A Study of the Influence of Assertions and Mutants on Test Suite Effectiveness

by

Yucheng Zhang

B.Sc., The University of British Columbia, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2016

Abstract

Test suite effectiveness is measured by assessing the portion of faults that can be detected by tests. To precisely measure a test suite's effectiveness, one need to pay attention to both tests and the set of faults used. Code coverage is a popular test adequacy criterion in practice. Code coverage, however, remains controversial as there is a lack of coherent empirical evidence for its relation with test suite effectiveness. More recently, test suite size has been shown to be highly correlated with effectiveness. However, previous studies treat test methods as the smallest unit of interest, and ignore potential factors influencing the correlation between test suite size and test suite effectiveness. We propose to go beyond test suite size, by investigating test assertions inside test methods. First, we empirically evaluate the relationship between a test suite's effectiveness and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. We compose 6,700 test suites in total, using 24,000 assertions of five real-world Java projects. We find that the number of assertions in a test suite strongly correlates with its effectiveness, and this factor positively influences the relationship between test suite size and effectiveness. Our results also indicate that assertion coverage is strongly correlated with effectiveness. Second, instead of only focusing on the testing side, we propose to investigate test suite effectiveness also by considering fault types (the ways faults are generated) and faults in different types of statements. Measuring a test suite's effectiveness can be influenced by using faults with different characteristics. Assessing test suite effectiveness without paying attention to the distribution of faults is not precise. Our results indicate that fault type and statement type where the fault is located can significantly influence a test suite's effectiveness.

Preface

This thesis presents two large-scale empirical studies on the influencing factors of test suite effectiveness, conducted by myself in collaboration with my supervisor Professor Ali Mesbah. Chapter 3 has been published as a full conference paper at the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015) [44]. The results described in Chapter 4 are submitted to a software testing conference as a full paper. I was responsible for devising the experiments, running the experiments, analyzing the results, and writing the manuscript. My supervisor was responsible for guiding me with the creation of the idea and experimental methodology, the design of the procedure for analyzing the experimental results, as well as writing Chapters 3 and 4.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank all people who helped me successfully obtain my MASc degree. First of all, I would like to thank my advisor Professor Ali Mesbah, for his patient guidance, encouragement and advice throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions so promptly.

I would also like to thank my fellow colleagues in the SALT lab at UBC for the stimulating discussions, critical feedback on my work, and for all the fun we have had in the last two years. They made my entire experience enjoyable.

Last but not the least, I would also like to thank my father Wen Zhang, my mother Huiping Dai, my uncle He Dai, my aunt Huihong Dai, my grandparents Ruying Ding and Houfa Dai, and all my relatives. I would not have the opportunity to pursue my study without their love and unwavering support.

Chapter 1

Introduction

Software testing has become an integral part of software development. A software product cannot be confidently released unless it is adequately tested. Code coverage is the most popular test adequacy criterion in practice. However, coverage alone is not the goal of software testing, since coverage without checking for correctness is meaningless. A more meaningful adequacy metric is the fault detection ability of a test suite, also known as test suite effectiveness.

Test suite effectiveness measures how well a test suite is capable of detecting faults. A common technique used to measure test suite effectiveness is mutation testing, in which small programmer errors are simulated to see if the test suite can kill the mutant. Naturally, the measurement is dependent on two components, namely, the test suite and faults simulated.

There have been numerous studies analyzing the relationship between test suite size, code coverage, and test suite effectiveness [17, 18, 23–26, 33]. More recently, Inozemtseva and Holmes [28] found that there is a moderate to very strong correlation between the effectiveness of a test suite and the number of test methods, but only a low to moderate correlation between the effectiveness and code coverage when the test suite size is controlled for. These findings imply that (1) the more test cases there are, the more effective a test suite becomes, (2) the more test cases there are, the higher the coverage, and thus (3) test suite size plays a prominent role in the observed correlation between coverage and effectiveness.

All these studies treat test methods as the smallest unit of interest. However, we believe such coarse-grained studies are not sufficient to show the main factors influencing a test suite's effectiveness. In this thesis, we propose to dissect test methods and investigate why test suite size correlates strongly with effectiveness. To that end, we focus on test assertions inside test methods. Test assertions are statements in test methods through which desired specifications are checked against actual program behaviour. As such, assertions are at the core of test methods. We hypothesize that assertions[1] have a strong influence on test suite effectiveness, and this influence, in turn,

[1]We use the terms 'assertion' and 'test assertion' interchangeably in this thesis.

is the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness.

Additionally, measuring test suite effectiveness not only depends on the quality of the tests, but it might also depend on which faults are seeded and where. Assessing test suite effectiveness without noticing the distribution of faults can be biased. We hypothesis that, seeded faults with different characteristics can influence measuring a test suite's effectiveness. In this thesis, we also measure test effectiveness against different types of faults and faults located in different types of statements. To the best of our knowledge, we are the first to conduct a large-scale empirical study to assess the relationship between test suite effectiveness and fault type and location.

In this thesis, We use mutants to simulate real faults. Mutants have been widely adopted to substitute real faults in the literature [18, 27, 28, 33, 37, 39]. There is also empirical evidence for mutants being representable for real faults [14, 15, 19, 29]. We generate mutants by using the seven default mutation operators provided by PIT [10].

1.1 Thesis Contribution

This thesis makes the following main contributions:

- The first large-scale study analyzing the relation between test assertions and test suite effectiveness. Our study composes 6,700 test suites in total, from 5,892 test cases and 24,701 assertions of five real-world Java projects in different sizes and domains.
- Empirical evidence that (1) test assertion quantity and assertion coverage are strongly correlated with a test suite's effectiveness, (2) assertion quantity can significantly influence the relationship between a test suite's size and its effectiveness, (3) the correlation between statement coverage and effectiveness decreases dramatically when assertion coverage is controlled for.
- A classification and analysis of the effectiveness of assertions based on their properties, such as (1) creation strategy (human-written versus automatically generated), (2) the content type asserted on, and (3) the actual assertion method types.
- Empirical evidence that assertions classified by their types, such as the content type they assert on, or assertion method types, (1) might not be fine-grained enough to differentiate from each other, and (2) cannot significantly influence a test suite's effectiveness

- A quantitative analysis of the relationship between (1) seeded fault type and test suite effectiveness, and (2) seeded fault location (in different statement types) and test suite effectiveness.
- Empirical evidence that fault type and statement type where fault is located can significantly influence a test suite's measured effectiveness.

1.2 Thesis Organization

This chapter serves to establish the overarching goal and motivation of this thesis. Chapter 2 discusses the related work. Chapter 3 describes in detail the experimental design and results found from the investigation of the influence of assertions on test suite effectiveness. Chapter 4 describes in detail the experiments we conducted to explore the influence of mutants on measuring test suite effectiveness. Chapter 5 discusses the findings from Chapter 3 and Chapter 4, and Chapter 6 concludes and presents future research directions.

An initial version of Chapter 3 is published as a full conference paper: Yucheng Zhang, and Ali Mesbah. "Assertions Are Strongly Correlated with Test Suite Effectiveness". In Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). 214-224, 2015 [44].

Chapter 2

Related Work

2.1 Coverage Metrics

There is a large body of empirical studies investigating the relationship between different coverage metrics (such as statement, branch, MC/DC) and test suite effectiveness [17, 18, 23–26]. All these studies find some degree of correlation between coverage and effectiveness. However, coverage remains a controversial topic [18, 28] as there is no strong evidence for its direct relation with effectiveness. The reason is that coverage is necessary but not sufficient for a test suite to be effective. For instance, a test suite might achieve 100% coverage but be void of test assertions to actually check against the expected behaviour, and thus be ineffective.

2.2 Test Suite Size

Researchers have also studied the relationship between test suite size, coverage, and effectiveness [28, 33]. In these papers, test suite size is measured in terms of the number of test methods in the suite. Different test suites, with size controlled, are generated for a subject program to study the correlation between coverage and effectiveness. Namin and Andrews [33] report that both size and coverage independently influence test suite effectiveness. More recently, Inozemtseva and Holmes [28] find that size is strongly correlated with effectiveness, but only a low to moderate correlation exists between coverage and effectiveness when size is controlled for. None of these studies, however, looks deeper into the test cases to understand why size has a profound impact on effectiveness. In our work, we investigate the role test assertions play in effectiveness.

2.3 Assertion Coverage

Schuler and Zeller [38] propose the notion of 'checked coverage'[2] as a metric to assess test oracle quality. Inspired by this work, we measure assertion coverage of a test suite as the percentage of statements directly covered by the assertions. We are interested in assertion coverage because it is a metric directly related with the assertions in the test suite. In the original paper [38], the authors evaluated the metric by showing that there is a similar trend between checked coverage, statement coverage, and mutation score. In this thesis, we conduct an empirical study on the correlation level between assertion coverage and test suite effectiveness. In addition, we compose a large set of test suites (up to thousands) for each subject under test, whereas only seven test suites were compared in the original paper. Moreover, we study the correlation between statement coverage and test suite effectiveness, to compare with the relationship between assertion coverage and test suite effectiveness, by composing test suites with assertion coverage controlled.

2.4 Test Characteristics

Cai and Lyu [18] studied the relationship between code coverage and fault detection capability under different testing characteristics. They found that the effect of code coverage on fault detection varies under different testing profiles. Also, the correlation between the two measures is strong with exceptional test cases, while weak in normal testing settings. However, they did not examine the role assertions might play in different profiles on the effectiveness of test cases. To the best of our knowledge, we are the first to investigate the influence of different assertion properties on suite effectiveness. We classify assertion properties in three categories, and study the effectiveness of each classification separately.

2.5 Mutation Operators

Researchers have also sought to find sufficient subsets of mutation operators to reduce the computational cost of mutation testing. Offutt and Rothermel empirically compared the effectiveness of selective mutation with standard mutation in [34, 35]. They empirically evaluated [35] the mean loss in mutation score using 2-selective, 4-selective, and 6-selective mutation. They

[2]We use the terms 'assertion coverage' and 'checked coverage' interchangeably in this thesis.

found [34] 5 out of 22 mutation operators, used by Mothra, suffice to efficiently implement mutation testing for achieving 99.5 percent mutation score. Wong and Mathur [42] implemented the idea of constraint mutation by using only two mutation operators. From their evaluation, 80 percent of mutants are reduced with only 5 percent loss in mutation score. Mresa and Bottaci [30] took the cost of detecting equivalent mutants into consideration while evaluating selective mutation.

In addition, researchers have explored [31, 32, 40, 41] the sufficient subset of mutation operators for measuring test effectiveness. More recently, Deng et al. [22] and Delamaro et al. [20] evaluated statement deletion mutation operator (SDL), and found mutants generated by SDL require tests that are highly effective at killing other mutants for Java and C programs, respectively. Researchers have also investigated guidelines for mutation reduction [13, 16, 21]. Zhang et al. [43] examined if operator-based mutant-selection techniques are superior to random mutation selection. All of the studies mentioned, compare between groups of mutation operators, in the context of mutation selection. They aim at reducing the number of mutants generated without significantly loss of test effectiveness. In this thesis, however, we compare between individual mutation operators. We are mainly interested in whether using mutants generated by different mutation operators can lead to a significant influence on measuring test suit effectiveness. To the best of our knowledge, we are the first to conduct a large scale empirical study on the influence of fault location on measuring test effectiveness

Chapter 3

Assertions Are Strongly Correlated with Test Suite Effectiveness

Code coverage is a popular test adequacy criterion in practice. Code coverage, however, remains controversial as there is a lack of coherent empirical evidence for its relation with test suite effectiveness. More recently, test suite size has been shown to be highly correlated with effectiveness. However, previous studies treat test methods as the smallest unit of interest, and ignore potential factors influencing this relationship. We propose to go beyond test suite size, by investigating test assertions inside test methods. We empirically evaluate the relationship between a test suite's effectiveness and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. We compose 6,700 test suites in total, using 24,000 assertions of five real-world Java projects. We find that the number of assertions in a test suite strongly correlates with its effectiveness, and this factor directly influences the relationship between test suite size and effectiveness. Our results also indicate that assertion coverage is strongly correlated with effectiveness and different types of assertions can influence the effectiveness of their containing test suites.

This chapter was partially published as "Assertions Are Strongly Correlated with Test Suite Effectiveness" in the Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) [44].

3.1 Experimental Design

The goal of this chapter is to study the relationship between assertions and test suite effectiveness. To achieve this goal, we design controlled experiments to answer the following research questions:

RQ1 Is the number of assertions in a test suite correlated with effectiveness?

RQ2 Is the assertion coverage of a test suite correlated with effectiveness?
RQ3 Does the type of assertions in a test suite influence effectiveness?

We examine these three main aspects of assertions in our study because (1) almost all test cases contain assertions, but the number of assertions varies across test suites (see Table 3.2); we aim to investigate if the number of assertions plays a role in effectiveness, (2) the fraction of statements in the source code executed and checked directly by assertions should intuitively be closely related to effectiveness; we set out to explore if and to what degree this is true; and (3) assertions have different characteristics, which may potentially influence a test suite's effectiveness, such as their method of creation (e.g., human-written, automatically generated), the type of arguments they assert on (e.g., boolean, string, integer, object), and the assertion method itself (e.g., `assertTrue`, `assertEquals`).

