Assessing Test Suite Effectiveness Using Static Metrics

Paco van Beckhoven[1,2], Ana Oprescu[1], and Magiel Bruntink[2]

[1]University of Amsterdam
[2]Software Improvement Group

Abstract

With the increasing amount of automated tests, we need ways to measure the test effectiveness. The state-of-the-art technique for assessing test effectiveness, mutation testing, is too slow and cumbersome to be used in large scale evolution studies or code audits by external companies. In this paper we investigated two alternatives, namely code coverage and assertion count. We discovered that code coverage outperforms assertion count by showing a relation with test suite effectiveness for all analysed project. Assertion count only displays such a relation in only one of the analysed projects. Further analysing this relationship between assertion count coverage and test effectiveness would allow to circumvent some of the problems of mutation testing.

1 Introduction

Software testing is an important part of the software engineering process. It is widely used in the industry for quality assurance as tests can tackle software bugs early in the development process and also serve for regression purposes [20]. Part of the software testing process is covered by developers writing automated tests such as unit tests. This process is supported by testing frameworks such as JUnit [19]. Monitoring the quality of the test code has been shown to provide valuable insight when maintaining high-quality assurance standards [18]. Previous research shows that as the size of production code grows, the size of test code grows along [43]. Quality control on test suites is therefore important as the maintenance on tests can be difficult and generate risks if done incorrectly [22]. Typically, such risks are related to the growing size and complexity which consequently lead to incomprehensible tests. An important risk is the occurrence of *test bugs i.e.*, tests that fail although the program is correct (*false positive*) or even worse, tests that do not fail when the program is not working as desired (*false negative*). Especially the latter is a problem when breaking changes are not detected by the test suite. This issue can be addressed by measuring the fault detecting capability of a test suite, *i.e.*, test suite effectiveness Test suite effectiveness is measured by the number of faulty versions of a System Under Test (SUT) that are detected by a test suite. However, as real faults are unknown in advance, mutation testing is applied as a proxy measurement. It has been shown that mutant detection correlates with real fault detection [26].

Mutation testing tools generate faulty versions of the program and then run the tests to determine if the fault was detected. These faults, called mutants, are created by so-called mutators which mutate specific statements in the source code. Each mutant represents a very small change to prevent changing the overall functionality of the program. Some examples of mutators are: replacing operands or operators in an expression, removing statements or changing the returned values. A mutant is killed if it is detected by the test suite, either because the program fails to execute (due to exceptions) or because the results are not as expected. If a large set of mutants survives, it might be an indication that the test quality is insufficient as programming errors may remain undetected.

1.1 Problem statement

Mutation analysis is used to measure the test suite effectiveness of a project [26]. However, mutation testing techniques have several drawbacks, such as limited availability across programming languages and being resource expensive [46, 25]. Furthermore, it often requires compilation of source code and it requires running tests which often depend

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

on other systems that might not be available, rendering it impractical for external analysis. External analysis is often applied in industry by companies such as Software Improvement Group (SIG) to advise companies on the quality of their software. All these issues are compounded when performing software evolution analysis on large-scale legacy or open source projects. Therefore our research goal has both industry and research relevance.

1.2 Research questions and method

To tackle these issues, our goal is to understand to what extent metrics obtained through static source code analysis relate to test suite effectiveness as measured with mutation testing.

Preliminary research [40] on static test metrics highlighted two promising candidates: assertion count and static coverage. We structure our analysis on the following research questions:

RQ 1 To what extent is assertion count a good predictor for test suite effectiveness?

RQ 2 To what extent is static coverage a good predictor for test suite effectiveness?

We select our test suite effectiveness metric and mutation tool based on state of the art literature. Next, we study existing test quality models to inspect which static metrics can be related to test suite effectiveness. Based on these results we implement a set of metrics using only static analysis.

To answer the research questions, we implement a simple tool that reads a project's source files and calculates the metrics scores using static analysis.

Finally, we evaluate the individual metrics' suitability as indicators for effectiveness by performing a case study using our tool on three projects: Checkstyle, JFreeChart and JodaTime. The projects were selected from related research, based on size and structure of their respective test suites. We focus on Java projects as Java is one of the most popular programming languages [15] and forms the subject of many recent research papers surrounding test effectiveness. We rely on JUnit [7] as the unit testing framework. JUnit is the most used unit testing framework for Java [44].

1.3 Contributions

In an effort to tackle the drawbacks of using mutation testing to measure test suite effectiveness, our research makes the following contributions: **1.** In-depth analysis on the relation between test effectiveness, assertion count and coverage as measured using static metrics for three large real-world projects. **2.** A set of scenarios which influence the results of the static metrics and their sources of imprecision. **3.** An tool to measure static coverage and assertion count using only static metrics.

Outline. Section 2 revisits background concepts. Section 3 introduces the design of the static metrics that will be investigated together with an effectiveness metric and a mutation tool. Section 4 describes the empirical method of our research. Results are shown in Section 5 and discussed in Section 6. Section 7 summarises related work and Section 8 presents the conclusion and future work.

2 Background

First, we introduce some basic terminology. Next, we describe a test quality model used as input for the design of our static metrics. We briefly introduce mutation testing and compare mutation tools. Finally, we summarize test effectiveness measures and describe mutation analysis.

2.1 Terminology

We define several terms used in this paper:

Test (case/method) An individual JUnit test.
Test suite A set of tests.
Test suite size Number of tests in a test suite.
Master test suite All tests of a given project.
Dynamic metrics Metrics that can only be measured by, *e.g.*, running a test suite. When we state that something is measured dynamically, we refer to dynamic metrics.
Static metrics Metrics measured by analysing the source code of a project. When we state that something is measured statically, we refer to static metrics.

2.2 Measuring test code quality

Athanasiou *et al.* introduced a Test Quality Model (TQM) based on metrics obtained through static analysis of production and test code [18]. This TQM consists of the following static metrics:

Code coverage is percentage of code tested, implemented via static call graph analysis [16].
Assertion-McCabe ratio indicates tested decision points in the code; computed as the total number of assertion statements in the test code divided by the McCabe's cyclomatic complexity score [33] of the production code.
Assertion Density indicates the ability to detect defects; computed as the number of assertions divided by Lines Of Test Code (TLOC).
Directness indicates the ability to detect the location a defect's cause when a test fails. Similar to code coverage, except that only methods directly called from a test are counted.
Maintainability based on an existing maintainability model [21], adapted for test suites. The model consists of the following metrics for test code: Duplication, Unit Size, Unit Complexity and Unit Dependency.

2.3 Mutation testing

Test effectiveness is measured by the number of mutants that were killed by a test suite. Recent research introduced a variety of effectiveness measures and mutants. We describe different types of mutants, mutation tools, types of effectiveness measures, and work on mutation analysis.

2.3.1 Mutant types

Not all mutants are equally easy to detect. Easy or weak mutants are killed by many tests and thus often easy to detect. Hard to kill mutants can only be killed by very specific tests and often subsume other mutants. Below is an overview of the different types of mutants in the literature:

Mutant represents a small change to the program, *i.e.*, a modified version of the SUT.

Equivalent mutants do not change the outcome of a program, *i.e.*, they cannot be detected. Given a loop that breaks if $i == 10$, and i increments by 1. A mutant changing the condition to $i >= 10$ remains undetected as the loop still breaks when i becomes 10.

Subsuming mutants are sole contributors to the effectiveness scores [36]. If mutants are subsumed, they are often killed "collaterally" together with the subsuming mutant. Killing these collateral mutants does not lead to more effective tests, but they influence the test effectiveness score calculation.

