%
OAN	5452	34.83%	3410	98.33%	146	89.73%	627	55.18%	2	0.00%
JDC	2	50.00%	1	100.00%	1	100.00%	5	80.00%	0	-
JSI	275	26.55%	17	29.41%	40	57.50%	217	19.35%	236	54.66%
ISI	20	0.00%	16	31.25%	0	-	24	16.67%	0	-
PCI	2635	0.00%	1363	47.25%	11	100.00%	317	64.35%	31	96.77%
PCC	11	0.00%	11	0.00%	0	-	13	92.31%	0	-
PCD	9	0.00%	8	37.50%	5	100.00%	9	88.89%	0	-
ISD	8	0.00%	8	12.50%	12	66.67%	11	63.64%	0	-
JSD	135	0.00%	43	20.93%	93	0.00%	259	9.65%	1	0.00%
EAM	702	25.07%	233	79.40%	260	84.62%	5524	33.76%	116	58.62%
PPD	0	-	0	-	0	-	4	0.00%	2	0.00%
IOR	84	0.00%	56	8.93%	0	-	24	25.00%	0	-
IOP	1	100.00%	0	-	0	-	8	50.00%	2	0.00%
PMD	19	0.00%	18	0.00%	2	0.00%	22	0.00%	0	-
PRV	409	14.18%	294	97.62%	80	97.50%	237	75.11%	55	85.45%
IOD	292	8.90%	196	39.80%	50	76.00%	225	17.78%	77	93.51%
JID	74	9.46%	8	87.50%	14	64.29%	23	8.7%	59	15.25%
JTI	104	48.08%	0	-	32	90.62%	128	26.56%	182	79.67%
JTD	53	67.92%	0	-	0	-	17	52.94%	161	83.85%
IPC	36	16.67%	10	30.00%	23	91.30%	78	62.82%	2	0.00%
OMD	28	3.57%	23	4.35%	13	84.62%	13	30.77%	1	0.00%
OMR	2775	3.24%	2662	98.08%	320	75.94%	1160	89.22%	4	0.00%
IHI	12	91.67%	0	-	0	-	80	82.50%	0	-
PNC	5	0.00%	0	-	2	100.00%	64	7.81%	0	-
Total	13685	19.89%	8407	85.93%	1240	74.35%	9154	43.34%	940	68.19%
	Commons CLI		**JSoup**		**Commons CSV**		**Commons Codec**		**Closure**	
Mutant Operators	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
EOC	0	-	1	0.00%	0	-	0	-	2	0.00%
EMM	8	75.00%	0	-	0	-	0	-	4929	34.79%
OAN	2	100.00%	11	63.64%	0	-	34	17.65%	10	60.00%
JDC	0	-	0	-	0	-	0	-	0	-
JSI	30	50.00%	20	40.00%	9	77.78%	15	40.00%	124	49.19%
ISI	0	-	14	0.00%	0	-	1	100.00%	1	100.00%
PCI	0	-	2	0.00%	0	-	0	-	307	68.40%
PCC	0	-	0	-	0	-	0	-	0	-
PCD	0	-	0	-	0	-	0	-	0	-
ISD	0	-	3	0.00%	1	100.00%	0	-	0	-
JSD	10	0.00%	3	0.00%	9	22.22%	35	0.00%	76	40.79%
EAM	96	42.71%	43	55.81%	38	65.79%	8	0.00%	997	48.04%
PPD	0	-	0	-	0	-	0	-	0	-
IOR	0	-	0	-	0	-	0	-	2	0.00%
IOP	0	-	3	0.00%	2	0.00%	0	-	3	0.00%
PMD	0	-	0	-	0	-	0	-	0	-
PRV	47	85.11%	7	57.14%	7	100.00%	0	-	135	82.96%
IOD	4	25.00%	30	26.67%	5	40.00%	12	33.33%	78	39.74%
JID	5	20.00%	2	0.00%	3	100.00%	4	25.00%	55	7.27%
JTI	21	47.62%	21	57.14%	5	0.00%	13	0.00%	58	46.55%
JTD	0	-	14	71.43%	0	-	1	0.00%	38	71.05%
IPC	1	100.00%	8	0.00%	0	-	6	0.00%	11	0.00%
OMD	0	-	0	-	0	-	0	-	0	-
OMR	16	31.25%	20	90.00%	1	0.00%	11	18.18%	10	30.00%
IHI	0	-	4	75.00%	0	-	0	-	1	100.00%
PNC	0	-	5	60.00%	0	-	0	-	0	-
Total	240	50.83%	211	45.97%	80	58.75%	140	14.29%	6837	39.61%

less often than other mutants. The percentages displayed in the bar graph can be deceiving because there are mutants on there with very high kill rates, but a low number of mutants created. We needed a way to determine if that high (or low) kill rate is really unusual, or just distorted because of the number of mutants created.