All our experimental data is publicly available.[3]

[3] `http://salt.ece.ubc.ca/software/assertion-study/`

Table 3.1: Characteristics of the subject programs.

ID	Subjects	Java SLOC	Test SLOC	Test cases	Assertions	Statement coverage	Assertion coverage
1	JFreeChart [8]	168,777	41,382	2,248	9,177	45%	30%
2	Apache Commons Lang [1]	69,742	41,301	2,614	13,099	92%	59%
3	Urban Airship Java Library [12]	35,105	11,516	503	1,051	72%	53%
4	lambdaj [9]	19,446	4,872	310	741	93%	65%
5	Asterisk-Java [2]	36,530	4,243	217	633	24%	10%
Total/Average		329,600	103,314	5,892	24,701	45%	30%

3.1.1 Terminology

Test case: a JUnit4 test method annotated with `@Test`. We use the terms 'test method' and 'test case' interchangeably in this chapter.
Test suite: the collection of a subject program's test cases.
Test suite size: number of test cases in a test suite.
Master/original test suite: the test suite written by the developers of a subject program.

3.1.2 Subject Programs

To automate data collection, we selected Java programs that use Apache Maven[4] as their build system, and JUnit4 as their testing framework. We select programs of different sizes to ensure the experiment results are not project size dependent.

Our set of subjects contains five projects in different application domains. JFreeChart [8] is a free Java chart library for producing charts. Apache Commons Lang [1] is a package of Java utility classes for the classes that are in java.lang's hierarchy. Urban Airship Java Library [12] is a Java client library for the Urban Airship API. Lambdaj [9] is a Java project for manipulating collections in a pseudo-functional and statically typed way. The last subject, Asterisk-Java [2], is a free Java library for Asterisk PBX integration.

The characteristics of these subject programs are summarized in Table 4.1. Lines of source code are measured using SLOCCount [11]. Columns 5–8 illustrate test suite size in terms of number of test methods, assertion quantity, statement coverage, and assertion coverage, of each subject's master test suite, respectively. Table 3.2 presents descriptive statistics regarding the number of assertions per test case for the subject systems.[5]

3.1.3 Procedure

To study the relationship between assertions and test suite effectiveness, a large set of test suites with different assertion related properties are required. In this section, we present how the experiments are conducted with respect to each research question. We first discuss the variables of interest, then explain how test data are collected by generating new test suites, and finally describe how the results are analyzed.

[4] `http://maven.apache.org`

[5] We were surprised to see such high max numbers of assertions per test case, so we manually verified these numbers. For instance, the 114 max assertions for JFreeChart are in the `testEquals` test method of the `org.jfree.chart.plot.CategoryPlotTest` class.

Table 3.2: Number of assertions per test case.

ID	Min	1st Q.	Median	3rd Q.	Max	Mean	σ
1	0	1	2	4	114	4.1	6.7
2	0	1	3	6	104	5.1	7.4
3	0	0	1	2	42	2.1	3.6
4	0	1	2	3	21	2.8	3.2
5	0	1	2	3	17	3.0	2.9

Effectiveness of Assertion Quantity (RQ1)

In order to answer RQ1, we investigate three variables, namely, number of test methods, number of assertions, and test suite effectiveness. We collect data by generating test suites in three ways, (1) randomly, (2) controlling test suite size, and (3) controlling assertion quantity. For each set of test suites, we compute the correlation between the three variables.

Number of test cases We implemented a tool that uses the JavaParser [6] library to identify and count the total number of test cases in a given test suite.

Number of assertions For each identified test case, the tool counts the number of test assertions (e.g., `assertTrue`) inside the body of the test case.

Test suite effectiveness Effectiveness captures the fault detection ability of a test suite, which can be measured as a percentage of faults detectable by a test suite. To measure the fault detection ability of a test suite, a large number of known real faults are required for each subject, which is practically unachievable. Instead, researchers generate artificial faults that resemble developer faults using techniques such as mutation testing. In mutation testing, small syntactical changes are made to random locations in the original program to generate a large number of mutants. The test suite is then run against each mutant. A mutant is killed if any of the test case assertions fail or the program crashes.

Mutation score. The mutation score, calculated as a percentage of killed mutants over total number of non-equivalent mutants, is used to estimate fault detection ability of a test suite. Equivalent mutants are syntactically different but semantically the same as the origin program, and thus undetectable by any test case. Since there is no trivial way of identifying equivalent mutants, similar to other studies [28], we treat all mutants that cannot be detected by a

program's original (master) test suite, as equivalent mutants when calculating mutation scores for our generated test suites.

Mutations are produced by transforming a program syntactically through mutation operators, and one could argue about the eligibility of using the mutation score to estimate a test suite's effectiveness. However, mutation testing is extensively used as a replacement of real fault detection ability in the literature [18, 28, 33]. There is also empirical evidence confirming the validity of mutation testing in estimating test suite effectiveness [14, 15, 19, 29].

We use the open source tool PIT [10] to generate mutations. We tested each of our subject programs to ensure their test suites can successfully execute against PIT. We use PIT's default mutation operators in all of our experiments.

Generating test suites To answer RQ1, we generate test suites in three different ways, from the master test suites of the subject programs.

Random test suites. We first generate a set of test suites by randomly selecting a subset of the test cases in the master test suite, without replacement. The size of each generated test suite is also randomly decided. In other words, we generate this set of test suites without controlling on test suite size or assertion quantity.

Controlling the number of test methods. Each test case typically has one or more assertions. A test suite with more test cases is likely to contain more assertions, and vice versa. From our observations, if test suites are randomly generated, there exists a linear relationship between test suite size and the number of assertions in the suites. If there is a linear relationship between two properties A (e.g., assertion quantity) and B (e.g., suite size), a relationship between A and a third property C (e.g., effectiveness) can easily transform to a similar relationship between B and C through transitive closure. To remove such indirect influences, we generate a second set of test suites by controlling the size. More specifically, a target test suite contains all of the test methods but only a subset of the assertions from the master test suite. Based on the total number of assertions in the master test suite, we first select a base number b, which indicates the size of the smallest test suite, and a step number x, which indicates size differences between test suites. Therefore, the i-th test suite to be generated, contains all of the test cases but only $b + x * i$ randomly selected assertions of the master test suite.

Controlling the number of assertions. We also generate another set of test suites by controlling on assertion quantity. To achieve this, we first assign test cases to disjoint *buckets* according to the number of assertions they

contain. For instance, for JFreeChart, test cases are assigned to three disjoint buckets, where bucket *low* contains test cases with 2 or less assertions, bucket *middle* contains test cases with 3 or 4 assertions, and bucket *high* contains the rest of test cases which have 5 or more assertions. We divide test cases in this way such that each bucket has a comparable size. Then we generate 100 test suites from each of the buckets randomly without replacement. Following this process, with a similar test suite size, test suites generated from bucket *high* always contain more assertions than test suites generated from bucket *middle*, and so forth.

Correlation analysis For RQ1, we use Pearson and Kendall's correlation to quantitively study the relationship between test suite size, assertion quantity, and test suite effectiveness. The Pearson correlation coefficient indicates the strength of a linear relationship between two variables. The Kendall's correlation coefficient measures the extent to which, as one variable increases, the other variable tends to increase, without requiring that increase to be represented by a linear relationship.

Effectiveness of Assertion Coverage (RQ2)

To answer RQ2, we measure a test suite's assertion coverage, statement coverage, and effectiveness. We collect data by first looking at the set of test suites which were randomly generated for RQ1, then generate a new set of test suites by controlling their assertion coverage. For each of the two sets of test suites, we study and compare the correlations between the three variables using the same analysis methods as described in Section 3.1.3.

Explicit mutation score Not all detectable faults in a program are detected by test assertions. From our observations, mutants can either be *explicitly killed* by assertions or *implicitly killed* by program crashes. Programs may crash due to unexpected exceptions. Program crashes are much easier to detect as they do not require dedicated assertions in test cases. On the other hand, all the other types of faults that do not cause an obvious program crash, are much more subtle and require proper test assertions for their detection. Since the focus of our study is on the role of assertions in effectiveness, in addition to the mutation score, we also compute the *explicit mutation score*, which measures the fraction of mutants that are explicitly killed by the assertions in a test suite. Table 3.3 provides mutation data in terms of the number of mutations generated for each subject, number of mutants killed by the test suites, number of mutants killed only by test

Table 3.3: Mutation data for the subject programs.

ID	Mutants	Killed (#)	Killed by Assertions (#)	Killed by Assertions (%)
1	34,635	11,299	7,510	66%
2	11,632	9,952	7,271	73%
3	4,638	2,546	701	28%
4	1,340	1,084	377	35%
5	4,775	957	625	65%

assertions (e.g., excluding crashes), and the percentage of mutants killed by assertions with respect to the total number of killed assertions.

From what we have observed in our experiments, PIT always generates the same set of mutants for a piece of source code when executed multiple times. Thus, to measure the *explicit mutation score* of a test suite, we remove all assertions from the test suite, measure its mutation score again, and then subtract the fraction of implicit killed mutants from the original mutation score.

Assertion coverage Assertion coverage, also called checked coverage [38], measures the fraction of statements in the source code executed via the backward slice of the assertion statements in a test suite.

We use the open source tool JavaSlicer [7] to identify *assertion checked statements*, which are statements in the source code executed through the execution of assertions in a test suite. JavaSlicer is an open-source dynamic slicing tool, which can be used to produce traces of program executions and offline dynamic backward slices of the traces. We automatically identify checked statements of a test suite by (1) identifying all assertion statements and constructing slicing criteria, (2) using JavaSlicer to trace each test class separately, and (3) mining the traces computed in the previous step to identify dynamic backward slices of the assertions, and finally (4) since each backward slice of an assertion includes statements from the test case, calls to the JUnit APIs, and statements from the source code, we filter out the data to keep only the statements pertaining to the source code.

For large test suites, we observed that using JavaSlicer is very time consuming. Thus, we employ a method to speed up the process in our experiments. For all the test classes in each original master test suite, we repeat steps 1–4 above, to compute the *checked statements* for each test method individually. Each statement in the source code is uniquely

identified by its classname and line number and assigned an ID. We then save information regarding the *checked statements* of each test method into a data repository. Once a new test suite is composed, its *checked statements* can be easily found by first identifying each test method in the test suite, then pulling the *checked statements* of the test method from the data repository, and finally taking a union of the *checked statements*. The *assertion coverage* of a generated test suite is thus calculated as the total number of *checked statements* of the suite divided by the total number of statements.

Statement coverage Unlike assertion coverage, which only covers what assertion statements execute, statement coverage measures the fraction of the source code covered through the execution of the whole test suite.

In this chapter, we select statement coverage out of the traditional code coverage metrics as a baseline to compare with assertion coverage. The reason behind our selection is twofold. First, statement coverage is one of the most frequently used code coverage metrics in practice since it is relatively easy to compute and has proper tool support. Second, two recent empirical studies suggest that statement coverage is at least as good as any other code coverage metric in predicting effectiveness. Gopinath et al. [26] found statement coverage predicts effectiveness best compared to block, branch, or path coverage. Meanwhile, Inozemtseva and Holmes [28] found that stronger forms of code coverage (such as decision coverage or modified condition coverage) do not provide greater insights into the effectiveness of the suite. We use Clover [3], a highly reliable industrial code coverage tool, to measure statement coverage.

Generating test suites To answer RQ2, we first use the same set of test suites that were randomly generated for RQ1. In addition, we compose another set of test suites by controlling assertion coverage. We achieve this by controlling on the number of *checked statements* in a test suite. Similarly, based on the total number of checked statements in the master test suite of a program, we predefine a base number b, which indicates assertion coverage of the smallest test suite, and a step number x, which indicates assertion coverage differences between the test suites. When generating a new test suite, a hash set of the current checked statements is maintained. If the target number for checked statements is not reached, a non-duplicate test method will be randomly selected and added to the test suite. To avoid too many trials of random selection, this process is repeated until the test suite has $[b + x * i, (b + x * i) + 10]$ checked statements. This way, the i-th target

test suite has an assertion coverage of $(b + x * i)/N$, where N is the total number of statements in the source code.

Effectiveness of Assertion Types (RQ3)

RQ3 explores the effectiveness of different characteristics of test assertions. To answer this research question, we first automatically assign assertions to different categories according to their characteristics, then generate a set of sample test suites for each category of assertions, and finally conduct statistical analysis on the data collected from the generated test suites.

Assertion categorization Assertions can be classified according to their characteristics. Some of these characteristics may be potential influence factors to test suite effectiveness. We categorize assertions in three ways:

Human-written versus generated. Human-written test cases contain precise assertions written by developers about the expected program behaviour. On the other hand, automatically generated tests contain generic assertions. Commonly, it is believed that human-written test cases have a higher fault detection ability than generated assertions. We test this assumption in our work.

Assertion content type. Test assertions either check the value of primitive data type or objects of different classes. We further classify Java's primitive data types into *numbers* (for int, byte, short, long, double, and float), *strings* (for char and String), and *booleans*. This way, depending on the type of the content of an assertion, it falls into one of the following classes: *number-content-type*, *string-content-type*, *boolean-content-type*, or *object-content-type*. We explore whether these assertion content types have an impact on the effectiveness of a test suite.