2.3.2 Comparison of mutation tools

Three criteria were used to compare mutation tools for Java: **1.** Effectiveness of the *mutation adequate test suite* of each tool. A mutation adequate test suite kills all the mutants generated by a mutation tool. Each test of this test suite contributes to the effectiveness score, *i.e.*, if one test is removed, less than 100% effectiveness score is achieved. A *cross-testing* technique is applied to evaluate the effectiveness each tool's mutation adequate test suite. The adequate test suite of each tool is run on the set of mutants generated by the other tools. If the mutation adequate test suite for tool A would detect all the mutants of tool B, but the suite of tool B would not detect all the mutants of tool A, then tool A would subsume tool B. **2.** Tool's application cost in terms of the number of test cases that need to be generated and the number of equivalent mutants that would have to be inspected. **3.** Execution time of each tool.

Kintis *et al.* analysed and compared the effectiveness of PIT, muJava and Major [27]. Each tool was evaluated using the cross-testing technique on twelve methods of six Java projects. They found that the mutation adequate test suite of muJava was the most effective, followed by Major and PIT. The ordering in terms of application cost was different: PIT required the least test cases and generated the smallest set of equivalent mutants.

Marki and Lindstrom performed similar research on the same mutation tools [32]. They used three small Java programs popular in literature. They found that none of the mutation tools subsumed each other. muJava generated the strongest mutants followed by Major and PIT, however, muJava generated significantly more equivalent mutants and was slower than Major and PIT.

Laurent *et al.* introduced PIT+, an improved version of PIT with an extended set of mutators [31]. They combined the test suites generated by Kintis *et al.* [27] into a mutation adequate test suite that would detect the combined set of mutants generated by PIT, muJava and Major. A mutation adequate test suite was also generated for PIT+. The set of mutants generated by PIT+ was equally strong as the combined set of mutants.

2.3.3 Effectiveness measures

We found three types of effectiveness measures:

Normal effectiveness calculated as the number of killed mutants divided by the total number of non-equivalents.

Normalised effectiveness calculated as the number of killed mutants divided by the number of covered mutants, *i.e.*, mutants located in code executed by the test suite. Intuitively, test suites killing more mutants while covering less code are more thorough than test suites killing the same number of mutants in a larger piece of source code [24].

Subsuming effectiveness is the percentage of killed subsuming mutants. Intuitively, strong mutants, *i.e.*, subsuming mutants, are not equally distributed [36], which could lead to skewed effectiveness results.

2.3.4 Mutation analysis

In this section, we describe research conducted on mutation analysis that underpins our approach.

Mutants and real faults. Just *et al.* investigated whether generated faults are a correct representation of real faults [26]. Statistically significant evidence shows that mutant detection correlates with real fault detection. They could relate 73% of the real faults to common mutators. Of the remaining 27%, 10% can be detected by enhancing the set of commonly used mutators. They used Major for generating mutations. Equivalent mutants were ignored as mutation scores were only compared for subsets of a project's test suite.

Code coverage and effectiveness. Inozemtseva and Holmes analysed the correlation between

code coverage and test suite effectiveness [24] on twelve studies. They found three main shortcomings: **1.** Studies did not control the suite size. As code coverage relates to the test suite size (more coverage is achieved by adding more tests), it remains unclear whether the correlation with effectiveness was due to size or coverage of the test suite. **2.** Small or synthetic programs limit generalisation to industry. **3.** Comparing only test suites that fully satisfy a certain coverage criterion. They argue that these results can be generalised to more realistic test suites. Eight studies showed a correlation between some coverage type and effectiveness independently of size; the strength varied, in some studies appearing only for high coverage.

They also conducted an experiment on five large open source Java projects. All mutants undetected by the master test suite were marked equivalent. To control for size, fixed size test suites are generated by randomly selecting tests from the master test suite. Coverage was measured using CodeCover [3] on statement, decision and modified condition levels. Effectiveness was measured using normal and normalised effectiveness. They found a low to moderate correlation between coverage and normal effectiveness when controlling for size. The coverage type had little impact on the correlation strength and only a weak correlation was found for normalised effectiveness.

Assertions and effectiveness. Zhang and Mesbah studied the relationship between assertions and test suite effectiveness [45]. Their experiment used five large open source Java projects, similarly to Inozemtseva and Holmes [24]. They found a strong correlation between assertion count and test effectiveness, even when test suite size was controlled for. They also found that some assertion types are more effective than others, *e.g.*, boolean and object assertions are more effective than string and numeric assertions.

3 Metrics and mutants

Our goal is to investigate to what extent static analysis based metrics are related to test suite effectiveness. First, we need to select a set of static metrics. Secondly, we need a tool to measure these metrics. Thirdly, we need a way to measure test effectiveness.

3.1 Metric selection

We choose two static analysis-based metrics that could predict test suite effectiveness. We analyse the state of the art TQM by Athanasiou *et al.* [18] because it is already based on static source code analysis. Furthermore, the TQM was developed in collaboration with SIG, the host company of this thesis, which means that knowledge of the model is directly available. This TQM consists of the following static metrics: Code Coverage, Assertion-McCabe ratio, Assertion Density, Directness and Test Code Maintainability (see also Section 2.2).

Test code maintainability relates to code readability and understandability, indicating how easily we can make changes. We drop maintainability as a candidate metric as we consider it the least related to completeness or effectiveness of tests.

The model also contains two assertion- and two coverage based metrics. Based on preliminary results we found that the number of assertions had a stronger correlation with test effectiveness than the two assertion based TQM metrics for all analysed projects. Similarly, the static code coverage performed better than directness in the correlation test with test effectiveness. To get a more qualitative analysis, we focus on one assertion based metric and one coverage based metric, respectively assertion count and static coverage.

Furthermore, coverage was shown to be related to test effectiveness [24, 35]. Others found a relation between assertions and fault density [28] and between assertions and test suite effectiveness [45].

3.2 Tool implementation

In this section, we explain the foundation of the tool and the details of the implemented metrics.

3.2.1 Tool architecture

Figure 1 presents the analysis steps. The rectangles are artefacts that form the in/output for the two processing stages.

The first processing step is performed by the Software Analysis Toolkit (SAT) [29], it constructs a call graph using only static source code analysis. Our analysis tool uses the call graph to measure both assertion count and static method coverage.

The SAT analyses *source code* and computes several metrics, *e.g.*, Lines of Code (LOC), McCabe complexity [33] and code duplication, which are stored in a *source graph*. This graph contains information on the structure of the project, such as which packages contain which classes, which classes contain which methods and the call relations between these methods. Each node is annotated with information such as lines of code. This graph is designed such that it can be used for many programming languages. By implementing our metrics on top of the SAT, we can do measurements for different programming languages.

3.2.2 Code coverage

Alves and Visser designed an algorithm for measuring method coverage using static source code analysis [16]. The algorithm takes as input a *call graph* obtained by static source code analysis. The

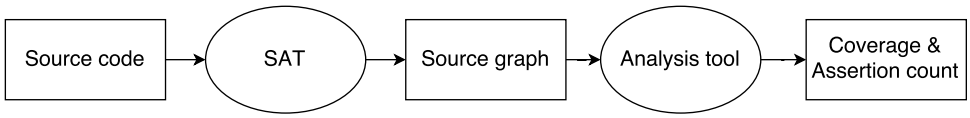


Figure 1: Analysis steps to statically measure coverage and assertion count.

calls from test to production code are counted by slicing the source graph and counting the methods. This includes indirect calls, *e.g.*, from one production method to another. Additionally, the constructor of each called method's class is included. They found a strong correlation between static and dynamic coverage. (The mean of the difference between static and dynamic coverage was 9%). We use this algorithm with the call graph generated by the SAT to calculate the static method coverage.