In statistics, "unusual" data points are called *outliers*—data points that are significantly different from other data points in the set. To determine the outliers in our data—the mutants with a significantly different kill rate than others—we

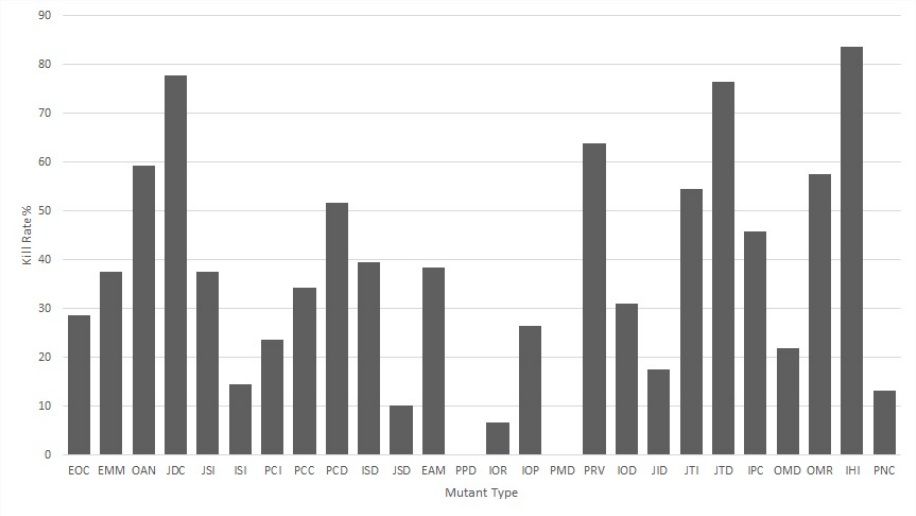

Figure 2: Class-Level Mutants Kill Rates

first completed a regression analysis, and then calculated studentized residuals.

Regression analysis can be used in statistics to estimate relationships among variables. In general, regression methods attempt to fit a model to observed data in order to quantify the relationship between two groups of variables. The fitted model can then be used to describe the relationship or predict new values. In our experiment, the data is the number of mutants created and the number of mutants killed. The regression analysis on our data provides, based on the data gathered, what the predicted—or *expected*—value would be for any number of mutants created for any mutant type. Figure 3 provides a scatterplot of our data. The line shows where the expected values would be. Specifically, it shows how many mutants would be expected to be killed (y-axis) according to the number of mutants that were created (x-axis).

The model created by regression analysis assumes that all mutant types are all killed at the same rate. However, we are trying to determine whether there are particular mutant types that are not killed at the same rate as others. To answer this question, we would need to determine which mutant types do not fit the regression model shown in Figure 3. The figure shows that many