For assertion content type, we apply dynamic analysis to automatically classify the assertions in a given test suite to the different categories. We first instrument test code to probe each assert statement for the type of content it asserts on. Then, we run the instrumented test code, and use the information collected to automatically assign assertions to the different content type categories.

Assertion method type. It is also possible to categorize assertions according to their actual method types. For instance, `assertTrue` and `assertFalse`, `assertEquals` and `assertNotEquals`, and `assertNull` and `assertNotNull` can be assigned to different categories. We investigate if these assertion method types have an impact on effectiveness.

For assertion method types, we parse the test code and syntactically identify and classify assertions to different assertion method type classes.

Generating test suites Under each assertion categorization, for each assertion type, we compose 50 sample test suites, each containing 100 assertions. A sample test suite contains all test methods in the master test suite, but only 100 randomly selected assertions of the target type. For instance, a sample test suite of the type *string-content-type* will contain all the test methods in the master test suite but only 100 randomly selected *string-content-type* assertions.

To quantitively compare the effectiveness between human-written and generated assertions, for each subject program, we generate (1) 50 sample test suites, each containing 100 human-written assertions from the master test suite, and (2) 50 sample test suites, each containing 100 automatically generated assertions using Randoop [36], a well-known feedback-directed test case generator for Java. We use the default settings of Randoop.

Analysis of variances For assertion content type and assertion method type, since there are multiple variables involved, we use the One-Way ANOVA (analysis of variance) statistical method to test whether there is a significant difference in test suite effectiveness between the variables. Before we conduct the ANOVA test, we used the Shapiro-Wilk test to pretest the normality of our data, and Levene's test to pretest the homogeneity of their variances. Both were positive. ANOVA answers the question whether there are significant differences in the population means. However, it does not provide any information about how they differ. Therefore, we also conduct a Tukey's Honest Significance Test to compare and rank the effectiveness of assertion types.

3.2 Results

In this section, we present the results of our experiments.

3.2.1 Effectiveness of Assertion Quantity

Ignoring test suite size Figure 3.1 depicts plots of our collected data for JFreeChart.[6] Figures 3.1a and 3.1b show that the relationship between

[6]Note that we observed a similar trend from the other subjects, and only include plots for JFreeChart due to space limitations.

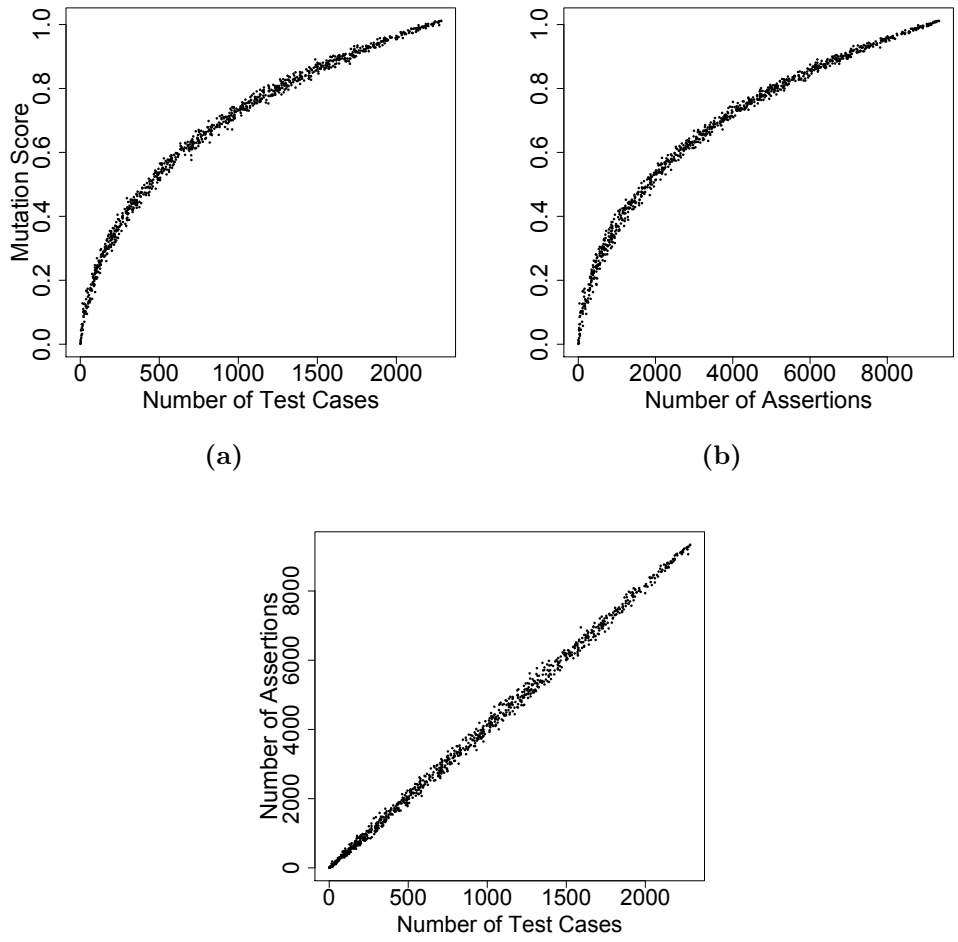


Figure 3.1: Plots of (a) suite size versus effectiveness, (b) assertion quantity versus effectiveness, and (c) suite size versus assertion quantity, for the 1000 randomly generated test suites from JFreeChart. The other four projects share a similar pattern consistently.

test suite size and effectiveness is very similar to the relationship between assertion quantity and effectiveness. As the plot in Figure 3.1c shows, there exists a linear relationship between the number of test methods and the number of assertions, in the 1000 randomly generated test suites.

Table 3.4: Correlation coefficients between test suite size and effectiveness (m), and assertion quantity and effectiveness (a). ρ_p shows Pearson correlations and ρ_k represents Kendall's correlations.

Subject ID	$\rho_p(m)$	$\rho_p(a)$	$\rho_k(m)$	$\rho_k(a)$	p-value
1	0.954	0.954	0.967	0.970	
2	0.973	0.973	0.969	0.969	
3	0.927	0.927	0.917	0.917	$< 2.2e-16$
4	0.929	0.928	0.912	0.930	
5	0.945	0.947	0.889	0.894	

Table 3.4 shows the Pearson (ρ_p) and Kendall's (ρ_k) correlations between effectiveness with respect to suite size (m) and assertion quantity a, for the test suites that are randomly generated for all the five subjects. As the table shows, there is a very strong correlation between number of assertions in a test suite and the test suite's effectiveness, and the correlation coefficients are very close to that of suite size and effectiveness. This is consistent with the plots of Figure 3.1. The correlations between assertion quantity and effectiveness are slightly higher or equal to the correlations between the number of test methods and effectiveness.

***Finding 1**: Our results indicate that, without controlling for test suite size, there is a very strong correlation between the effectiveness of a test suite and the number of assertions it contains.*

Controlling for test suite size Table 3.5 shows our results when we control for test suite size. Column 2 shows the number of assertions in the smallest test suite, and column 3 shows the difference in assertion quantity between generated test suites. Columns 3 and 4 present the Pearson and Kendall's correlations, respectively, between the assertion quantity and the effectiveness of the test suites that are generated from the five subjects by controlling test suite size. As the high correlation coefficients indicate in this table, even when test suite size is controlled for, there is a very strong correlation between effectiveness and the number of assertions.

Table 3.5: Correlations between number of assertions and suite effectiveness, when suite size is controlled for.

Subject ID	Base	Step	$\rho_p(a)$	$\rho_k(a)$	p-value
1	1,000	50	0.976	0.961	
2	100	100	0.929	0.970	
3	0	10	0.948	0.846	$< 2.2e-16$
4	100	10	0.962	0.839	
5	100	5	0.928	0.781	

***Finding 2**: Our results suggest that, there is a very strong correlation between the effectiveness of a test suite and the number of assertions it contains, when the influence of test suite size is controlled for.*

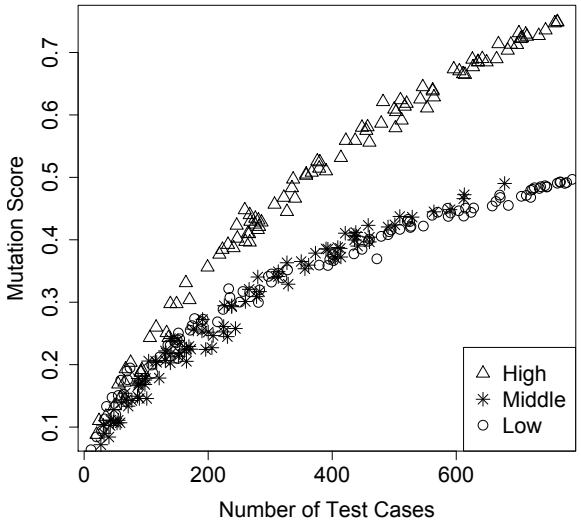


Figure 3.2: Plot of mutation score against suite size for test suites generated from assertion buckets *low*, *middle*, and bucket *high* from JFreeChart. The other four projects share a similar pattern consistently.

Controlling for assertion quantity Figure 3.2 plots the effectiveness of the test suites generated by controlling for the number of assertions. Three

buckets of high, middle and low in terms of the number of assertions were used for generating these test suites (see Section 3.1.3). From low to high, each bucket contained 762, 719, and 742 test cases in total, and the average number of assertions per test case was 0.9, 2.5, and 9.1, respectively. From the curves in the plot, we can see that the effectiveness increases as the number of test methods increase. However, comparing the curves, there is a clear upward trend in a test suite's effectiveness as its assertion quantity level increases. For every given test suite taken from the lower curve on the plot, there exits a test suite with the same suite size that has a higher effectiveness because it contains more assertions.

***Finding 3**: Our results indicate that, for the same test suite size, assertion quantity can significantly influence the effectiveness.*

3.2.2 Effectiveness of Assertion Coverage

To answer RQ2, we first computed the assertion coverage, statement coverage, and mutation score of the randomly generated test suites (see section 3.1.3). Figure 3.4a plots our results. The two fitted lines both have a very high adjusted R^2 and p-value smaller than $2.2e-16$; this indicates a very strong correlation between assertion coverage and effectiveness as well as statement coverage and effectiveness. The plot also shows that a test suite having the same assertion coverage as another test suites's statement coverage, is much more effective in detecting faults. Compared with statement coverage, assertion coverage is a more sensitive predictor of test suite effectiveness.

Figure 3.4b plots assertion coverage against number of assertions in a test suite. From the plot, assertion coverage of a test suite increases as test suite size increases. However, the increasing rate of assertion coverage decreases as test suite size increases. There is a strong increasing linear relationship between assertion coverage and test suite effectiveness. Therefore, it is expected that, test suite effectiveness increases as test suite size increases but with a diminishing increasing rate, which is again consistent with our results in section 3.2.1.

***Finding 4**: Our results suggest that, assertion coverage is very strongly correlated with test suite effectiveness. Also, ignoring the influence of assertion coverage, there is a strong correlation between statement coverage and the effectiveness.*