However, the static coverage algorithm has four sources of imprecision [16]. The first is conditional logic, *e.g.*, a switch statement that for each case invokes a different method. Second is dynamic dispatch (*virtual calls*), *e.g.*, a parent class with two subclasses both overriding a method that is called on the parent. Third, library/framework calls, *e.g.*, *java.util.List.contains*() invoke the *.equals*() method of each object in the list. The source code of third party libraries is not included in the analysis making it impossible to trace which methods are called from the framework. And fourth, the use of Java reflection, a technique to invoke methods dynamically during runtime without knowledge of these methods or classes during compile time.

For the first two sources of imprecision, an optimistic approach is chosen *i.e.*, all possible paths are considered covered. Consequently, the coverage is overestimated. Invocations by the latter two sources of imprecision remain undetected, leading to underestimating the coverage.

3.2.3 Assertions

We measure the number of assertions using the same call graph as the static method coverage algorithm. For each test, we follow the call graph through the test code to include all direct and indirect assertion calls. Indirect calls are important because often tests classes contain some utility method for asserting the correctness of an object. Additionally, we take into account the number of times a method is invoked to approximate the number of executed assertions. Only assertions that are part of JUnit are counted.

Identifying tests. By counting assertions based on the number of invocations from tests, we should also be able to identify these tests statically. We use the SAT to identify all invocations to assertion methods and then slice the call graph backwards following all *call* and *virtual call* edges. All nodes within scope, that have no parameters and have no incoming edges, are marked as tests.

Assertion content types. Zhang and Mesbah found a significant difference between the effectiveness of assertions and the type of objects they assert [45]. Four assertion content types were classified: numeric, string, object and boolean. They found that object and boolean assertions are more effective than string and numeric assertions. The type of objects in an assertion can give insights in the strength of the assertion. We will include the distribution of these content types in the analysis.

We use the SAT to analyse the type of objects in an assertion. The SAT is unable to detect the type of an operator expression used inside a method invocation, *e.g.*, `assertTrue(a >= b);`, resulting in unknown assertion content types. Also, fail statements are put in a separate category as these are a special type of assertion without any content type.

3.3 Mutation analysis

In this section we discuss our choice for the mutation tool and test effectiveness measure.

3.3.1 Mutation tool

We presented four candidate mutation tools for our experiment in Section 2.3.2: Major, muJava, PIT and PIT+. MuJava has not been updated in the last two years and does not support JUnit 4 and Java versions above 1.6 [9]. Conforming to these requirements would decrease the set of projects we could use in our experiment as both JUnit 4 and Java 1.7 have been around for quite some time. Major does support JUnit 4 and has recently been updated [8]. However, it only works in Unix environments [32]. PIT targets industry [27], is open source and actively developed [12]. Furthermore, it supports a wide scale of build tooling and is significantly faster than the other tools. PIT+ is based on a two-year-old branched version of PIT and was only recently made available [10]. The documentation is very sparse, the source code is missing. However, PIT+ generates a stronger set of mutants than the other three tools whereas PIT generates the weakest set of mutants.

Based on these observations we decided that PIT+ would be the best choice for measuring test effectiveness. Unfortunately, PIT+ was not available at the start of our research. We first did the analysis based on PIT and then later switched to PIT+. Because we first used PIT, we selected projects that used Maven as a build tool. PIT+ is based on an old version, 1.1.5, not yet supporting Maven. To enable using the features of PIT's new version we merged the mutators provided by PIT+ into the regular version of PIT [11].

3.3.2 Dealing with equivalent mutants

Equivalent mutants are mutants that do not change the outcome of the program. Manually removing equivalent mutants is time-consuming and generally undecidable [35]. A commonplace solution is to mark all the mutants that are not killed by the project's test suite as equivalent. The resulting non-equivalent mutants are always detected by at least one test. The disadvantage of this approach is that many mutants might be falsely marked as equivalent. The number of false positives depends for example on the coverage of the tests: if the mutated code is not covered by any of the tests, it will never be detected and consequently be marked as equivalent. Another cause of false positives could be the lack of assertions in tests, *i.e.*, not checking the correctness of the program's result. The percentage of equivalent mutants expresses to some extent the test effectiveness of the project's test suite.

With this approach, the complete test suite of each project will always kill all the remaining non-equivalent mutants. As the number of non-equivalent mutants heavily relies on the quality of a project's test suite, we cannot use these effectiveness scores to compare between different projects. To compensate for that, we will compare sub test suites within the same project.

3.3.3 Test effectiveness measure

Next, we evaluate both normalised and subsuming effectiveness in the subsections below and describe our choice for an effectiveness measure.

Normalised effectiveness. Normalised effectiveness is calculated by dividing the killed mutants with the number of non-equivalent mutants that are present in the code executed by the test.

Given the following example in which there are two Tests T_1 and T_2 for Method M_1. Suppose M_1 is **only** covered by T_1 and T_2. In total, there are five mutants $Mu_{1..5}$ generated for M_1. T_1 detects Mu_1 and T_2 detects Mu_2. As T_1 and T_2 are the only tests to kill M_1, the mutants $Mu_{3..5}$ remain undetected and are marked as equivalent. Both tests only cover M_1 and detect 1 of the two mutants resulting in a normal effectiveness score of 0.5. A test suite consisting of only the above tests would detect all mutants in the covered code, resulting in a normalised effectiveness score of 1.

We notice that the normalised effectiveness score heavily relies on how mutants are marked as equivalent. Suppose the mutants marked as equivalent were valid mutants but the tests failed to detect them (*false positive*), *e.g.*, due to missing assertions. In this scenario, the (normalised) effectiveness score suggests that a bad test suite is actually very effective. Projects that have ineffective tests will only detect a small portion of the mutants. As a result, a large percentage will be marked as equivalent. This increases the chances of false positives which decrease the reliability of the normalised effectiveness score.

Given a project of which only a portion of the code base is thoroughly tested. There is a high probability that the equivalent mutants are not equally distributed among the code base. Code covered by poor tests is more likely to contain false positives than thoroughly tested code. The poor tests scramble the results *e.g.*, a test with no assertions can be incorrectly marked as very effective.

Normalised effectiveness is intended to compare the thoroughness of two test suites, *i.e.*, penalise the test suites that cover lots of code but only a small number of mutants. We believe that it is less suitable as a replacement for normal effectiveness

We consider normal effectiveness scores more reliable when studying the relation with our metrics. Normal effectiveness is positively influenced by the breadth of a test and penalises small test suites as a score of 1.0 can only be achieved if all mutants are found. However, this is less of a problem when comparing test suites of equal sizes.

Subsuming effectiveness. Current algorithms for identifying subsuming mutants are influenced by the overlap between tests. Suppose there are five mutants, $Mu_{1..5}$, for method M_1. There are 5 tests, $T_{1..5}$, that kill $Mu_{1..4}$ and one test, T_6, that kills all five mutants.

Amman *et al.* defined subsuming mutants as follows: "one mutant subsumes a second mutant if every test that kills the first mutant is guaranteed also to kill the second [17]." According to this definition, Mu_5 subsumes $Mu_{1..4}$ because the set of tests that kill Mu_5 is a subset of the tests that kill $Mu_{1..4}$: $\{T_6\} \subset \{T_{1..5}\}$. The tests $T_{1..5}$ will have a subsuming effectiveness score of 0.

Our goal is to identify properties of test suites that determine their effectiveness. If we would measure the subsuming effectiveness, $T_{1..5}$ would be significantly less effective. This would suggest that the assertion count or coverage of these tests did not contribute to the effectiveness, even though they still detected 80% of all mutants.