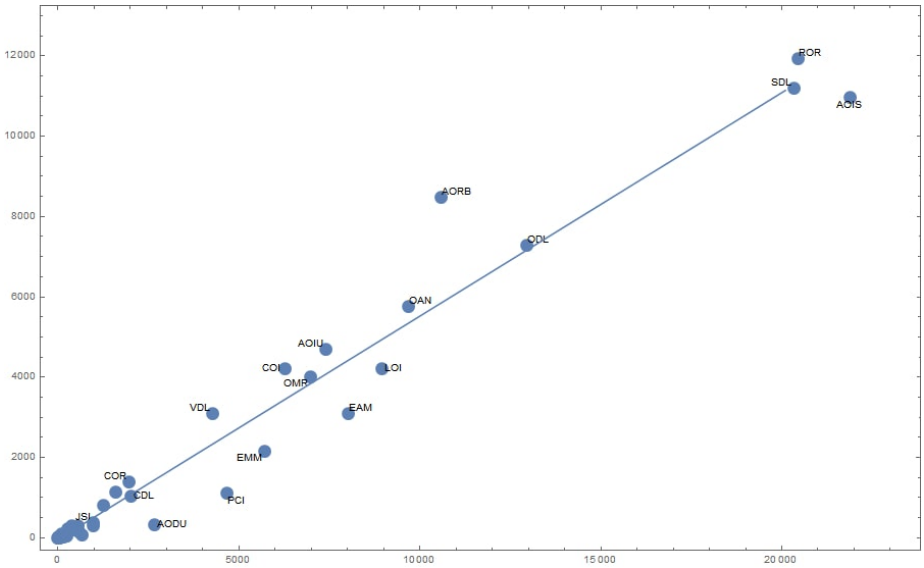

Figure 3: Scatter Plot of Mutant Types and Killed Mutants

of the mutant operators *do* follow that linear behavior. However, there are a few exceptions. AODU, PCI, EMM, EAM, LOI, and AOIS fall below the line, while AORB and ROR are above the line. The line shows how many mutants are expected to be killed according to the number of mutants created if the data set was linear (meaning all fault types are found at the same rate). When the results for a particular mutant type do not fall on the line (meaning it did not kill the number of mutants expected, according to the linear regression model), it indicates that particular mutant type *could* be an outlier. However, we cannot make any solid conclusions based on the visual appearance of the points plotted in Figure 3. Therefore, we calculate the *residuals* for each mutant type. Residuals are found by determining the difference between the observed value and the expected value (as shown in Equation 1).

$$Residual = ObservedValue - PredictedValue \tag{1}$$

One problem with residuals is the magnitude of the residual depends on the unit of measurement. This can make it difficult to determine unusual values. This problem can be eliminated by dividing the residual by an estimate of their

standard deviation, known as *studentized residuals*. Studentized residuals help identify outliers which are not consistent with the rest of the data. Data points that have a studentized residual with an absolute value of two or greater are considered statistically significant at the 95% a- level. A plot of the studentized residual values for each mutant operator is shown in Figure 4. Mutant operators with values of zero or near zero mean the results for this mutant operator are either the value that was expected according to the regression analysis (for values of zero) or close to expected (for values near zero). Results that are further from zero indicate that the rate of detection was different than what was expected.

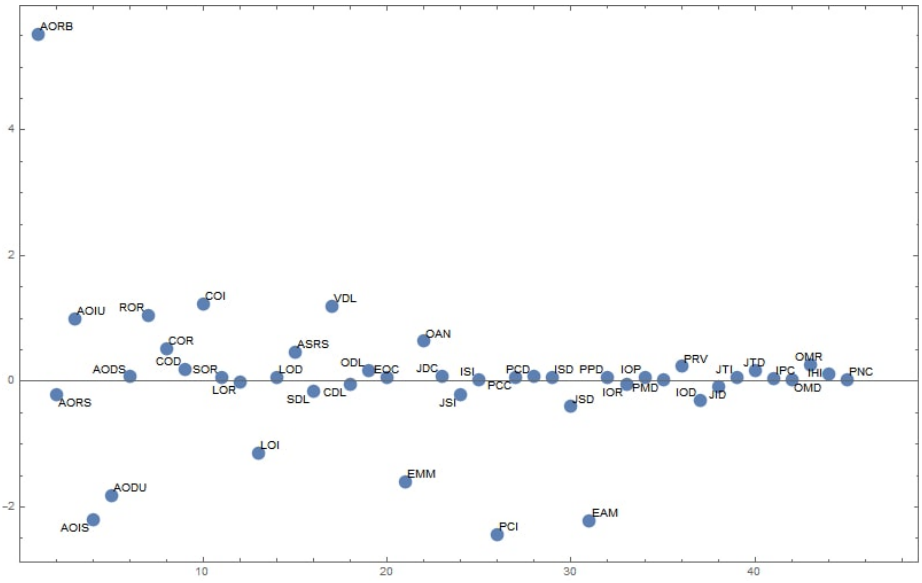

Figure 4: Studentized Residual with AORB

Table 6 provides the exact calculated studentized residual values for each mutant operator. The table includes five columns: the mutation operator, the total number of mutants created for that operator, the mutant's kill rate, the studentized residual using all mutants, and the studentized residual excluding AORB. The studentized residual for all mutants identified AORB as an *extreme* outlier, having a value of 5.53 (well above the absolute value of 2 which qualifies