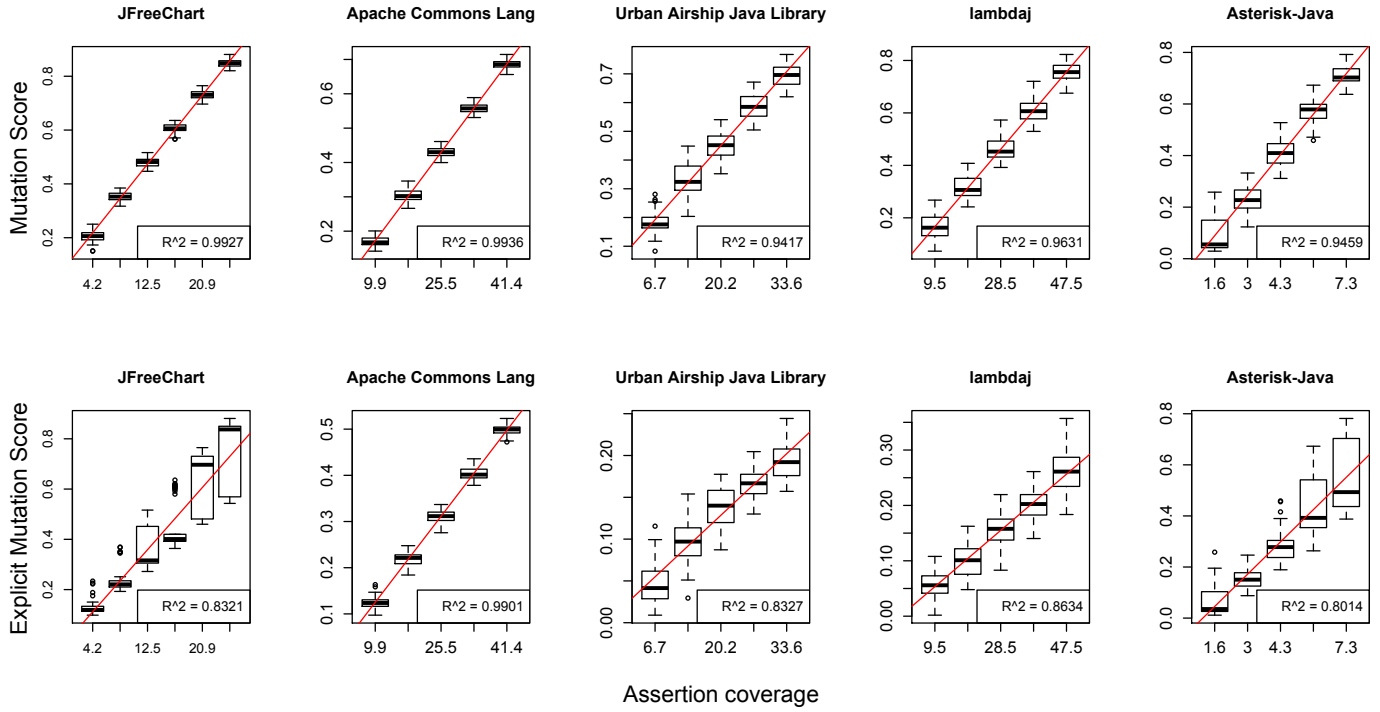
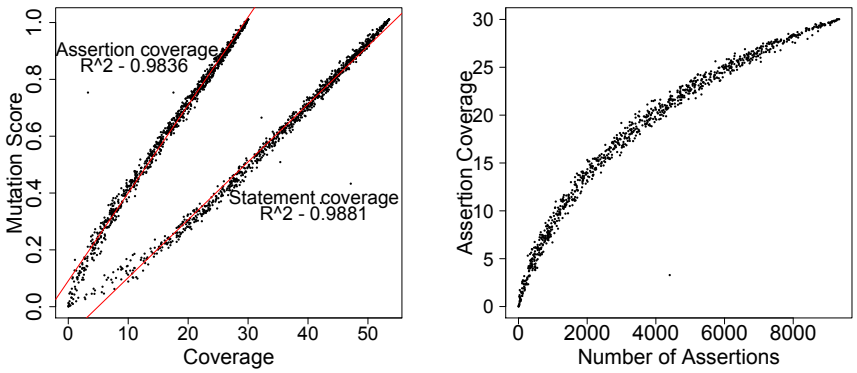


Figure 3.3: Mutation score (above) and explicit mutation score (below) plot against assertion coverage for the five subject programs. Each box represents the 50 test suites of a given assertion coverage that were generated from the original (master) test suite for each subject. The Kendall's correlation is 0.88–0.91 between assertion coverage and mutation score, and 0.80–0.90 between assertion coverage and explicit mutation score.

Table 3.6: Statistics of test suites composed at different assertion coverage levels.

Subject ID	Assertion Coverage	Stat. Coverage Corr.		Mutation Score		Statement Coverage
		$\rho_{general}$	$\rho_{explicit}$	*general*	*explicit*	
1	4.2%	0.62	0.17	0.21	0.13	15%
	8.4%	0.60	0.22	0.35	0.24	23%
	12.5%	0.59	0.11	0.48	0.35	30%
	16.7%	0.63	0.33	0.61	0.45	36%
	20.9%	0.58	0.26	0.73	0.61	41%
	25.1%	0.71	0.32	0.85	0.75	47%
2	9.9%	0.67	0.49	0.17	0.12	21%
	17.7%	0.67	0.51	0.30	0.22	34%
	25.5%	0.65	0.48	0.43	0.31	46%
	33.7%	0.66	0.50	0.56	0.40	57%
	41.4%	0.58	0.27	0.69	0.50	68%
3	6.7%	0.62	0.06	0.18	0.05	19%
	13.5%	0.74	0.06	0.33	0.10	30%
	20.2%	0.76	0.01	0.46	0.14	39%
	26.9%	0.75	0.07	0.59	0.17	48%
	33.6%	0.76	0.05	0.70	0.20	55%
4	9.5%	0.76	0.21	0.17	0.06	19%
	19.0%	0.73	0.33	0.31	0.10	32%
	28.5%	0.70	0.30	0.46	0.15	45%
	38.0%	0.63	0.23	0.61	0.20	58%
	47.5%	0.50	0.10	0.76	0.26	70%
5	1.6%	0.73	0.63	0.10	0.06	4%
	3.0%	0.76	0.35	0.23	0.15	8%
	4.3%	0.70	0.38	0.41	0.28	12%
	5.8%	0.60	0.25	0.57	0.43	16%
	7.3%	0.62	0.24	0.71	0.56	19%



(a) (b)

Figure 3.4: Plots of (a) mutation score against assertion coverage and statement coverage, (b) assertion coverage against assertion quantity, for the 1000 randomly generated test suites from JFreeChart.

Controlling for assertion coverage Figure 3.3 shows box plots of our results for the test suites generated by controlling their assertion coverage. The adjusted R^2 value for each regression line is shown in the bottom right corner of each box plot. It ranges from 0.94 to 0.99 between assertion coverage and mutation score, and 0.80 to 0.99 between assertion coverage and explicit mutation score. This indicates assertion coverage can predict both mutation score and explicit mutation score well.

Table 3.6 summarizes statistics for these test suites. Column 3 contains the Kendall's correlations between statement coverage and mutation score (0.50–0.76), column 4 presents the Kendall's correlations between statement coverage and explicit mutation score (0.01–0.63). When assertion coverage is controlled for, there is a *moderate to strong* correlation between statement coverage and mutation score, and only a *low to moderate* correlation between statement coverage and explicit mutation score. For instance, only about 1/3 of the mutants generated for Urban Airship Library (ID 5) and lambdaj (ID 4) are explicitly detectable mutants; correspondingly there is only a weak correlation (0.01–0.33) between their statement coverage and explicit mutation score. A higher fraction ($\approx$ 2/3) of the mutants generated for the other three subjects are explicitly detectable mutants, and thus the correlation between their statement coverage and explicit mutation score

increases significantly (from 0.11 to 0.63).

Columns 5–7 in Table 3.6 pertain to the average mutation score, average explicit mutation score, and average statement coverage of the test suites at each assertion coverage level, respectively. As the results show, a slight increase in assertion coverage can lead to an obvious increase in the mutation score and explicit mutation score. For instance, for JFreeChart (ID 1), when assertion coverage increases by around 4%, the mutation score increases by around 12.4% and explicit mutation score increases by around 11%. On the other hand, a 4% increase in the statement coverage does not always increase either mutation score or explicit mutation score. This shows again that assertion coverage is a more sensitive indicator of test suite effectiveness, compared to statement coverage.

***Finding 5**: Our results suggest that, assertion coverage is capable of predicting both mutation score and explicit mutation score. With assertion coverage controlled for, there is only a moderate to strong correlation between statement coverage and mutation score, and a low to moderate correlation between statement coverage and explicit mutation score. Test suite effectiveness is more sensitive to assertion coverage than statement coverage.*

3.2.3 Effectiveness of Assertion Types

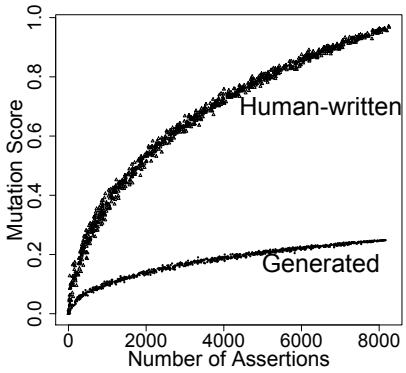
Initial Study

To answer RQ3, we first examined the 9,177 assertions of JFreeChart.

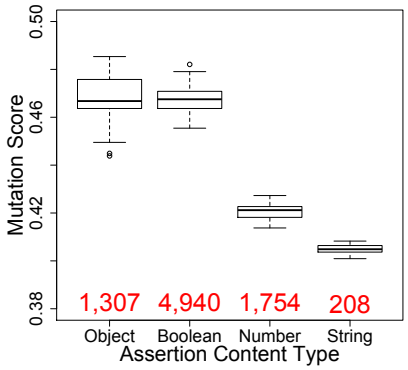
Assertion generation strategy Figure 3.5a plots the effectiveness of human-written test suites and Randoop generated test suites against assertion quantity. As we can observe, the effectiveness of human-written and generated test suites both increase as the assertion quantity increases. However, the effectiveness of the generated test suites gets saturated much faster than human-written test suites.

From our observations of the composed test suites, the 50 human-written sample test suites are effective in killing mutants, while the 50 generated test suites can hardly detect any mutant. We increased the assertion quantity in the sample test suites to 500, but still saw the same pattern.

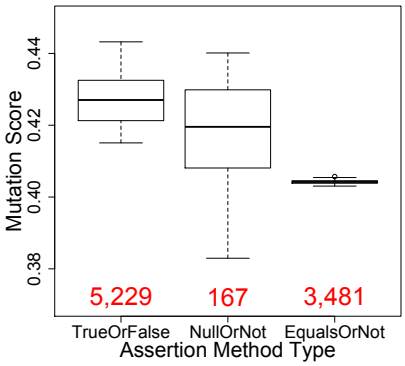
***Finding 6**: Our results indicate that, human-written test assertions are far more effective than automatically generated test assertions.*



(a)



(b)



(c)

Figure 3.5: Plots of (a) assertion quantity versus effectiveness of human-written and generated tests, (b) assertion content types versus effectiveness, and (c) assertion method types versus effectiveness. In (b) and (c), each box represents the 50 sample test suites generated for each type; the total number of assertions of each type are indicated in red.

Assertion content type Assertions are also classified based on the types of the content they assert on. Figure 3.5b box plots the effectiveness of the sample test suites that exclusively contain assertions on object, boolean, number, or string types. Tables 3.7 and 3.8 show the ANOVA and the Tukey's Honest Significance test, respectively. The F value is 1544 with a p-value

Table 3.7: One-Way ANOVA on the effectiveness of assertion content types and actual assertion types.

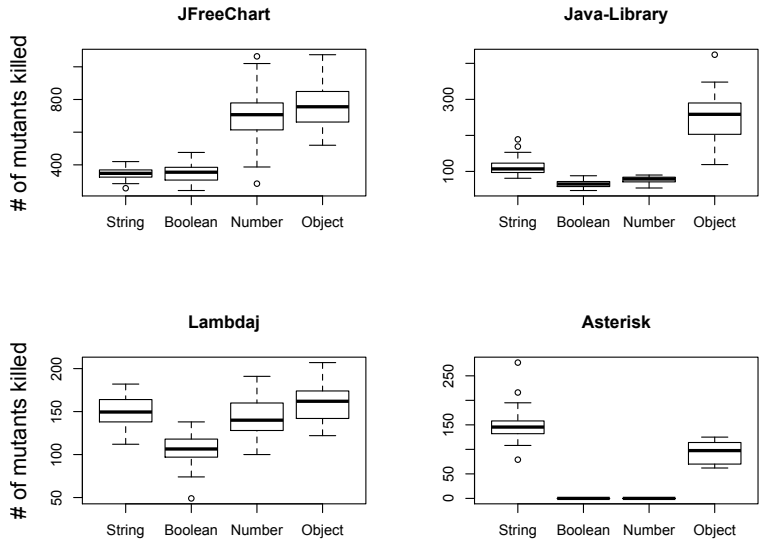
	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Assertion Content Types					
Type	3	0.15675	0.05225	1544	<2e-16
Residuals	196	0.00663	0.00003		
Assertion Method Types					
Type	2	0.01398	0.006988	87.87	<2e-16
Residuals	147	0.01169	0.000080		

very close to 0, thus we can confidently reject the null hypothesis of equal variances (effectiveness) for the four assertion content types. Table 3.8 shows the estimated difference in mutation score in column 2, and the 95% confidence interval of the difference in columns 3 and 4. The Tukey's test indicates that there is a significant difference between the effectiveness of assertions that assert on boolean/object, string, and number types. Assertions that assert on boolean types are as effective as assertions that assert on objects.

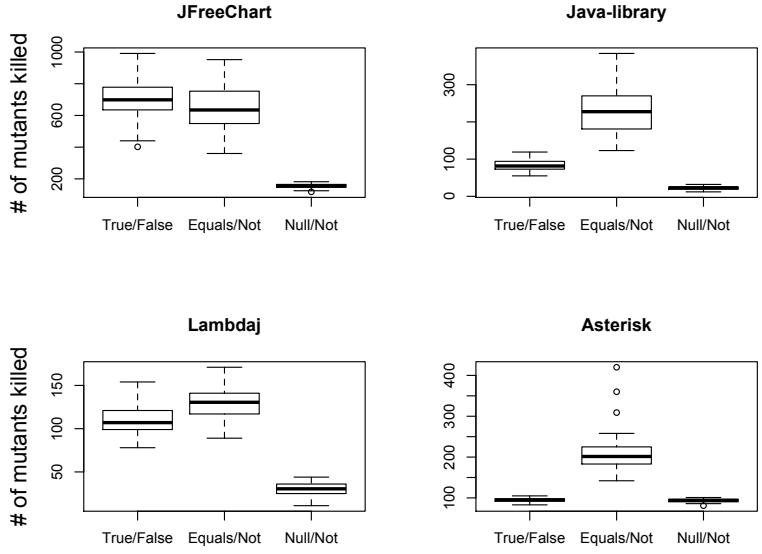
Assertion method type Assertions can also be classified by their actual method types. Figure 3.5c plots the effectiveness of the sample test suites that belong to the three assertion method types. In this chapter, we did not study *assertSame* and *assertNotSame*, because there were only 27 of them in JFreeChart, which is a low number to be representative. The bottom half of tables 3.7 and 3.8, present the ANOVA and Tukey's Honest Significance test, respectively, for assertion method types. The F value is 87.87 with a p-value very close to 0, thus we can reject the null hypothesis of equal variances (effectiveness) for the three assertion method types. The Tukey's test shows that there exists a significant difference between the effectiveness of the three assertion method types.