Another vulnerability of this approach is that it is vulnerable to changes in the test set. If we remove T_6, the mutants previously marked as "subsumed" are now subsuming because Mu_5 is no longer detected. Consequently, $T_{1..5}$ now detect all the subsuming mutants. In this scenario, we decreased the quality of the master test suite by removing a single test, which leads to a significant increase in the subsuming effectiveness score of tests, $T_{1..5}$. This can lead to strange results over time, as the addition of tests can lead to drops in the effectiveness of others.

Choice of effectiveness measure. Normalised effectiveness loses precision when large amounts of mutants are incorrectly marked as equivalent. Furthermore, normalised effectiveness is intended as a measurement for the thoroughness of a test suite which is different from our definition of effectiveness. Subsuming effectiveness scores change when tests are added or removed which makes the measure very sensitive to change. Furthermore, subsuming effectiveness penalises tests that do not kill a subsuming mutant.

We choose to apply normal effectiveness as this measure is more reliable. It also allows for comparing with similar research on effectiveness and assertions/coverage [24, 45]. We refer to test suite effectiveness also as normal effectiveness.

4 Are static metrics related to test suite effectiveness?

Mutation tooling is resource expensive and requires running the test suites *i.e.*, dynamic analysis. To address these problems, we investigate to what extent static metrics are related to test suite effectiveness. In this section, we describe how we will measure whether static metrics are a good predictor for test suite effectiveness.

4.1 Measuring the relationship between static metrics and test effectiveness

We consider two static metrics, assertion count and static method coverage, as candidates for predicting test suite effectiveness.

4.1.1 Assertion count

We hypothesise that assertion count is related to test effectiveness. Therefore, we first measure assertion count by following the call graph from all tests. As our context is static source code analysis, we should be able to identify the tests statically. Thus, we next compare the following approaches:

Static approach we use static call graph slicing (Section 3.2.3) to identify all tests of a project and measure the total assertion count for the identified tests.

Semi-dynamic approach we use Java reflection (Section 4.3) to identify all the tests and measure the total assertion count for these tests.

Finally, we inspect the type of the asserted object as input for the analysis of the relationship between assertion count and test effectiveness.

4.1.2 Static method coverage

We hypothesise that static method coverage is related to test effectiveness. To test this hypothesis, we measure the static method coverage using static call graph slicing. We include dynamic method coverage as input for our analysis to: a) inspect the accuracy of the static methods coverage algorithm and b) to verify if a correlation between method coverage and test suite effectiveness exists.

4.2 Case study setup

We study our selected projects using an experiment design based on work by Inozemtseva and Holmes [24]. They surveyed similar studies on the relation between test effectiveness and coverage and found that most studies implemented the following procedure: **1.** Create faulty versions of one or more programs. **2.** Create or generate many test suites. **3.** Measure the metric scores of each suite. **4.** Determine the effectiveness of each suite. We describe our approach for each step in the following subsections.

4.2.1 Generating faults

We employ mutation testing as a technique for generating faulty versions, mutants, of the different projects that will be analysed. We employ PIT as a mutation tool. Mutants are generated using the default set of mutators [1]. All mutants that are not detected by the master test suite are removed.

4.2.2 Project selection

We have chosen three projects for our analysis based on the following set of requirements: The projects had in the order of hundreds of thousands LOC and thousands of tests.

Based on these criteria we selected a set of projects: Checkstyle[1], JFreeChart[5] and Joda-Time [6]. Table 1 shows properties of the projects. Java LOC and TLOC are generated using David A. Wheeler's SLOCCount [14].

Checkstyle is a static analysis tool that checks if Java code and Javadoc comply with some coding rules, implemented in checker classes. Java and Javadoc grammars are used to generate Abstract Syntax Trees (ASTs). The checker classes visit the AST, generating messages if violations occur. The core logic is in the `com.puppycrawl.tools.checkstyle.checks` package, representing 71% of the project's size. Checkstyle is the only project that used continuous integration and quality reports on GitHub to enforce quality, *e.g.*, the build that is triggered by a commit would break if coverage or effectiveness would drop below a certain threshold. We decided to use the build tooling's class exclusion filters to get more representative results. These quality measures are needed as there are several developers that contributed to the project. The project currently has five active team members [2].

[1] `http://pitest.org/quickstart/mutators/`

JFreeChart is a chart library for Java. The project is split into two parts: the logic used for data and data processing, and the code focussed on construction and drawing of plots. Most notable are the classes for the different plots in the `org.jfree.chart.plot` package, which contains 20% of the production code. JFreeChart is build and maintained by one developer [5].

JodaTime is a very popular date and time library. It provides functionality for calculations with dates and times in terms of periods, durations or intervals while supporting many different date formats, calendar systems and time zones. The structure of the project is relatively flat, with only five different packages that are all at the root level. Most of the logic is related to either formatting dates or date calculation. Around 25% of the code is related to date formatting and parsing. JodaTime was created by two developers, only of them is maintaining the project [6].

4.2.3 Composing test suites

It has been shown that test suite size influences the relation with test effectiveness [35]. When a test is added to a test suite it can never decrease the effectiveness, assertion count or coverage. Therefore, we will only compare tests suites of equal sizes similar to previous work [24, 45, 35].

We compose test suites of relative sizes, *i.e.*, test suites that contain a certain percentage of all tests in the master test suite. For each size, we generate 1000 test suites. We selected the following range of relative suite sizes: 1%, 4%, 9%, 16%, 25%, 36%, 49%, 64% and 81%. Larger test suite were not included because the differences between the generated test suites would become too small. Additionally, we found that this sequence had the least overlap in effectiveness scores for the different suite sizes while still including a wide spread of the test effectiveness across different test suites.

Our approach differs from existing research [24] in which they used suites of sizes: 3, 10, 30, 100, 300, 1000 and 3000. A disadvantage of this approach is that the number of test suites for JodaTime is larger than for the others because JodaTime is the only project that has more than 3000 tests. Another disadvantage is that a test suite with 300 tests might be 50% of the master test suite for one project and only 10% of another project's test suite. Additionally, most composed tests suites in this approach represent only a small portion of the master test suite. With our approach, we can more precisely study the behaviour of the metrics as the suites grow in size. Furthermore, we found that test suites with 16% of all tests already dynamically covered 50% to 70% of the methods covered by the master test suite.

4.2.4 Measuring metric scores and effectiveness

For each test suite, we measure the effectiveness, assertion count and static method coverage. The dynamic equivalents of both coverage metrics are included to evaluate their comparison. We obtain the dynamic coverage metrics using JaCoCo [4].

4.2.5 Statistical analysis

To determine how we will calculate the correlation with effectiveness we analyse related work on the relation between test effectiveness and assertion count [45] and coverage [24]. Both works have similar experiment set-ups in which they generated sub test suites of fixed sizes and calculated metric and effectiveness scores for these suites. Furthermore, both studies used a *parametric* and *non-parametric* correlation test, respectively *Pearson* and *Kendall*. We will also consider the Spearman rank correlation test, another nonparametric test, as it is commonly used in literature. A parametric test assumes the underlying data to be normally distributed whereas nonparametric tests do not.

The Pearson correlation coefficient is based on the covariance of two variables, *i.e.*, the metric and effectiveness scores, divided by the product of their standard deviations. Assumptions for Pearson include the absence of outliers, the normality of variables and linearity. The Kendall's Tau rank correlation coefficient is a rank based test used to measure the extent to which rankings of two variables are similar. Spearman is a rank based version of the Pearson correlation tests, commonly used as its computation is more lightweight than Kendall's. However, our data set leads to similar computation time for Spearman and Kendall.