Table 6: Mutant Operator Totals and Studentized Residuals

Mutant Operator	Total Created	Percent Killed	Studentized Residual (AORB Included)	Studentized Residual (AORB Excluded)
AORB	10586	80.14%	5.53	-
AORS	587	24.53%	-0.21	-0.23
AOIU	7402	63.48%	1	1.55
AOIS	21900	50.11%	-2.21	-2.14
AODU	2674	12.12%	-1.82	-2.31
AODS	92	54.35%	0.07	0.13
ROR	20443	58.34%	1.05	2.11
COR	1958	70.38%	0.53	0.77
COD	384	74.22%	0.19	0.28
COI	6268	67.21%	1.23	1.84
SOR	100	48.00%	0.06	0.11
LOR	437	42.79%	-0.01	0.02
LOI	8952	47.16%	-1.15	-1.22
LOD	9	33.33%	0.07	0.12
ASRS	1588	71.41%	0.47	0.68
SDL	20337	55.00%	-0.16	0.44
VDL	4283	72.19%	1.2	1.74
CDL	2022	51.58%	-0.06	0.01
ODL	12963	56.11%	0.17	0.61
EOC	14	28.57%	0.07	0.12
EMM	5710	37.58%	-1.59	-1.91
OAN	9694	59.32%	0.64	1.14
JDC	9	77.78%	0.08	0.13
JSI	983	37.54%	-0.21	-0.21
ISI	76	14.47%	0.02	0.06
PCI	4666	23.55%	-2.43	-3.12
PCC	35	34.29%	0.06	0.11
PCD	31	51.61%	0.07	0.12
ISD	43	39.53%	0.06	0.11
JSD	664	10.09%	-0.4	-0.48
EAM	8017	38.46%	-2.22	-2.7
PPD	6	00.00%	0.07	0.12
IOR	166	06.63%	-0.05	-0.04
IOP	19	26.32%	0.07	0.11
PMD	61	00.00%	0.02	0.06
PRV	1271	63.81%	0.24	0.37
IOD	969	30.96%	-0.3	-0.34
JID	247	17.41%	-0.07	-0.06
JTI	564	54.43%	0.06	0.13
JTD	284	76.41%	0.17	0.25
IPC	175	45.71%	0.05	0.09
OMD	78	21.79%	0.03	0.07
OMR	6979	57.42%	0.27	0.56
IHI	97	83.51%	0.12	0.18
PNC	76	13.16%	0.02	0.06

a value as an outlier with 95% confidence). A value with such an extreme residual value can pull the expected values line towards that observed value in such a way that other outliers could be missed. For this reason, we calculated the studentized residual values for the data excluding AORB as well. The studentized residual values excluding AORB are provided in the last column in Table 6 and a plot is shown in Figure 5.