Followup Study

One year after we conducted the initial study, we conducted a followup study on assertion content type and method type. We examined the 17182 assertions of JFreeChart, Urban Airship Java Library, lambdaj, and Asterisk-Java on their most up to date versions.



(a)



(b)

Figure 3.6: Plots of (a) number of mutants killed versus assertion content types, (b) number of mutants killed versus assertion method types. Each box represents the 50 sample test suites generated for each assertion type.

Table 3.8: Tukey's Honest Significance Test on the effectiveness of assertion content types and assertion method types. Each of the sample test suites used for the comparison contains 100 assertions of a target type.

Types	diff	lwr	upr	p adj
Assertion Content Types				
Boolean vs. Object	-0.0002	-0.0032	0.0028	0.9985
Number vs. Boolean	-0.0470	-0.0500	-0.0440	0.0000
String vs. Number	-0.0156	-0.0186	-0.0126	0.0000
Assertion Method Types				
assertNull/Not vs. assertTrue/False	-0.0103	-0.0145	-0.0060	1e-07
assertEquals/Not vs. assertNull/Not	-0.0133	-0.0175	-0.0091	0e+00

Assertion content type

Figure 3.6a box plots the effectiveness of the sample test suites that exclusively contain assertions on object, boolean, number, or string types. Test suites with *object-content-type* assertions are the most effective in JFreeChart, Java-library, and Lambdaj, and second effective in Asterisk. Test suites with *string-content-type* assertions test suites are least effective in JFreeChart, but most effective in Asterisk. The results for JFreeChart is different from the initial study. Therefore, there is not a consistent pattern between the effectiveness of assertions assert on different content types.

Table 4.3 shows the One-Way ANOVA test on the effectiveness of assertion content types for project JFreeChart. The F value is 0.128 with P value equals to 0.943 (close to 1) tells there is not a significant difference between the effectiveness of different content types of assertions. We receive a same message from the tests conducted on the rest three subjects.

***Finding 7**: There is not a consistent ranking nor a statistical significant difference between assertion content types in terms of their test effectiveness.*

Table 3.9: One-Way ANOVA on the effectiveness of actual assertion types and assertion content types on JFreeChart.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Assertion Content Types					
Type	3	18883	6294	0.128	0.943
Residuals	196	9603200	48996		
Assertion Method Types					
Type	2	659	330	0.005	0.995
Residuals	147	10721504	72935		

Assertion method type

Figure 3.6b plots the effectiveness of the sample test suites that belong to the three assertion method types. Test suites with `assertTrue/False` are most effective in JFreeChart, second effective in Java-library and Lambdaj, but least effective in Asterisk. Test suites with `assertEquals/Not` are most effective in all of the projects except for second effective in JFreeChart. Test suites with `assertNull/Not` are least effective in all of the projects except for second least effective in Asterisk. The results for JFreeChart is also different from the initial study. Therefore, there is not a consistent ranking in the effectiveness of assertions of different method types.

Table 4.3 shows the One-Way ANOVA test on the effectiveness of assertion method types for project JFreeChart. The F value is 0.005 with P value equals to 0.995 (close to 1) tells there is not a significant difference between the effectiveness of different content types of assertions. We get a similar test result for the rest three subjects.

***Finding 8**: There is not a consistent ranking nor a statistical significant difference between assertion method types in terms of their test effectiveness.*

Chapter 4

Fault Type and Location Influence Measuring Test Suite Effectiveness

Test suite effectiveness is measured by assessing the portion of faults that can be detected by tests. To precisely measure a test suite's effectiveness, one needs to pay attention to both tests and the set of faults used to measure effectiveness. Instead of only focusing on the testing side, we propose to investigate test suite effectiveness also from fault types (the ways faults are generated) and fault location. We empirically evaluate the relationship between test suite effectiveness, assertions, and faults based on 17,182 assertions and 18,820 artificial faults generated from four real-world Java projects. Our results indicate that fault type and statement type where the fault is located can significantly influence a test suite's effectiveness. Assessing test suite effectiveness without paying attention to the type and distribution of faults can provide misleading results.

4.1 Experimental Design

Our goal in this study is to answer the following research questions through controlled experiments:

RQ1 Is measuring test effectiveness influenced by fault type?
RQ2 Is measuring test effectiveness influenced by fault location?

4.1.1 Subject Programs

We select four subject programs, which were also used in the previous chapter. We conduct our experiments on the latest versions of these subjects. The characteristics of these subject programs are summarized in Table 4.1. Column version contains the version number and Github [5] commit id of the subjects. Lines of source code and test code are measured using SLOCCount [11].

The table also shows the total number of test assertions, and the number of mutants generated, for each subject program.

Table 4.1: Characteristics of the subject programs.

ID	Subjects	Version	Java SLOC	Test SLOC	Assertions	Mutants
1	JFreeChart [8]	jfreechart-1.0.19	168,777	41,382	11,915	13,744
2	Urban Airship Java Library [12]	96cedd21fbe37ddbc555008c353c3a8736fda0e3	35,105	11,516	1,935	2,719
3	lambdaj [9]	bd3afc7c084c3910454a793a872b0a76f92a43fd	19,446	4,872	2,674	1,308
4	Asterisk-Java [2]	5e9b16f2816cf5e6d6c6fa81e924ccdf3ead197f	36,530	4,243	658	1,049
Total/Average			329,600	103,314	17,182	18,820

Table 4.2: Default mutation operators provided by PIT.

ID	Mutation operator	Definition
CBM	Conditionals Boundary Mutator	replaces the relational operators <, <=, >, >=.
IM	Increments Mutator	mutate increments, decrements and assignment increments and decrements of local variables (stack variables).
INM	Invert Negatives Mutator	inverts negation of integer and floating point numbers.
MM	Math Mutator	replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation
NCM	Negate Conditionals Mutator	mutate all conditionals ==, !=, <=, >=, <, >.
RVM	Return Values Mutator	mutates the return values of method calls.
VMC	Void Method Calls Mutator	removes method calls to void methods.

4.1.2 Procedure

Fault Types (RQ1)

RQ1 explores how easy different types of faults can be detected by assertions. To answer this question, we first observe the distribution of mutants generated by different mutation operators. Then, we automatically sample subsets of mutants for each type, with size controlled for, and statistically compare the number of detectable mutants in the subsets of the original test suite. Finally, we remove each type of mutant separately, and examine if they cause any significant difference in measuring test effectiveness.

Simulating fault types We use mutants to simulate real faults. Mutants have been widely adopted to substitute real faults in the literature [18, 27, 28, 33, 37, 39]. There is also empirical evidence for mutants being representable for real faults [14, 15, 19, 29]. In mutation testing, mutation operators are used to simulate different types of programming errors. They define the rules of how mutants are constructed from the original program. Therefore, we assign mutants to different fault types according to their mutation operators. Again, we use PIT [10] to generate mutants. We use the (seven) default mutation operators provided by PIT, shown in table 4.2.

Assertions vs. detectable mutants matrix To speed up our experimentation, it is necessary to construct a matrix, which maps from assertions (and crashing statements) to their detected mutants, or vice versa. We noticed (1) PIT stops executing the rest of the test cases after a test case first detects a mutant, (2) PIT stops executing the rest of statements after a statement crashes the program, and (3) PIT does not provide a fine grained mapping between mutants generated and test cases, nor test assertions. In other word, PIT does not provide the functionality to create such a matrix. We extended PIT to construct this matrix. We first modified PIT so that it always executes all test cases even after any test case fails. We then instrument the test suite by surrounding its test statements with JUnit ErrorCollector [4]. The ErrorCollector allows the execution of a test to continue even after a test failure. This way, we record failure information during the execution of all tests against each mutant.

Subsets of mutants Some mutation operators generate more mutants than others. Also, a mutation operator may generate more mutants for a different program. To quantitatively compare the mutants generated by different

mutation operators, we randomly sample subsets of mutants generated by each mutation operator by controlling on the mutant quantity. For each subject program, we randomly select 50 subsets of mutants generated by each mutation operator. Each subset contains 10 randomly selected mutants of the target type without replacement. We picked 10 according to the least number of mutants generated by the mutation operators, so that each subset does not contain too many duplicate mutants.

Omitting mutation operator In addition to comparing between mutants generated by each of the mutation operators, we compare between mutants generated by all of the mutation operators and mutants generated by omitting each mutation operator, separately. To achieve this goal, we first sample 50 subsets of mutants of size 100 from the mutants generated by all mutation operators. Then, we leave out each of the mutation operators in turn, and sample 50 subsets of mutants of size 100 from the mutants generated by the rest of mutation operators. We evaluate if there is any statistically significant influence in test effectiveness when a mutation operator is missing.

Fault Location: Types of Statement Mutated (RQ2)

RQ2 studies whether fault location influences the test effectiveness measurement. To answer this question, we examine if the statement type where a program is mutated affects how easy a mutant can be detected. We start with observing the distribution of mutants in different types of statements. Then, we correlate the distribution with the mutation score of the subjects.

Types of statement Mutation happens in different types of statements. To simplify our experimental design, we classify program statements as follows:

1. ***Conditional statements*** can change the control flow of the program and include `if-else`, `for`, and `while` constructs.
2. ***Return statements*** are statements with `return` keyword.
3. ***Statements*** are normal statements that are neither conditional statements nor return statements.

Mutants can also be located in nested statements. For example, a statement can be nested inside an `if-else` statement. Therefore, we also classify mutants based on their level of nesting.

Table 4.3: One-Way ANOVA on the effectiveness of actual assertion types and assertion content types on JFreeChart.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Assertion Content Types					
Type	3	18883	6294	0.128	0.943
Residuals	196	9603200	48996		
Assertion Method Types					
Type	2	659	330	0.005	0.995
Residuals	147	10721504	72935		

Subsets of mutants Different number of mutants can be generated in different types of statements. Thus, as we did for RQ1, we compose subsets of mutants in different types of statements by controlling on the mutants quantity. For each subject program, we randomly sample 50 subsets of mutants which are generated by mutating a program in each statement type. Each subset contains 100 randomly selected mutant, without replacement. We pick 100 according to the total number of mutants generated in each type of statement for the subjects.

4.2 Results

4.2.1 Fault Type (RQ1)

Distribution of fault types

Figure 4.1 shows bar charts of the distribution of the number of mutable locations, number of mutants generated, and number of mutants detected by different mutation operators. Note that we did not include the mutation operator *Invert Negatives Mutator* in this figure, since it generates very few mutants in the subject programs. Each stacked bar as a whole illustrates the number of mutable locations of the mutation operator in a program's production code. It also shows the number of mutants actually generated and detected, separately. The correlation scores between the number of mutable locations and mutants generated are listed on the top left conner of the bar charts. The figure shows *Return Values Mutator*, *Negate Conditionals Mutator*, and *Void Method Calls Mutator* always generate more mutants than *Conditional boundary Mutator*, *Increments Mutator*, and *Math Mutator*. And the distribution of mutants generated by the mutation operators aligns with the distribution of possible mutable locations of the operators. The correlation scores range betwee [0.9801–0.6559], which indicates that the number of mutants generated are strongly to very strongly correlated with

the number of mutable locations.

***Finding 9**: Our results show that, different mutation operators generate different number of mutants, which largely depends on the number of mutable locations present in the source code.*

Controlling mutant type and quantity

To examine how easy mutants generated by different mutation operators can be detected, we control mutant type by sampling subsets of mutants generated by each mutation operator, one at a time. Each subset contains a fixed number of mutants. We observe the number of mutants that can be detected in the subsets by the original test suite. We performed one-way ANOVA tests of equal mean on the number of detectable mutants between the six mutation operators. The F values were 466.9, 944.3, 830.4, 135.5, respectively with p-values close to 0. Thus, we can confidently reject the equal mean hypothesis and conclude that mutants generated by different mutation operators are not at the same level of easiness to be detected.

We further pair-wisely compare the number of detectable mutants between the mutation operators by conducting a Tukey's Honest Significance Test. The results show some significant and consistent patterns within some of the fault type pairs. Table 4.5 illustrates these patterns. Column *diff* shows the estimated difference in the number of detectable mutants. Column *lwr* and *upr* show the 95% confidence interval of the difference. The p-values are very close to zero, which indicate the patterns we found are significant. From the table, we can conclude that mutants generated by *Increments Mutator* are easier to be detected than mutants generated by *Conditionals Boundary Mutator*, *Negate Conditionals Mutator* is easier than *Conditionals Boundary Mutator*, *Return Values Mutator* is easier than *Conditionals Boundary Mutator*, *Negate Conditionals Mutator* is easier than *Math Mutator*, *Math Mutator* is harder than *Increments Mutator*, and *Void Method Calls Mutator* is harder than *Return Values Mutator*. All the other pairwise comparisons either show inconsistent or non-significant results.