We discard Pearson because we cannot make assumptions on our data distribution. Moreover, Kendall "is a better estimate of the corresponding population parameter and its standard error is known [23]". As the advantages of Spearman over Kendall do not apply in our case and Kendall has advantages over Spearman, we choose Kendall's Tau rank correlation test. The correlation coefficient is calculated with R's "Kendall" package [13]. We use the Guilford scale (Table 2) for verbal descriptions of the correlation strength [35].

4.3 Evaluation tool

We compose 1000 test suites of nine different sizes for each project. Running PIT+ on the master test suite took from 0.5 to 2 hours depending on the project. As we have to calculate the effectiveness of 27,000 test suites, this approach would take too much time. Our solution is to measure the test effectiveness of each test only once. We then combine the results for different sets of tests

Table 1: Characteristics of the selected projects. Total Java LOC is the sum of the production LOC and TLOC

Property	Checkstyle	JFreeChart	JodaTime
Total Java LOC	73,244	134,982	84,035
Production LOC	32,041	95,107	28,724
TLOC	41,203	39,875	55,311
Number of tests	1875	2,138	4,197
Method Coverage	98%	62%	90%
Date cloned from GitHub	4/30/17	4/25/17	3/23/17
Citations in literature	[43, 39]	[45, 24, 31, 26, 16]	[24, 31, 26, 39]
Number of generated mutants	95,185	310,735	100,893
Number of killed mutants	80,380	80,505	69,615
Number of equivalent mutants	14,805	230,230	31,278
Equivalent mutants (%)	15.6%	74.1%	31.0%

Table 2: Guilford scale for the verbal description of correlation coefficients.

Correlation coefficient	below 0.4	0.4 to 0.7	0.7 to 0.9	above 0.9
Verbal description	low	moderate	high	very high

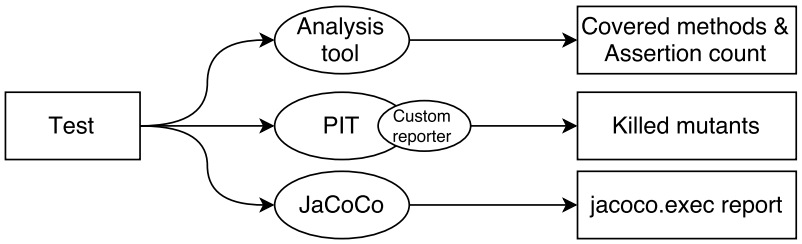


Figure 2: Overview of the experiment set-up to obtain the relevant metrics for each test.

to simulate test suites. To get the scores for a test suite with n tests, we combine the coverage results, assertion counts and killed mutants of its tests. Similarly, we calculate the static metrics and dynamic coverage only once for each test.

Detecting individual tests. We use a reflection library to detect both JUnit 3 and 4 tests for each project according to the following definitions:

JUnit 3 All methods in non-abstract subclasses of JUnit's `TestCase` class. Each method should have a name starting with "test", be public, void and have no parameters.

JUnit 4 All public methods annotated with JUnit's `@Test` annotation.

We verified the number of detected tests with the number of executed tests reported by each project's build tool.

We also need to include the set-up and tear-down logic of each test. We use JUnit's test runner API to execute individual tests. This API ensures execution of the corresponding set-up and tear-down logic. This extra test logic should also be included in the static coverage metric to get similar results. With JUnit 3 the extra logic is defined by overriding `TestCase.setUp()` or `TestCase.tearDown()`. JUnit 4 uses the `@before` or `@after` annotations. However, the SAT does not provide information on the used annotations. A common practice is to still name these methods `setUp` or `tearDown`. We include methods that are named setUp or tearDown and are located in the same class as the tests in the coverage results.

Aggregating metrics. To aggregate effectiveness, we need to know which mutants are detected by each test as the set of detected mutants could overlap. However, PIT does not provide a list of killed mutants. We solved this issue by creating a custom reporter using PIT's plug-in system to export the list of killed mutants.

The coverage of two tests can also overlap. Thus, we need information on the methods covered by each test. JaCoCo exports this information in a jacoco.exec report file, a binary file containing all the information required for aggregation. We aggregate these files via JaCoCo's API. For the static coverage metric, we export the list of covered methods in our analysis tool.

The assertion count of a test suite is simply calculated as the sum of each test's assertion count.

Figure 2 provides an overview of the involved tools used and the data they generate. The evaluation tool's input is raw test data and the sizes

of the test suites to create. We then compose test suites by randomly selecting a given number of tests from the master test suite. The output of the analysis tool is a data set containing the scores on the dynamic and static metrics for each test suite.

5 Results

We first present the results of our analysis on the assertion count metric, followed by the results of our analysis on code coverage.

Table 3 provides an overview of the assertion count, static and dynamic method coverage, and the percentage of mutants that were marked as equivalent for the master test suite of each project.

5.1 Assertion count

Figure 3 shows the distribution of the number of assertions for each test of each project.

We notice some tests with exceptionally high assertion counts. We manually checked these tests and found that the assertion count was correct for the outliers. We briefly explain a few outliers:

- `TestLocalDateTime_Properties.testPropertyRoundHour` (140 asserts), checks the correctness of rounding 20 times, with for each check 7 assertions on year, month, week, etc.
- `TestPeriodFormat.test_wordBased_pl_regEx` (140 asserts) calls and asserts the results of the polish regex parser 140 times.
- `TestGJChronology.testDurationFields` (57 asserts), tests for each duration field whether the field names are correct and if some flags are set correctly.
- `CategoryPlotTest.testEquals` (114 asserts), incrementally tests all variations of the `equals` method of a plot object. The other tests with more than 37 assertions are similar tests for the `equals` methods of other types of plots.

Figure 4 shows the relation between the assertion count and normal effectiveness. Each dot represents a generated test suite; and its colour of the dot represents the size of the suite relative to the total number of tests. The normal effectiveness, *i.e.*, the percentage of mutants killed by a given test suite is shown on the y-axis. The normalised assertion count is shown on the x-axis. We normalised the assertion count as the percentage of the total number of assertions for a given project. For example, as Checkstyle has 3819 assertions (see Table 3), a test suite with 100 assertions would have a normalised assertion count of $\frac{100}{3819} * 100 \approx 2.6\%$.

We observe that test suites of the same relative suite are clustered. For each group of test suites, we calculated the Kendall correlation coefficient between normal effectiveness and assertion count. These coefficients for each set of test suites of a given project and relative size are shown in Table 4. We highlight statistically significant correlations that have a p-value < 0.005 with two asterisks (**), and results with a p-value < 0.01 with a single asterisk (*).

We observe a statistically significant, low to moderate correlation for nearly all groups of test suites for JFreeChart. For JodaTime and Checkstyle, we notice significant but weaker correlations: 0.08-0.2 compared to JFreeChart's 0.14-0.4.

Table 5 shows the results of the two test identification approaches for the assertion count metric (see Section 4.1.1). False positives are tests that were incorrectly marked as tests. False negatives are tests that were not detected.

Figure 5 shows the distribution of asserted object types. Assertions for which we could not detect the content type are categorised as *unknown*.

5.2 Code coverage

Figure 6 shows the relation between static method coverage and normal effectiveness. A dot represents a test suite and its colour, the relative test suite size. Table 6 shows the Kendall correlation coefficients between static coverage and normal effectiveness for each set of test suites. We highlight statistically significant correlations that have a p-value < 0.005 with two asterisks (**), and results with a p-value < 0.01 with a single asterisk (*).