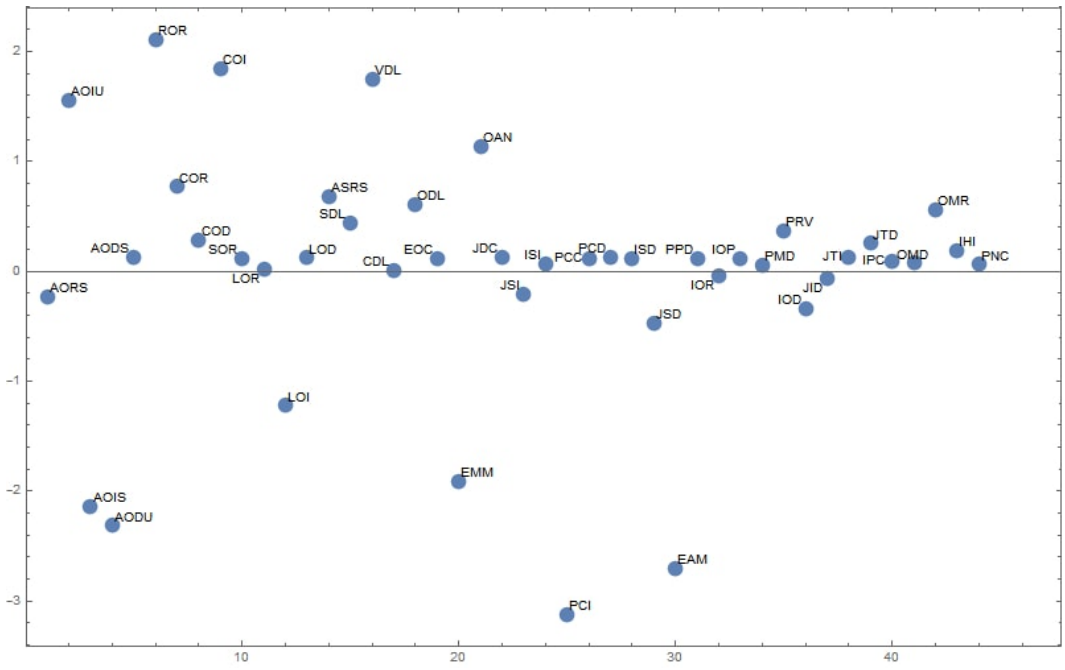

Figure 5: Studentized Residual without AORB

Examining the data with all mutant operators included reveals four outliers: AORB (with a value of 5.53), AOIS (with a value of -2.21), PCI (with a value of -2.43), and EAM (with a value of -2.22). A positive value indicates that the mutant operator was found *more often* than expected, while a negative value indicates the mutant type was killed *less often* than expected. Examining the data excluding the extreme outlier, AORB, reveals five outliers: AOIS (with a value of -2.14), AODU (with a value of -2.31), ROR (with a value of 2.11), PCI (with a value of -3.12), and EAM (with a value of -2.7).

5. Discussion

The results in the previous section identified six mutant types as outliers. Two of the mutant types are identified as outliers by having a kill rate *higher* than expected and four mutants types had a kill rate *lower* than expected. Since these are the unusual values that could be contributing the variance in the effectiveness of test suites, we direct our attention to these mutant types. Specifically, in this section, we provide a description of each mutant type of interest, a sample mutant that could be created using that mutant operator, and a discussion of the results for that mutant type.

5.1. Arithmetic Operator Replacement (Binary) (AORB)

The mutant with the most extreme results was AORB. It had a studentized residual value of 5.53. There were 10,586 mutants created for AORB with an overall kill rate of 80.14%. AORB is the *Arithmetic Operator Replacement (Binary)* mutant. It will replace basic binary arithmetic operators with other binary arithmetic operators. An example mutant, where the subtraction operator is replaced with the modulus operator, is provided below.

Original:	AORB Mutant:
`return i - 1;`	`return i % 1;`

This is a type of mutant operator where a high kill rate would be expected. Performing a completely different arithmetic operation will typically produce very different results, and should be caught by an effective test suite. Another factor that could have contributed to the results being an extreme outlier is the way MuJava seeds this particular type of fault. When MuJava sees the opportunity to insert this type of fault in the source code, it will create a mutation for every arithmetic operation. For instance, in the example above, it would create a mutant for i % 1 as shown, but also for i + 1, i * 1, and i / 1. This leads to four mutants created for each opportunity found. Theoretically, if the test code is able to kill one of these mutants, it should kill the other three as well.

5.2. Relational Operator Replacement (ROR)

ROR is the *Relational Operator Replacement* mutant. It will replace a relational operator with another relational operator. An example is provided below.

Original:	OAN Mutant:
`return x > 0;`	`return x < 0;`

This mutation operator makes a drastic change to the program, often producing results exactly opposite of the intended action. It was not surprising that it was found at a higher rate than other mutation operators. In fact, it may be more surprising that it was *just* over the threshhold (at 2.11) to be considered an outlier.

5.3. Type Cast Operator Insertion (PCI)

PCI is the *type cast operator insertion* mutant. The PCI operator changes the actual type of an object reference to the parent or child of the original declared type. An example is provided below.

Original:	PCI Mutant:
`Child cRef = new Child();`	`Child cRef = new Child();`
`Parent pRef = cRef;`	`Parent pRef = cRef;`
`pRef.toString();`	`((Child)pRef).toString();`

There were 4,666 PCI mutants created with an overall kill rate of 23.6%. This particular mutant could be hard to find because a child and parent class have many similarities. Many of the methods behave exactly the same way, otherwise there would be no point in creating a child class. The test suite would have to determine which methods in the child class behave differently than the version from the parent class, and then focus on those methods. This mutation operator

likely results in a larger number of *equivalent mutants* than some of the other mutation operators, as it carries a high likelihood of producing a mutation with the same result, regardless of input used. Identifying equivalency is not generally a decidable problem, so we do not have the means currently of determining how often this particular mutation produces equivalent results. In practice, however, faults of this nature are possible and should be guarded against. Test oracles should be able to distinguish between versions of the system where the right method is called, and versions where the incorrect (parent) method is called.