***Finding 10**: Our results indicate that, there is a significant difference between how easy mutants generated by different mutation operators can be detected.*

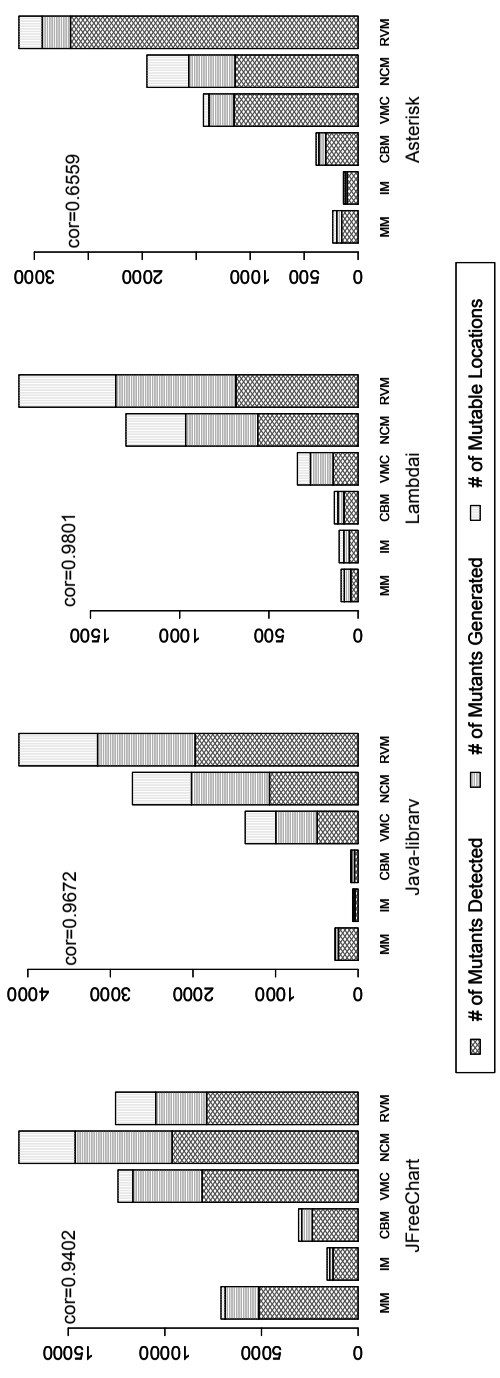


Figure 4.1: Plots of number of mutable locations, number of mutants generated, and number of mutants detected by all of the mutation operators.

Table 4.4: One-Way ANOVA on the number of mutants killed. Each of the sample subsets of mutants used for the comparison contains 10 mutants generated by a specific mutation operator.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
JFreeChart					
Operator	5	4207	841.4	466.9	<2e-16
Residuals	594	1070	1.8		
Java-library					
Operator	5	5549	1109.7	944.3	<2e-16
Residuals	594	698	1.2		
Asterisk					
Operator	5	4863	972.6	830.4	<2e-16
Residuals	594	696	1.2		
Lambdaj					
Operator	5	1103.7	220.74	135.5	<2e-16
Residuals	594	967.6	1.63		

Omitting mutation operators. We explored whether omitting any single mutation operator can lead to a significant difference in measuring test effectiveness.

Our statistical analysis shows that omitting a single mutation operator does not always cause a significant difference in test effectiveness. For example, without using *Math Mutator*, there exists a significant difference in test effectiveness for subject 1 (p-value equals to 0), but no significant difference in the rest of the subjects. It is similar for *Void Method Calls Mutator*, *Conditionals Boundary Mutator* and *Negate Conditionals Mutator*. However, there always exists a significant loss in test effectiveness if omitting *Return Values Mutator*. Without using *Increments Mutator* and *Invert Negatives Mutator*, there is no significant change in test effectiveness.

Combining this finding with Figure 4.1, which illustrates the distribution of number of mutants generated by the mutation operators, we can see that *Increments Mutator* and *Invert Negatives Mutator* always generate the least number of mutants in the subjects. The number of neglect mutants may not be significant enough compared to the total number of mutants generated to influence measuring test suite effectiveness. *Return Values Mutator* generates the most number of mutants in Java-library, Lambdaj, and Asterisk, and second most number of mutants in JFreeChart. It always generates relatively more mutants than other mutation operators. Therefore, omitting *Return Values Mutator* is likely to reveal the influence on measuring test effectiveness if there is any difference between mutants generated by *Return Values Mutator* and other mutation operators. We observed that omitting *Return Values*

Table 4.5: Tukey's Honest Significance Test on the number of mutants killed. Each of the sample subsets of mutants used for the comparison contains 10 mutants generated by a specific mutation operator..

diff	lwr	upr	p adj
IM-CBM			
4.74	4.197245	5.282755	0
4.40	3.96167569	4.8383243	0
6.35	5.9124172	6.78758276	0
2.36	1.8439404	2.8760596	0
NCM-CBM			
2.90	2.357245	3.442755	0
4.63	4.19167569	5.0683243	0
5.18	4.7424172	5.61758276	0
2.32	1.8039404	2.8360596	0
RVM-CBM			
4.94	4.397245	5.482755	0
4.92	4.48167569	5.3583243	0
4.66	4.2224172	5.09758276	0
2.05	1.5339404	2.5660596	0
MM-IM			
-6.49	-7.032755	-5.947245	0
-7.55	-7.98832431	-7.1116757	0
-1.77	-2.2075828	-1.33241724	0
-3.43	-3.9460596	-2.9139404	0
NCM-MM			
4.65	4.107245	5.192755	0
7.78	7.34167569	8.2183243	0
0.60	0.1624172	1.03758276	0.0015
3.39	2.8739404	3.9060596	0
VMC-RVM			
-5.71	-6.252755	-5.167245	0
-0.60	-1.03832431	-0.1616757	0.0014
-2.09	-2.6060596	-1.5739404	0

Mutator always causes a significant loss in test effectiveness, which potentially indicates that *Return Values Mutator* may generate easy to detect mutants. Intuitively, *Return Values Mutator* always mutates return statements.

***Finding 11**: Omitting certain mutation operators, such as 'Return Values Mutator', always cause a significant loss in the measured test effectiveness.*

This finding gives us the insight that there may be a difference in measuring test effectiveness when mutations are located in different types of statements.

4.2.2 Fault Location (RQ2)

Table 4.6: Tukey's Honest Significance Test on the number of mutants killed between using all mutation operators and omitting each mutation operator in turn. Each of the sample subsets contains 100 mutants either generated by all mutation operators or leaving out each operator in turn.

Subject	diff	lwr	upr	p adj
Without ReturnValsMutator				
1	-8.22	-9.4971	-6.9429	0
2	-3.64	-4.832444	-2.447556	0
3	-3.89	-5.051185	-2.728815	0
4	-4.56	-5.881209	-3.238791	0
Without MathMutator				
1	4.25	2.985603	5.514397	0
2	1.23	0.05409759	2.405902	0.0404424
3	-0.05	-1.25359	1.15359	0.9347913
4	-0.67	-1.932948	0.5929484	0.2967606
Without VoidMethodCallMutator				
1	8.15	6.959959	9.340041	0
2	0.82	-0.3725293	2.012529	0.1766477
3	1.52	0.3547874	2.685213	0.0108299
4	13.3	12.18597	14.41403	0
Without ConditionalsBoundaryMutator				
1	0.96	-0.2902678	2.210268	0.1315736
2	0.97	-0.1547487	2.094749	0.0905699
3	0.05	-1.137623	1.237623	0.9339165
4	2.22	1.061811	3.378189	0.0002076
Without IncrementsMutator				
1	-0.88	-2.200685	0.4406852	0.1903678
2	0.41	0.7976888	1.617689	0.5039673
3	-1.06	-2.170466	0.0504664	0.0612489
4	-0.24	-1.386161	0.9061614	0.6801049
Without NegateConditionalsMutator				
1	-7.35	-8.735988	-5.964012	0
2	0.95	-0.2667233	2.166723	0.1252245
3	-13.37	-14.60836	-12.13164	0
4	-3.01	-4.160555	-1.859445	6e-07
Without InvertNegsMutator				
1	-0.69	-1.957531	0.5775308	0.2843542
2	0.59	-0.5303536	1.710354	0.3003026
3	-0.35	-1.443749	0.7437485	0.5287379
4	-0.45	-1.690054	0.7900544	0.475069

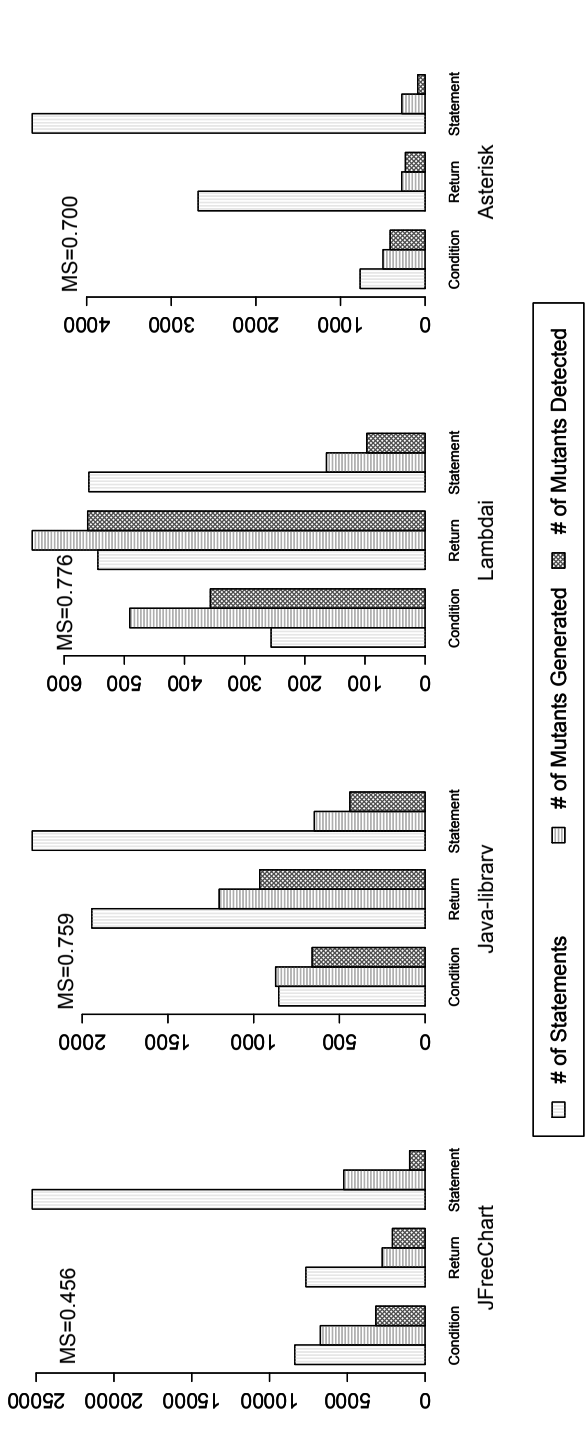


Figure 4.2: Plots of number of different types of statements, number of mutants generated by mutating different types of statements, and number of mutants detected from the generated mutants.

Distribution of mutants in different types of statements Figure 4.2 summarizes the statistics of mutants located in different types of statements in the source code. In each bar chart, a group of three bars, from left to right, represents (1) number of the type of statements, (2) number of mutants located in that type of statements, and (3) number of detected mutants from (2). The top left corner of each bar chart lists the overall mutation score of the subject program.

The distribution of number of different types of statements varies between subjects. For example, there are less return statements than conditional statements in JFreeChart, but more return statements than conditional statements in the rest three subjects. The distribution of mutants located in different types of statements is also subject dependent. However, a larger portion of return and conditional statements is always mutated compared to normal statements.

In addition, the number of mutants generated in conditional statements is larger than the number of conditional statements in Java-library and Lambdaj. The number of mutants generated in return statements is larger than the number of return statements in Lambdaj as well. The observation indicates that a statement in a program can be mutated more than once and in different ways.

***Finding 12**: Normal statements are less likely to be mutated compared to return statements and conditional statements. A program statement may be mutated more than once in different ways.*

Compare across subjects In Figure 4.3, for each type of statement, the mutation score is plotted against the percentage of mutants located in the type of statement, for each subject program separately. The distribution of mutants in different types of statements has a strong to very strong correlation with the mutation score of the subjects. The correlation between the mutation score and the ratio of mutants is very strong if a program is mutated in return statement (0.86) and normal statement (-0.87), and strong if a program is mutated in conditional statement (-0.62). The negative correlation indicates a relationship between two variables in which one variable increases as the other decreases, and vice versa. For instance, the mutation score will increase as the ratio of mutants in normal and conditional statement decreases, and vice versa.