5.2.1 Static vs. dynamic method coverage

To evaluate the quality of the static method coverage algorithm, we compare static coverage with its dynamic counterpart for each suite (Figure 7). A dot represents a test suite, colours represent the size of a suite relative to the total number of tests. The black diagonal line illustrates the ideal line: all test suites below this line overestimate the coverage and all the test suites above underestimate the coverage. Table 7 shows the Kendall correlations between static and dynamic method coverage for the different projects and suite sizes. Each correlation coefficient maps to a set of test suites of the corresponding suite size and project. Coefficients with one asterisk (*) have a p-value < 0.01 and coefficients with two asterisks (**) have a p-value < 0.005. We observe a statistically significant, low to moderate correlation for all sets of test suites for JFreeChart and JodaTime.

5.2.2 Dynamic coverage and test suite effectiveness

Figure 8 shows the relation between dynamic method coverage and normal effectiveness. Each dot represents a test suite; its colour represents the size of that suite relative to the total number

Table 3: Results for the master test suite of each project.

Project	Assertions	Static coverage	Dynamic coverage	Equivalent mutants
Checkstyle	3,819	85%	98%	15.6%
JFreeChart	9,030	60%	62%	74.1%
JodaTime	23,830	85%	90%	31.0%

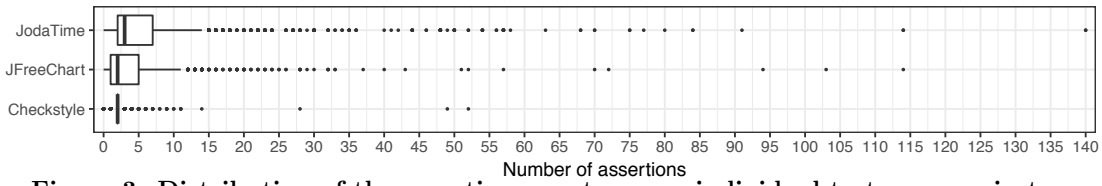


Figure 3: Distribution of the assertion count among individual tests per project.

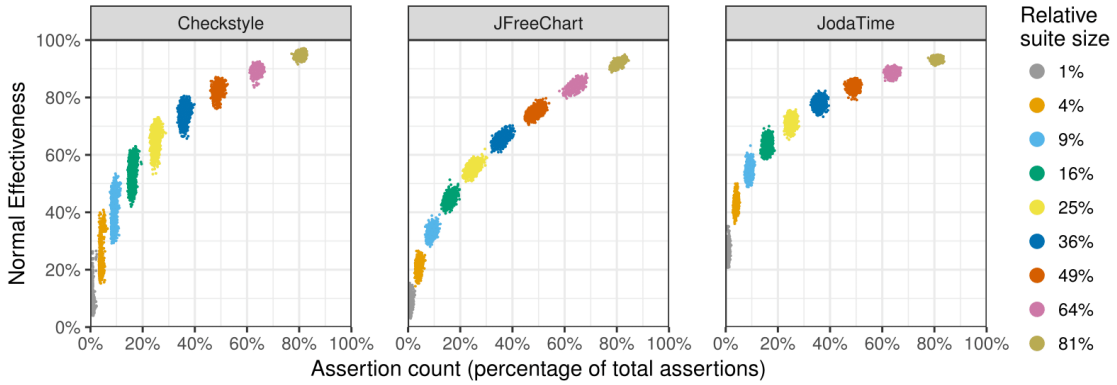


Figure 4: Relation between assertion count and test suite effectiveness.

Table 4: Kendall correlations between assertion count and test suite effectiveness.

Project	Relative test suite size								
	1%	4%	9%	16%	25%	36%	49%	64%	81%
Checkstyle	-0.04	0.08**	0.13**	0.18**	0.20**	0.16**	0.16**	0.12**	0.10**
JFreeChart	0.03	0.14**	0.23**	0.32**	0.34**	0.35**	0.39**	0.40**	0.36**
JodaTime	0.05	0.11**	0.13**	0.13**	0.07**	0.09**	0.07**	0.10**	0.06*

Table 5: Comparison of different approaches to identify tests for the assertion count metric.

Project	Semi-static approach		Static approach			
	Number of tests	Assertion count	Number of tests (diff)	Assertion count (diff)	False positives	False negatives
CheckStyle	1,875	3,819	1,821 (-54)	3,826 (+0.18%)	5	59
JFreeChart	2,138	9,030	2,172 (+34)	9,224 (+2.15%)	39	7
JodaTime	4,197	23,830	4,180 (-17)	23,943 (+0.47%)	15	32

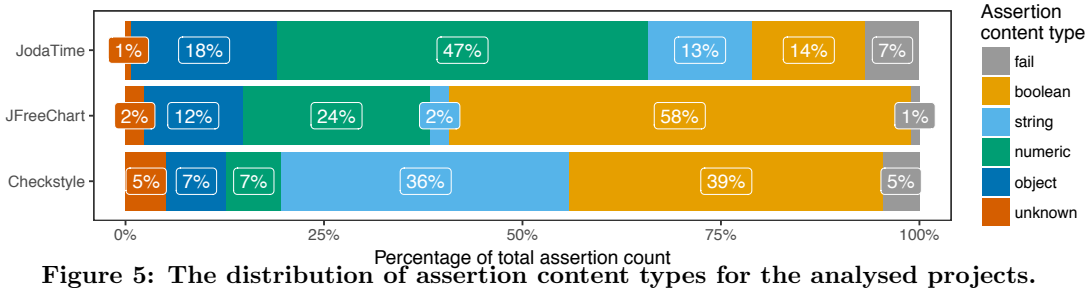


Figure 5: The distribution of assertion content types for the analysed projects.

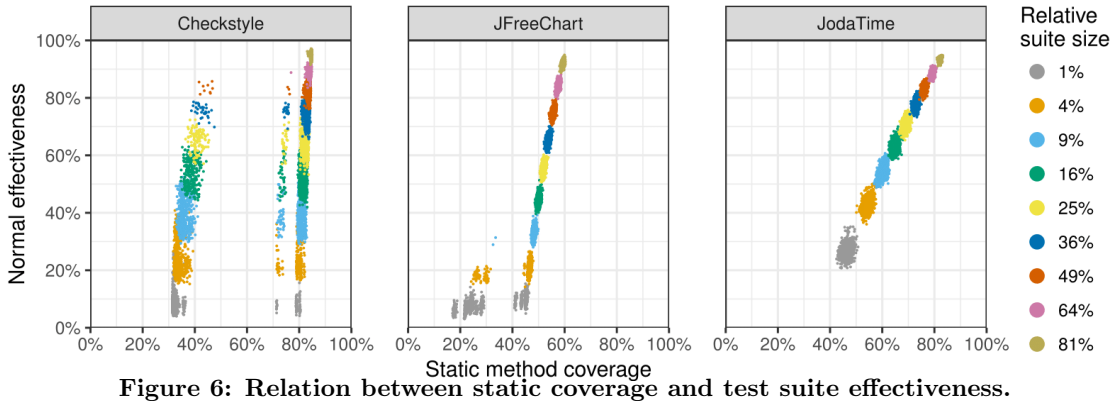


Figure 6: Relation between static coverage and test suite effectiveness.

Table 6: Kendall correlations between static method coverage and test suite effectiveness.