5.4. Accessor Method Change (EAM)

This was one of the more surprising results. EAM is the *Accessor Method Change* mutant. The EAM mutant changes a call to an accessor method with the call to a different compatible accessor method. An example mutant is provided below.

Original:	EAM Mutant:
`point.getX();`	`point.getY();`

It is surprising this mutant operator was found less often than expected because retrieving the wrong value would be something that could certainly throw calculations and results off. This is an important finding because this could be an easy mistake for a programmer to make with most modern IDE's having some form of autocomplete or IntelliSense. It would be easy to accidentally choose the wrong method if all options are similarly-named. Test oracles could be made to find more of these mutants by validating the values of class attributes more frequently and more thoroughly.

5.5. Arithmetic Operator Insertion (Shortcut) (AOIS)

AOIS is another unexpected and interesting result. AOIS is the *Arithmetic Operator Insertion* mutant operator. It will increase or decrease a variable using the increment or decrement operator. An example is shown below.

Original:	AOIS Mutant:
`return i;`	`return ++i;`

Incrementing or decrementing a variable at places where it is not warranted could produce incorrect results or have other unpredictable outcomes. For example, if a mutant was created to increment a variable in the loop header, it could cause the loop to not execute the correct number of times. A mutant of this type going undetected could cause many unwanted behaviors. Better testing of boundary values could improve detection of these kind of "off-by-one" errors. This type of fault may also corrupt the internal state of the class, but may be hard to detect through inspection of method output alone. Oracles that more throughly inspect internal state may also help here.

5.6. Arithmetic Operator Deletion (Unary) (AODU)

Similar to the last two, AODU is also a surprising result. AODU is the *Arithmetic Operator Deletion*. It deletes basic unary arithmetic operators (+, -, ++, –, !). An example is shown below.

Original:	AODU Mutant:
`return -1;`	`return 1;`

Like AOIS, changing the value of a variable by changing the unary arithmetic operator should cause incorrect results and unpredictable behavior. The fact that test cases miss these type of errors suggests that the test oracles being used to check results are not specific enough—that they are allowing slight variations in the output, or ignoring whether a value is positive or negative.

5.7. General Trends

The last three mutants discussed presented three very surprising, and interesting, results. Each of these mutation types change the internal state of the

program by using the wrong value in some way—one by retrieving the wrong variable, one by incrementing or decrementing a value, and one by deleting a unary arithmetic operator. Each of these types of mutants could be caught more often by adding more validation to the values of variables in the program at different times throughout the test methods. For instance, better testing of boundary values could assist in situations where the result may be slightly off of the correct value. Regardless of the level of code coverage, the test must have an oracle sufficiently powerful to detect an exposed fault. Merely executing a line of code does not ensure that a fault is triggered. At the same time, we must design oracles that are thorough enough to detect a fault when it is triggered. The role of the test oracle is often downplayed in testing research [46]. More research is needed on this side of the equation—in deciding which variables to monitor and evaluate using the oracle, and in the types of assertions to use to maximize the likelihood of fault detection. The level of coverage and the thoroughness of the oracle have a dual influence on whether faults are detected, and we must make improvements in both regards.

5.8. Impact of the Study

The results of our study advance the state-of-the art by revealing the nature of faults that are getting missed most frequently (with statistical significance) by human-written test suites achieving high code coverage. Specifically, as described in the last section, three of the fault types missed at a statistically higher rate have a common problem: the internal state of the program is corrupted, and the results either do not change the output—or the oracle is not specific enough to detect the change. Developing test suites and test oracles more capable of finding these faults will improve the overall effectiveness of test suites.

Based on our results, we suggest that in order to strengthen test suites, test creation methods should consider whether test oracles are validating internal state along with meeting high code coverage. In order to find faults where the program has an incorrect internal state, tests need to not only execute the buggy code, but also have a test oracle sufficient to catch the buggy state.

Our recommendation is supported by the nature of the faults missed in our study as well as recent research investigating the relationship between assertions and test suite effectiveness [47, 13, 46]. Assertions are a type of a test oracle. Zhang and Mesbah conducted an empirical study and found that the number of assertions in a test suite strongly correlates with its effectiveness [13]. Chen et.al investigated how assertions impacted coverage-based test suite reduction techniques [47]. They found assertions are significantly correlated with the effectiveness of test suites used by coverage-based test suite reduction. In our past work, we found that the selection of program variables to monitor and check with the oracle has a major impact on fault detection [46]. We found that inspection of internal state has a major impact on the likelihood of fault detection, and that developers should monitor and inspect key bottleneck points in program execution.