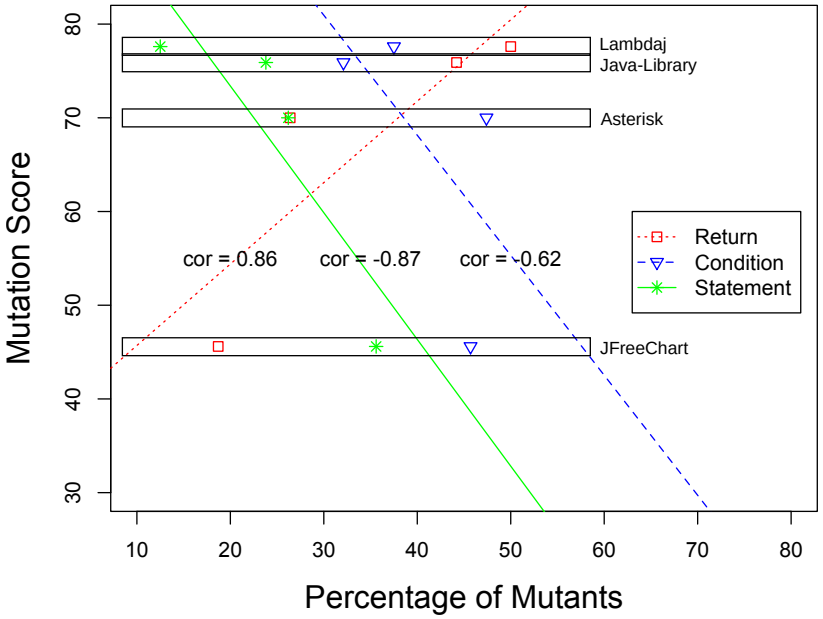


Figure 4.3: Plot of mutation score against percentage of mutants generated in different types of statements. For each type of statement, a data point represents one of the for subject programs.

***Finding 13**: The percentage of mutants generated by mutating different types of statements is strongly to very strongly correlated with test suite effectiveness.*

Compare within subjects

We compared the number of detectable mutants in the subsets, by controlling on statement type and mutants quantity. Figure 4.4 shows box-plots of our results. From the plots, we can observe a clear decreasing trend of number of detectable mutants by the original test suite located in: return statements, conditional statements, and normal statements. This result is consistent with our findings in Section 4.2.1 when comparing between subjects.

We performed ANOVA tests of equal mean for the number of detectable mutants located in different type of statements. The F values were 4659.6,

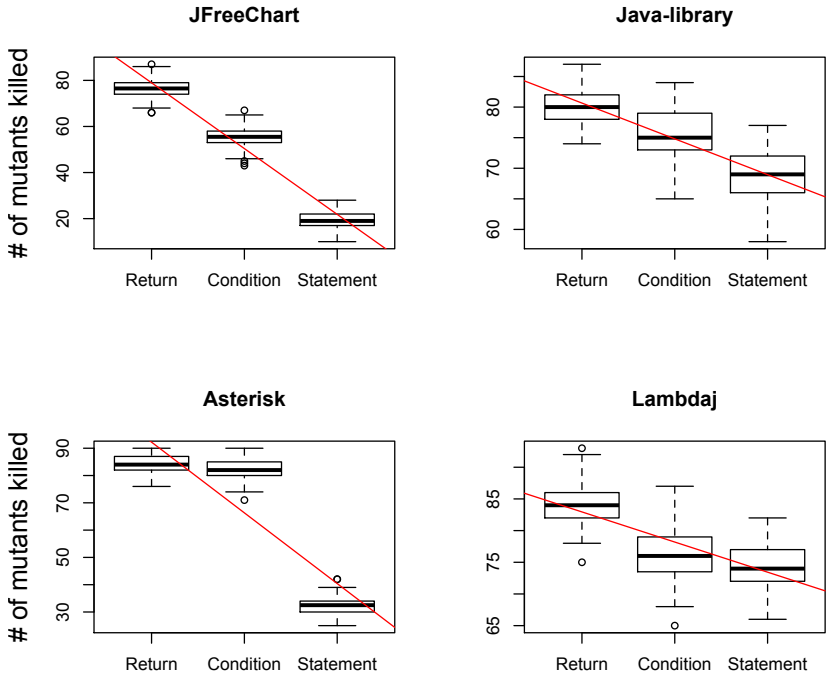


Figure 4.4: A box-plot of number of mutants killed at different mutation locations : return statements, condition statements, and normal statements. Each box represents the 100 subsets of mutants that were randomly selected from all mutants generated at the location for each subject. Each subset contains 100 mutants.

241.35, 7936.5 and 195.07 respectively for the subjects with p-value very close to 0. Thus we can confidently reject the null hypothesis of equal variance (of number of detectable mutants) for the four subjects. Next we applied Tukey's Honest Significance test of pair-wisely comparing between mutants located in different type of statements. The test estimated the pairwise difference between number of killed mutants in different type of statements. The test result is consistent with what we have observed in the box-plot in figure 4.4.

***Finding 14**: Our results indicate that, the types of statements mutated significantly influence test suite effectiveness when comparing within each subject. Mutants in return statements are easiest to be killed. Mutants in condition statements are easier to be killed then those in normal statements.*

Table 4.7: Tukey's Honest Significance Test on the number of mutants killed. Each of the sample subsets of mutants used for the comparison contains 100 mutants.

Types	diff	lwr	upr	p adj
JFreeChart				
Return-Condition	20.80	19.38946	22.21054	0
Statement-Condition	-36.31	-37.72054	-34.89946	0
Statement-Return	-57.11	-58.52054	-55.69946	0
Java-library				
Return-Condition	4.69	3.427404	5.952596	0
Statement-Condition	-7.01	-8.272596	-5.747404	0
Statement-Return	-11.70	-12.962596	-10.437404	0
Asterisk				
Return-Condition	1.65	0.5488635	2.751137	0.0014
Statement-Condition	-50.16	-51.2611365	-49.058863	0
Statement-Return	-51.81	-52.9111365	-50.708863	0
Lambdaj				
Return-Condition	7.76	6.552742	8.9672583	0
Statement-Condition	-1.75	-2.957258	-0.5427417	0.0021
Statement-Return	-9.51	-10.717258	-8.3027417	0

Table 4.8: Distribution of mutation location for each mutation operator.

Mutation Operators	Java-Library						Asterisk					
	Con	%	*Ret*	%	*Smt*	%	*Con*	%	*Ret*	%	*Smt*	%
ReturnValsMutator	39	3	1143	97	0	0	2	0.8	259	99.2	0	0
VoidMethodCallMutator	26	5.2	0	0	475	94.8	2	0.9	0	0	231	99.1
NegateConditionalsMutator	763	80.7	53	5.6	129	13.7	402	93.5	13	3.0	15	3.5
MathMutator	0	0	2	5.3	36	94.7	14	31.1	4	8.9	27	60.0
ConditionalsBoundaryMutator	30	83.3	3	8.3	3	8.3	61	96.8	1	1.6	1	1.6
IncrementsMutator	14	82.4	0	0	3	17.6	16	94.1	0	0	1	5.9
InvertNegsMutator	0	0	0	0	0	0	0	0	0	0	0	0
	JFreeChart						**Lambdaj**					
ReturnValsMutator	18	0.7	2600	98.7	17	0.65	119	17.7	555	82.3	0	
VoidMethodCallMutator	171	4.8	0	0	3395	95.2	11	8.7	0	0	116	91.3
NegateConditionalsMutator	4818	86.1	26	0.5	167	3.3	293	72.5	81	20.0	30	7.4
MathMutator	54	3.1	116	6.7	1569	90.2	14	38.9	10	27.8	12	33.3
ConditionalsBoundaryMutator	517	93.3	7	1.3	30	5.4	29	85.3	5	14.7	0	0
IncrementsMutator	149	86.6	0	0	23	13.4	25	80.6	0	0	6	19.4
InvertNegsMutator	5	11.1	7	15.6	33	73.3	0	0	2	100	0	0

Statement type vs. mutation operator Table 4.8 summarizes the number of mutants generated in different types of statements by each mutation operator. The first column indicates which mutation operator is responsible for generating the mutants. Conlumn *Con* contains number of mutants generated in conditional statements, *Ret* shows number of mutants generated in return statements, and *Smt* presents number of mutants generated in normal statements. Column *%* represents the ratio of mutants generated in the type of statements as stated in the previous column by the operator. For example, for subject JFreeChart, 18 mutants (0.7 %) are generated in conditional statements, 2600 (98.7 %) in return statements, and 17 (0.65 %) in normal statements by the *Return Values Mutator* operator.

From the table, we can observe that most of the mutants generated by *Return Values Mutator* are in a return statement (82.3–99.2%). Mutants generated by *Void Method Calls Mutator* are always in normal statements (94.8–99.1%). *Negate Conditionals Mutator*, *Conditionals Boundary Mutator*, and *Increments Mutator* usually mutate conditional statements (81–93.5%). However, *Math Mutator* and *Invert Negatives Mutator* do not show a consistent pattern across projects. Therefore, five out of the seven mutators studied in this thesis are able to influence the distribution of mutants generated in different types of statements.

***Finding 15**: Our results indicate that, 5/7 of the mutation operators studied correlate with the type of statement mutated.*

Table 4.9: Statistics of mutants at different nesting level.

Levels	JFreeChart				Java-Library				Lambdaj				Asterisk			
	N_{stmt}	M_{total}	M_{killed}	MS	N_{stmt}	M_{total}	M_{killed}	MS	N_{stmt}	M_{total}	M_{killed}	MS	N_{stmt}	M_{total}	M_{killed}	MS
1	24810	8894	4281	48.1	3892	1384	1101	79.6	811	698	545	78.1	6465	739	486	65.8
2	10970	3279	1627	49.6	979	878	582	66.3	318	180	136	75.6	1168	217	169	77.9
3	3391	979	286	29.2	175	121	106	87.6	112	27	16	59.3	317	59	48	81.4
>=4	2121	569	75	13.2	43	335	273	81.5	118	28	20	71.4	142	34	31	91.2

Table 4.10: Statistics of explicit detectable mutants at different nesting level.

Levels	JFreeChart			Lambdaj			Java-Library			Asterisk		
	killed	*explicit*	%	*killed*	*explicit*	%	*killed*	*explicit*	%	*killed*	*explicit*	%
1	4281	2814	65.7	355	197	55.5	1101	641	58.2	486	382	78.6
2	1627	1337	82.2	169	71	42.0	582	168	28.9	169	140	82.8
3	286	209	73.1	389	247	63.5	106	42	39.6	48	40	83.3
>=4	75	40	53.3	102	49	48.0	273	89	32.6	31	22	71.0

Nesting levels Table 4.9 shows the statistics of statements and mutants at different nested levels. Column N_{stmt} shows total number of statements at different nested levels, column M_{total} indicates total number of mutants generated at each level, column M_{killed} shows the number of detectable mutants at that level, and column MS calculates the mutation score of the original test suite if only consider the mutants generated at the level. The mutation score decreases as nested level increases from 48.1% to 13.2% for JFreeChart, but increases from 65.8 % to 91.2 % in Asterisk. There is not a clear increasing/decreasing trend in mutation score as nested level goes up in Java-library and Lambdaj. Therefore, we did not observe any correlation between the nested level of a mutant and how easy/hard it can be detected. Table 4.10 summaries the distribution of explicitly detected mutants by assertions out of all detected mutants. There is no correlation between the level of nesting and the ratio of explicitly detected mutants. This might be due to the presence of dedicated test assertions that testers write for nested statements. There is thus no evidence that testers pay less attention to deeper nested statements.

***Finding 16**: Our results indicate that, there is no correlation between how easy a mutant can be killed and the depth of nesting of the mutated statement.*

Chapter 5

Discussion

5.1 Test Suite Size vs. Assertion Quantity

From the findings 1 and 2, the number of assertions in a test suite is very strongly correlated with its effectiveness with or without controlling for the influence of test size. However, according to finding 3, if we in turn control for the number of assertions, a test suite's effectiveness at a same test size level can be directly influenced by the number of assertions it contains. Thus, test suite size is not sufficient in predicting the effectiveness without considering the influence of assertion quantity. In addition, assertion quantity provides extra indications about the suite's explicit mutation score, which constitutes a large portion of the mutation score. Therefore, test suite size can predict the effectiveness only under the assumption that there is a linear relationship between the number of test methods and the number of assertions in the test suite. We believe this is an interesting finding, which explains why previous studies [28] have found a strong correlation between suite size and effectiveness.

5.2 Implicit vs. Explicit Mutation Score

We noticed an interesting phenomenon, namely, that mutants that are implicitly detectable can also be detected by assertions, if the mutated statement falls in the coverage of the assertion. However, mutants that are explicitly detectable by assertions can never be detected by non-assertion statements of the tests. This is because explicitly detectable mutants cannot be detected by simply executing the mutated part of a program; i.e., a specific assertion statement is required to catch the program's unexpected behaviour. This is due to the fact that explicitly detectable mutants inject logical faults into a program that lead to a contradiction with the programmers' expectations. From our observations, more than half of all detectable mutants (28%–73%) are explicitly detected by assertions in a test suite; and therefore assertions strongly influence test suite effectiveness. If we only focus on explicitly de-

tectable mutants, then test assertions are the only means to achieve suite effectiveness. This might also explain why statement coverage achieves a relatively low correlation with explicit mutation score.