Project	Relative test suite size								
	1%	4%	9%	16%	25%	36%	49%	64%	81%
Checkstyle	-0.05	-0.01	-0.02	-0.02	0.00	-0.04	-0.01	0.00	0.01
JFreeChart	0.49**	0.28**	0.23**	0.26**	0.27**	0.28**	0.31**	0.31**	0.26**
JodaTime	0.13**	0.28**	0.32**	0.28**	0.24**	0.25**	0.23**	0.20**	0.21**

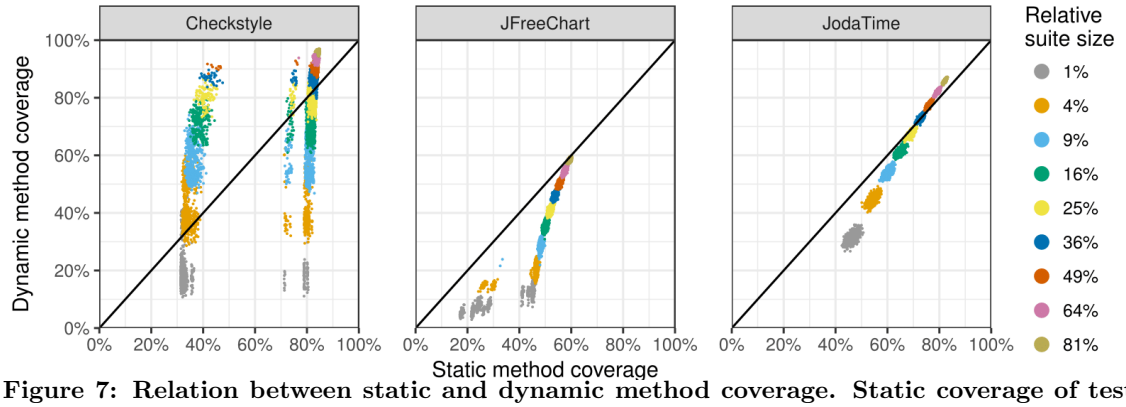


Figure 7: Relation between static and dynamic method coverage. Static coverage of test suites below the black line is overestimated, above is underestimated.

Table 7: Kendall correlation between static and dynamic method coverage.

Project	Relative test suite size								
	1%	4%	9%	16%	25%	36%	49%	64%	81%
Checkstyle	-0,03	-0,01	0,01	-0,02	0,00	0,00	0,05	0,10**	0,15**
JFreeChart	0,67**	0,33**	0,28**	0,31**	0,33**	0,35**	0,43**	0,45**	0,44**
JodaTime	0,35**	0,44**	0,48**	0,47**	0,51**	0,51**	0,52**	0,54**	0,59**

of tests. Table 8 shows the Kendall correlations between dynamic method coverage and normal effectiveness for the different groups of test suites for each project. Similarly to the other tables, two asterisks indicate that the correlation is statistically significant with a p-value < 0.005.

6 Discussion

We structure our discussion as follows: First, for each metric, we compare the results across all projects, perform an in-depth analysis on some of the projects and then answer to the corresponding research question. Next, we describe the practicality of this research and the threats to validity.

6.1 Assertions and test suite effectiveness

We observe that test suites of the same relative size form groups in the plots in Figure 4, *i.e.*, the assertion count and effectiveness score of same size test suites are relatively close to each other.

For JFreeChart, groups of test suites with a relative size >=9% exhibit a diagonal shape. This shape is ideal as it suggests that test suites with more assertions are more effective. These groups also show the strongest correlation between assertion count and effectiveness (Table 4).

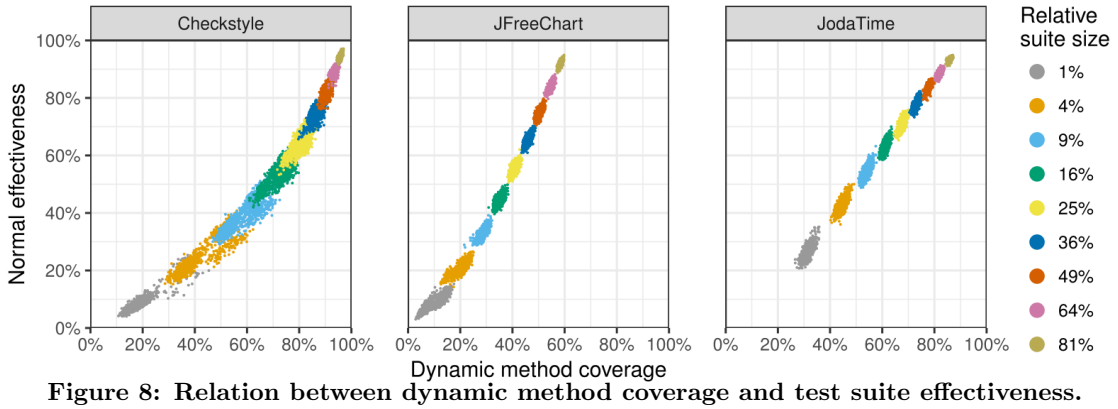


Figure 8: Relation between dynamic method coverage and test suite effectiveness.

Table 8: Kendall correlation between dynamic method coverage and test suite effectiveness.

Project	Relative test suite size								
	1%	4%	9%	16%	25%	36%	49%	64%	81%
Checkstyle	0.67**	0.71**	0.68**	0.59**	0.45**	0.36**	0.33**	0.31**	0.36**
JFreeChart	0.65**	0.59**	0.52**	0.48**	0.44**	0.47**	0.47**	0.49**	0.45**
JodaTime	0.48**	0.49**	0.53**	0.51**	0.48**	0.52**	0.48**	0.47**	0.44**

We notice that the normalised assertion count of a test suite is close to the relative suite size, *e.g.*, suites with a relative size of 81% have a normalised assertion count between 77% and 85%. The difference between the relative suite size and normalised assertion count is directly related to the variety in assertion count per test. More variety means that a test suite could exist with only below average assertion counts, resulting in a ¡80% normalised assertion count.

We analyse each project to find to what extent assertion count could predict test effectiveness.

6.1.1 Checkstyle

We notice a very low, statistically significant correlation between assertion count and test suite effectiveness for most of Checkstyle's test suite groups.

Most of the Checkstyle's tests target the different checks in Checkstyle. Out of the 1875 tests, 1503 (80%) tests belong to a class that extends the `BaseCheckTestSupport` class. The `BaseCheckTestSupport` class contains a set of utility methods for creating a checker, executing the checker and verifying the messages generated by the checker. We notice a large variety in test suite effectiveness among the tests that extend this class. Similarly, we expect the same variety in assertion counts. However, the assertion count is the same for at least 75% of these tests.

We found that 1156 of these tests (62% of the master test suite) use the *BaseCheckTestSupport.verify* method for asserting the checker's results. The `verify` method iterates over the expected violation messages which are passed as a parameter. This iteration hides the actual number of executed assertions. Consequently, we detect only two assertions for tests which might execute many assertions at runtime. In addition to the verify method, we found 60 tests that directly applied assertions inside for loops.

> **Finding 1:** Assertions within in an iteration block skew the estimated assertion count. These iterations are a source of imprecision because the actual number of assertions could be much higher than the assertion count we measured.

Another consequence of the high usage of `verify` is that these 1156 tests all have the same assertion count. Figure 3 shows similar results for the distribution of assertions for Checkstyle's tests.

The effectiveness scores for these 1156 tests range from 0% to 11% (the highest effectiveness score of an individual test). This range shows that the group of tests with two assertions include both the most and least effective tests. There are approximately 1200 tests for which we detect exactly two assertions. As this concerns 64% of all tests, we state there is too little variety in the assertion count to make predictions on the effectiveness.

> **Finding 2:** 64% of Checkstyle's tests have identical assertion counts. Variety in the assertion count is needed to distinguish between the effectiveness of different tests.

6.1.2 JFreeChart

JFreeChart is the only project exhibiting a low to moderate correlation for most groups of test suites.