These studies show that the choice and formulation of test oracles is important and makes a difference in test suite effectiveness. Our study advances this knowledge by providing empirical evidence showing *why* test oracles are important. We have identified specific fault types that create internal state problems, and provide evidence that test suites evaluated with coverage metrics alone did not detect them at the expected rate. Researchers and developers can improve their test suites by adding or improving test oracles in situations where bugs, similar to the ones shown in our study, are possible. Furthermore, researchers can use the evidence found in our study as motivation for developing test evaluation methods that consider code coverage and test oracles together, and their dual influence on test suite effectiveness.

6. Conclusions and Future Work

Code coverage is widely used as a method for evaluating the effectiveness of test suites. However, research has shown achieving high code coverage is not always a good indicator of fault-finding capability. Many past studies investigating the correlation between code coverage and fault detection have had

inconsistent findings, only sometimes showing a correlation between the two. In this work, we investigated one possible source of the inconsistencies observed: fault type. Specifically, we investigated how effective test suites achieving high code coverage were at detecting 45 different types of faults. Our results show that the effectiveness of finding the different faults varied greatly, and certain types of faults were missed at a much higher rate than others. Specifically, three of the four fault types identified as outliers were fault types that could corrupt internal state by using an incorrect value in some way. Based on this result, we suggest that test oracles should more thoroughly inspect internal state and boundary values, that developers carefully consider which variables are inspected using the test oracle, and that the dual influence of code coverage and test oracle be considered when evaluating test suites.

There is still much work to be done on this particular problem. Identifying the types of faults that are missed most frequently is only the first step. The information gained from this study can be used to propose improvements to test suites that would make them more capable of finding these faults. Also, although our study identifies one factor that is affecting the ability of test suites to find faults, there could be other factors as well—particularly centered around the role and formulation of test oracles. Additionally, other methods of evaluating test suites could be proposed and studied to identify whether they are a better indicator of a test suites' ability to find faults.

7. Acknowledgements

This work has been partially supported by the Office of Sponsored Awards and Research Support (SARS) from the University of South Carolina Upstate.

References

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

[2] A. Groce, M. A. Alipour, R. Gopinath, Coverage and its discontents, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward!'14, ACM, New York, NY, USA, 2014, pp. 255–268. doi:10.1145/2661136.2661157.
URL http://doi.acm.org/10.1145/2661136.2661157

[3] RTCA/DO-178C, Software considerations in airborne systems and equipment certification.

[4] M. Heimdahl, M. Whalen, A. Rajan, M. Staats, On MC/DC and implementation structure: An empirical study, in: Digital Avionics Systems Conference (DASC), 2008, pp. 5–B.

[5] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, F. Tip, A framework for automated testing of javascript web applications, in: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, 2011, pp. 571–580.

[6] Q. Yang, J. J. Li, D. M. Weiss, A survey of coverage-based testing tools, The Computer Journal 52 (5) (2007) 589–597.

[7] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72–82.

[8] A. S. Namin, J. H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the eighteenth International Symposium on Software Testing and Analysis, 2009, pp. 57–68.

[9] G. Gay, The fitness function for the job: Search-based generation of test suites that detect real faults, in: Proceedings of the International Conference on Software Testing, ICST 2017, IEEE, 2017.

[10] G. Gay, M. Staats, M. Whalen, M. Heimdahl, The risks of coverage-directed test case generation, Software Engineering, IEEE Transactions on PP (99). doi:10.1109/TSE.2015.2421011.

[11] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435–445.

[12] G. Gay, A. Rajan, M. Staats, M. Whalen, M. P. E. Heimdahl, The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage, ACM Trans. Softw. Eng. Methodol. 25 (3) (2016) 25:1–25:34. doi:10.1145/2934672.
URL http://doi.acm.org/10.1145/2934672

[13] Y. Zhang, A. Mesbah, Assertions are strongly correlated with test suite effectiveness, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 214–224.

[14] A. Schwartz, M. Hetzel, The impact of fault type on the relationship between code coverage and fault detection, in: Proceedings of the 11th International Workshop on Automation of Software Test, ACM, 2016, pp. 29–35.