5.3 Statement vs. Assertion Coverage

From findings 4 and 5, assertion coverage is a good estimator of both mutation score and explicit mutation score. If the influence of assertion coverage is controlled, there is a passable level of correlation between statement coverage and mutation score, while only a weak correlation between statement coverage and explicit mutation score. Therefore, statement coverage is a valid estimator of mutation score only under the assumption that not all of generated mutants are explicitly detectable mutants. In other words, statement coverage is not an adequate metric of logical-fault detection ability. Statement coverage includes more statements than assertion coverage from source code, without providing any extra insights for predicting test suite effectiveness. Compared with statement coverage, assertion coverage is very strongly correlated with the effectiveness regardless of the distribution of implicitly or explicitly detectable mutants. Our results suggest that testers should aim at increasing the assertion coverage of their test suite instead of its statement coverage, when trying to improve a test suite's effectiveness.

5.4 Assertion Type

Findings 8 and 7 indicate that there is no consistent ranking nor a significant difference between the effectiveness of different types of assertions. We manually assessed the assertions in the sample test suites and found that the assertions of one type can be easily interpreted as another type. For example, `assertEquals/Not` can be easily interpreted as `assertTrue/False`: `assertEquals(A, B)` can also be written as `assertTrue(A.equals(B))`, and `assertNotEquals(A, B)` can also be written as `assertFalse(A.equals(B))`. For `assertEquals/Not(A, B)`, the assertion content type is the type of A and B, whereas the assertion content type of `assertTrue/False(A.equals(B))` is boolean, with effectiveness of the assertion stays the same. `assertNull/Not` can be interpreted as `assertTrue/False` as well: `assertNull(A)` can be written as `assertTrue(A==null)`, where the assertion content type changes from the type of A to boolean. Therefore, the way we classify assertions is not able to distinguish them into disjoint sets. Assertions should be classified according to a more fine-grained methodology to measure their impact on

test effectiveness.

5.5 Distribution of Mutants

Finding 9 shows that the number of mutants generated by different mutation operators is not distributed evenly. Their distributions are strongly correlated with the distribution of mutable locations of the mutators in the source code. Finding 15 tells us that mutants generated by 5/7 mutation operators are strongly correlated with program statement types. In other word, the distribution of mutants is strongly correlated with the code characteristics of a program. Therefore, we believe, in the context of mutation testing, researchers and tester should always consider (and report) the characteristics their subjects, such as the distribution of different statement types.

5.6 Mutation Selection

Findings 10 and 11 indicate that there is always a significant difference between how easy mutants generated by different mutators can be killed. And, omitting *Return Values Mutator* always influences measuring test suite effectiveness. There are many influencing factors of whether omitting a mutation operator will cause a significant change in test suite effectiveness. One reason can be, mutants generated by the operator are easy/hard to detect mutants compare to all of the mutants generated for the subject. For example, as suggested by finding 14, mutants generated by *Return Values Mutator* are easy to detect since most of them are in return statements. In addition, the total number of mutants generated by a mutation operator is also important. Omitting a small number of mutants may not be significant enough to reveal the difference. Therefore, the two factors should be carefully considered before omitting any mutation operator when assessing test suite effectiveness through mutation testing. For example, if there is a big portion of return statements in the program, *Return Values Mutator* is likely to generate a large number of easy-to-detect mutants in return statements, and omitting the operator is likely to influence measuring test suite effectiveness.

5.7 Statement Type

From findings 13 and 14 we learn that the type of statement where a program is mutated significantly influences how easy a mutant is detected. Mutants in return statements are easiest to be killed. We believe this is because

return statements are the statements where values propagate and are passed between methods. Since tests target the return statements of methods, errors in return statements are easiest to be revealed. Mutants in conditional statements are harder to be detected compared to return statements. Faults in conditional statements can change the control flow of a program, therefore requiring a specific test to be revealed. From finding 12, PIT is more likely to mutate return statements and conditional statements than normal statements. Therefore, PIT may overestimate a test suite's effectiveness, since mutants in normal statements are hardest to be detected. This is important to consider when measuring test suite effectiveness.

5.8 Threats to Validity

Internal validity: To conduct the controlled experiments, we made use of many existing tools, such as PIT [10], Clover [3], and JavaSlicer [7]. We assumed these tools are able to produce valid results. Therefore, any erroneous behaviours of these tools might introduce unknown factors to the validity of our results. To mitigate such factors as much as possible, we tested our own code that uses these external tools.

Similar to previous studies [28], we treated mutants that cannot be detected by the master test suites as equivalent mutants, which might overestimate the number of equivalent mutants. However, since we are mainly concerned with the correlations between mutation/explicit mutation score and the other metrics, subtracting a constant value from the total number of mutants generated, will not impact the correlations.

External validity: We studied the relationship between assertions and test suite effectiveness using more than 24,000 assertions collected from five open source Java programs. However, programs written in Java may not be representative of the programs written in other languages. Thus, our results might not extend to other languages. Moreover, the assertions we examined in this thesis are JUnit4 assertions, and our results may not apply to assertions used in other testing frameworks. We mainly looked at the 9,177 assertions for JFreeChart [8] when comparing the effectiveness of different assertion types. Although the set of assertions used is large and written by real developers, our findings may not generalize to other programs. In addition, we used PIT to conduct mutation testing; PIT stops executing once a test assertion detects a mutant. However, it is helpful to know all the assertions that would fail when studying assertion types. We mainly looked

at mutants that are generated by the seven default mutation operators of PIT. Although the mutants generated have covered all types of program statements (return, conditional, and normal statements), they may not be representative of all existing types of program faults. We used Randoop to generate test cases with assertions, which were compared to human-written assertions. However, there also exist other test oracle generation strategies than feedback-directed random test generation, and using a different test generation strategy might influence the results. We used Randoop, because of its relative ease of use. .

Construct validity: The mutants we used in this thesis are generated by using PIT [10]. When assessing the distribution of different types of faults and faults in different types of statements, the implementation decision of the tool can influences what we observe. Utilizing a different mutation testing tool may influence the results. We used PIT because it is the most popular Java mutation testing tool, which has also been widely adopted in the literature, such as [26, 28, 37, 39, 44].

Our empirical data as well as the five subject programs are all available online, making our study repeatable.

Chapter 6

Conclusions and Future Work

In this thesis, we studied the influence of assertions and mutants on measuring test suite effectiveness. First, we examined the correlation between assertion quantity and the effectiveness, and further analyzed the influence of assertion quantity on the correlation between test suite size and the effectiveness. Second, we investigated the relationship between assertion coverage and suite effectiveness, and explored the impact of assertion coverage on the relation between statement coverage and effectiveness. Third, we compared the effectiveness of different assertion characteristics. Fourth, we investigated different types of faults (simulated by mutants generated by different mutation operators). We explored the distribution of mutants generated by different mutation operators, compared different types of mutants in terms of test suite effectiveness, and studied the influence of omitting each single mutation operator, separately. Finally, we investigated faults located in different types of program statements. We observed the distribution of mutants generated in different types of program statements, compared mutants generated in different types of statements in terms of test effectiveness, studied the relationship between types of statements and operators, and the influence of depth of nesting of program statements and the effectiveness. Based on an analysis of over 24,000 assertions collected from five cross-domain real-world Java programs, we found that:

- There is a very strong correlation between the *number of assertions* and test suite effectiveness, with or without controlling for the number of test methods in the test suite. Thus, the number of assertions in a test suite can significantly influence the prediction power of test suite size for the effectiveness.
- There is a very strong correlation between *assertion coverage* and test suite effectiveness. With assertion coverage controlled for, there is a moderate to strong correlation between statement coverage and mutation score, and only a weak to moderate correlation between statement coverage and explicit mutation score. Therefore, statement coverage is an adequate metric of test suite effectiveness only under

the assumption that the faults to be detected are not only explicitly detectable, while assertion coverage is a good estimator of test suite effectiveness without such a constraint.

- Types of assertions can influence the effectiveness of their containing test suites: human-written assertions are more effective than generated assertions.
- There is no evidence to support that assertion content types and assertion method types significantly influence test suite effectiveness.
- There is a significant difference between mutants generated by different mutation operators in terms of test effectiveness. The distribution of different types of mutants is strongly correlated with the distribution of mutable locations of the operators. Omitting a single mutation operator does not always significantly influence test suite effectiveness. From our observations, omitting Return Val Mutator always leads to a loss in test suite effectiveness, Omitting Increments Mutator and Invert Negatives Mutator can not introduce a significant difference in test suite effectiveness.
- There is a significant difference between mutants located in different types of statements, in terms of how easy they can be killed. We found that the distribution of mutants in different types of statements strongly correlates with test suite effectiveness. Mutants generated in return statements are easiest to be detected. Mutants generated in conditional statements are easier to be detected than mutants generated in normal statements. Five out of seven mutation operators studied in this thesis are correlated with mutation location.

Our results indicate that it might be sufficient to use the assertion quantity and assertion coverage as criteria to measure a suite's adequacy, since these two metrics are *at least* as good as suite size and statement coverage. In addition, fault type and fault location significantly influence measuring test suite effectiveness. To precisely assess a test suite's effectiveness, it is essential to describe the set of faults used by indicating the distribution of fault types and locations.

For future work, we would like to conduct experiments using more programs to further validate our findings. We plan to include more mutation operators in our studies in the future to improve generalization of the findings. We did not find a significant difference between different assertion types since different types of assertions can sometimes be difficult to distinguish one from another. To obtain a more fine-grained analysis, assertions should be classified more precisely as disjoint groups. For example, *assertEquals(A, B)*

and *assertTrue(A.equals(B))* should always be classified as one type instead of two. Moreover, we will conduct a taxonomy on assertions to more precisely differentiate their characteristics.

Bibliography

[1] Apache commons lang. http://commons.apache.org/proper/commons-lang/. Accessed: 2014-10-30.

[2] Asterisk-java. https://blogs.reucon.com/asterisk-java/. Accessed: 2014-10-30.

[3] Clover. https://www.atlassian.com/software/clover/overview/. Accessed: 2014-10-01.

[4] ErrorCollector. http://junit.org/junit4/javadoc/4.12/org/junit/rules/ErrorCollector.html. Accessed: 2015-09-30.

[5] Github. https://github.com/. Accessed: 2014-09-05.

[6] JavaParser. https://code.google.com/p/javaparser/. Accessed: 2014-09-05.

[7] JavaSlicer. https://www.st.cs.uni-saarland.de/javaslicer/. Accessed: 2014-10-15.

[8] JFreeChart. http://www.jfree.org/jfreechart/. Accessed: 2014-09-13.

[9] Lambdaj. https://code.google.com/p/lambdaj/. Accessed: 2014-10-30.

[10] Pit. http://pitest.org. Accessed: 2014-10-07.

[11] SLOCCount. http://www.dwheeler.com/sloccount/. Accessed: 2014-12-15.

[12] Urban airship java library. http://docs.urbanairship.com/reference/libraries/java/. Accessed: 2014-10-30.

[13] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30. IEEE Computer Society, 2014.

[14] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 402–411. ACM, 2005.

[15] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32:608–624, 2006.

[16] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c? *Software Testing, Verification and Reliability*, 11(2), 2001.

[17] Lionel Briand and Dietmar Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE, pages 148–157. IEEE Computer Society, 1999.

[18] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the International Workshop on Advances in Model-based Testing*, A-MOST, pages 1–7. ACM, 2005.

[19] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 158–171. ACM, 1996.

[20] Marcio Eduardo Delamaro, Lin Deng, Vinicius Humberto Serapilha Durelli, Nan Li, and Jeff Offutt. Experimental evaluation of sdl and one-op mutation for c. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 203–212. IEEE Computer Society, 2014.

[21] Marcio Eduardo Delamaro, Lin Deng, Nan Li, Vinicius Durelli, and Jeff Offutt. Growing a reduced set of mutation operators. In *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security*, ARES '14, pages 81–90. IEEE Computer Society, 2014.

[22] Lin Deng, Jeff Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation operator. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 84–93. IEEE Computer Society, 2013.

[23] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.*, 19:774–787, 1993.

[24] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 153–162. ACM, 1998.

[25] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 154–164. ACM, 1991.

[26] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 72–82. ACM, 2014.

[27] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 738–748. IEEE Press, 2012.

[28] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 435–445. ACM, 2014.

[29] 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 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 654–665. ACM, 2014.

[30] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing Verification and Reliability*, 9:205–232, 1999.

[31] Akbar Siami Namin and James H. Andrews. Finding sufficient mutation operators via variable reduction. In *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, pages 5–. IEEE Computer Society, 2006.

[32] Akbar Siami Namin and James H. Andrews. On sufficiency of mutants. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 73–74. IEEE Computer Society, 2007.

[33] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 57–68. ACM, 2009.

[34] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, 1996.

[35] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 100–107. IEEE Computer Society Press, 1993.

[36] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE Computer Society, 2007.

[37] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558. ACM, 2016.

[38] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST, pages 90–99. IEEE Computer Society, 2011.

[39] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 237–247. ACM, 2015.

[40] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings*

of the 30th International Conference on Software Engineering, ICSE '08, pages 351–360. ACM, 2008.

[41] Roland H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4. ACM, 2009.

[42] W. Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *J. Syst. Softw.*, 31:185–196, 1995.

[43] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444. ACM, 2010.

[44] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 214–224. ACM, 2015.