[15] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, et al., An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[16] C. S. Jensen, M. R. Prasad, A. Møller, Automated testing with targeted event sequence generation, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, 2013, pp. 67–77.

[17] B. N. Nguyen, B. Robbins, I. Banerjee, A. Memon, Guitar: an innovative tool for automated testing of gui-driven software, Automated Software Engineering 21 (1) (2014) 65–105.

[18] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium

and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 416–419.

[19] I. Ghosh, N. Shafiei, G. Li, W.-F. Chiang, JST: An automatic test generation tool for industrial java applications with strings, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 992–1001.

[20] P. D. Marinescu, C. Cadar, Katch: high-coverage testing of software patches, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 235–245.

[21] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, A. M. Memon, Using gui ripping for automated testing of android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 258–261.

[22] F. Gross, G. Fraser, A. Zeller, Search-based system testing: high coverage, no false alarms, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM, 2012, pp. 67–77.

[23] K. Inkumsah, T. Xie, Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution, in: 23rd International Conference on Automated Software Engineering, 2008, pp. 297–306.

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

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

[26] F. Del Frate, P. Garg, A. P. Mathur, A. Pasquini, On the correlation between code coverage and software reliability, in: Sixth International Symposium on Software Reliability Engineering, 1995, pp. 124–132.

[27] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, D. Marinov, Comparing non-adequate test suites using coverage criteria, in: Proceedings of the International Symposium on Software Testing and Analysis, 2013, pp. 302–313.

[28] P. S. Kochhar, F. Thung, D. Lo, Code coverage and test suite effectiveness: Empirical study with real bugs in large systems, in: 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 560–564.

[29] M. Staats, G. Gay, M. Whalen, M. Heimdahl, On the danger of coverage directed test case generation, in: Fundamental Approaches to Software Engineering, 2012, pp. 409–424.

[30] A. Perez, R. Abreu, A. van Deursen, A test-suite diagnosability metric for spectrum-based fault localization approaches, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, 2017, pp. 654–664.

[31] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, ACM Trans. Softw. Eng. Methodol. 24 (4) (2015) 23:1–23:49. doi:10.1145/2699688.
URL http://doi.acm.org/10.1145/2699688

[32] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated white-box test generation really help software testers?, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA, ACM, New York, NY, USA, 2013, pp. 291–301. doi:10.1145/2483760.2483774.
URL http://doi.acm.org/10.1145/2483760.2483774

[33] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, J. Benefelds, An industrial evaluation of unit test generation: Finding real faults in a financial

application, in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP), ICSE 2017, ACM, New York, NY, USA, 2017.

[34] P. G. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, ACM SIGSOFT Software Engineering Notes 23 (6) (1998) 153–162.

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

[36] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria, in: Proceedings of the 16th International Conference on Software Engineering, 1994, pp. 191–200.

[37] Y.-S. Ma, Y.-R. Kwon, J. Offutt, Inter-class mutation operators for java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering, 2002, pp. 352–363.

[38] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of mujava, in: Proceedings of the 2006 International Workshop on Automation of Software Test, 2006, pp. 78–84.

[39] V. Debroy, W. E. Wong, Insights on fault interference for programs with multiple bugs, in: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, IEEE, 2009, pp. 165–174.

[40] N. DiGiuseppe, J. A. Jones, Fault interaction and its repercussions, in: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE, 2011, pp. 3–12.

[41] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings. 27th International Conference on Software Engineering, 2005, pp. 402–411.

[42] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing, in: 22nd International Symposium on the Foundations of Software Engineering, 2014, pp. 654–665.

[43] R. Just, The major mutation framework: Efficient and scalable mutation analysis for java, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 433–436.

[44] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava: a mutation system for java, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 827–830.

[45] J. Offutt, Y.-S. Ma, Description of mujava's method-level mutation operators, in: Electronics and Telecommunications Research Institute, Korea, Tech. Rep, 2005.

[46] G. Gay, M. Staats, M. Whalen, M. Heimdahl, Automated oracle data selection support, Software Engineering, IEEE Transactions on PP (99) (2015) 1–1. doi:10.1109/TSE.2015.2436920.

[47] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, How do assertions impact coverage-based test-suite reduction?, in: Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on, IEEE, 2017, pp. 418–423.