We found many *strong* assertions in JFreeChart's tests. By strong, we mean that two large objects, *e.g.*, plots, are compared in an assertion. This assertion uses the object's `equals` implementation. In this `equals` method, around 50 lines long, many fields of the plot, such as `Paint` or `RectangleInsets` are compared, again relying on their consecutive `equals` implementation. We also notice that most outliers for JFreeChart in Figure 3 are tests for the equals methods which suggests that the `equals` methods contain much logic.

> **Finding 3:** Not all assertions are equally strong. Some only cover a single property, *e.g.*, a string or a number, whereas others compare two objects, potentially covering many properties. For JFreeChart, we notice a large number of assertions that compare plot objects with many properties.

Next, we searched for the combination of loops and assertions that could skew the results, and found no such occurrences in the tests.

6.1.3 JodaTime

The correlations between assertion count and test suite effectiveness for JodaTime are similar to that of Checkstyle, and much lower than those of JFreeChart. We further analyse JodaTime to find a possible explanation for the weak correlation.

Assertions in for loops. We searched for test utility methods similar to the `verify` method of Checkstyle, *i.e.*, a method that has assertions inside an iteration and is used by several tests. We observe that the four most effective tests, shown in Table 9, all call `testForwardTransitions` and/or `testReverseTransitions`, both are utility methods of the `TestBuilder` class. The rank columns contain the rank relative to the other tests of to provide some context in how they compare. Ranks are calculated based on the descending order of effectiveness or assertion count. If multiple tests have the same score, we show the average rank. Note that the utility methods are different from the tests in the top 4 that share the same name. The top 4 tests are the only tests calling these utility methods. Both methods iterate over a two-dimensional array containing a set of approximately 110 date time transitions. For each transition, 4 to 7 assertions are executed, resulting in more than 440 executed assertions.

Additionally, we found 22 tests that combined iterations and assertions. Out of these 22 tests, at least 12 tests contained fix length iterations, *e.g.*, `for(int i = 0; i < 10; i++)`, that could be evaluated using other forms of static analysis.

In total, we found only 26 tests of the master test suite (0.6%) that were directly affected by assertions in for loops. Thus, for JodaTime, assertions in for loops do not explain the weak correlation between assertion count and effectiveness.

Assertion strength. JodaTime has significantly more assertions than JFreeChart and Checkstyle. We observe many assertions on numeric values as one might expect from a library that is mostly about calculations on dates and times. For example, we noticed many utility methods that checked the properties of `Date`, `DateTime` or `Duration` objects. Each of these utility methods asserts the number of years, months, weeks, days, hours, *etc.* This large number of numeric assertion corresponds with the observation that 47% of the assertions are on numeric types (Figure 5).

However, the above is not always the case. For example, we found many tests, related to parsing dates or times from a string or tests for formatters, that only had a 1 or 2 assertions while still being in the top half of most effective tests.

We distinguish between two types of tests: a) tests related to the arithmetic aspect with many assertions and b) tests related to formatting with only a few assertions. We find that assertion count does not work well as a predictor for test suite effectiveness since the assertion count of a test does not directly relate to how effective the test is.

> **Finding 4:** Almost half of JodaTime's assertions are on numeric types. These assertions often occur in groups of 3 or more to assert a single result. However, a large number of effective tests only contains a small number of mostly non-numeric assertions. This mix leads to poor predictions.

6.1.4 Test identification

We measure the assertion count by following the static call graph for each test. As our context is static source code analysis, we also need to be able to identify the individual tests in the test code. We compare our static approach with a semi-static approach that uses Java reflection to identify tests.

Table 5 shows that the assertion count obtained with the static-approach is closer to the dynamic approach than the assertion count obtained through the semi-static approach.

For all projects the assertion count of the static approach is higher. If the static algorithm does not identify tests, there are no call edges between the tests and the assertions. The absence of edges implies that these tests either have no assertions or an edge in the call graph was missing. These tests do not contribute to the assertion count.

Table 9: JodaTime's four most effective tests

Test	Normal Effectiveness		Assertions	
	Score	Rank	Score	Rank
`TestCompiler.testCompile()`	17.23%	1	13	361.5
`TestBuilder.testSerialization()`	14.61%	2	13	361.5
`TestBuilder.testForwardTransitions()`	12.94%	3	7	1,063.5
`TestBuilder.testReverseTransitions()`	12.93%	4	4	1,773.0

We notice that the methods that were incorrectly marked as tests, false positives, are methods used for debugging purposes or methods that were missing the `@Test` annotation. The latter is most noticeable for JFreeChart. We identified 39 tests that were missing the `@Test` annotation. Of these 39 tests, 38 tests correctly executed when the `@Test` annotation was added. According to the repository's owner, these tests are valid tests [2].

Based on the results of these three projects, we also show that the use of call graph slicing gives accurate results on a project level.

6.1.5 Assertion count as a predictor for test effectiveness

We found that the correlation for Checkstyle and JodaTime is weaker than for JFreeChart. Our analysis indicates that the correlation for Checkstyle is less strong because of a combination of assertions in for loops (Finding 1) and the assertion distribution (Finding 2). However, this does not explain the weak correlation for JodaTime. As shown in Figure 3, JodaTime has a much larger spread in the assertion count of each test. Furthermore, we observe that the assertion-iteration combination does not have a significant impact on the relationship with test suite effectiveness compared to Checkstyle. We notice a set of strong assertions for JFreeChart (Finding 3) whereas JodaTime has mostly weak assertions (Finding 4).

RQ 1: To what extent is assertion count a good predictor for test suite effectiveness?

Assertion count has potential as a predictor for test suite effectiveness because assertions are directly related to detection of mutants. However, more work on assertions is needed as the correlation with test suite effectiveness is often weak or statistically insignificant.

For all three projects, Table 3, we observe different assertion counts. Checkstyle and JodaTime are of similar size and quality, but Checkstyle only has 16% of the assertions JodaTime has. JFreeChart has more assertions than Checkstyle, but the production code base that should be tested is also three-times bigger. A test quality model that includes the assertion count should incorporate information about the strength of the assertions, either by incorporating assertion content types, assertion coverage [45] or size of the asserted object. Furthermore, such a model should also include information about the size of a project.

If assertion count would be used, we should measure the presence of its sources of imprecision to judge the reliability. This measurement should also include the intensity of the usage of errornous methods. For example, we found hundreds of methods and tests with assertions in for-loops. However, only few methods that were often used had a significant impact on the results.

6.2 Coverage and effectiveness

We observe a diagonal-like shape for most groups of same size test suites in Figure 6. This shape is ideal as it suggests that within this group, test suites with more static coverage are more effective. These groups also show the strongest correlation between static coverage and test suite effectiveness, as shown in Table 6.

Furthermore, we notice a difference in the spread of the static coverage on the horizontal axis. For example, coverage for Checkstyle's tests suites can be split into three groups: around 30%, 70% and 80% coverage. JFreeChart shows a relatively large spread of coverage for smaller tests suites, ranging between 18% and 45% coverage, but the coverage converges as test suites grow in size. JodaTime is the only project for which there is no split in the coverage scores of same size test suites. We consider these differences in the spread of coverage a consequence of the quality of the static coverage algorithm. These differences are further explored in Section 6.2.1. We perform an in-depth analysis on Checkstyle in Section 6.2.2 because it is the only project which does not exhibit either a statistically significant correlation between static coverage and test effectiveness, or one between static coverage and dynamic method coverage.

6.2.1 Static vs. dynamic method coverage

When comparing dynamic and static coverage in Figure 7, we notice that the degree of over- or underestimation of the coverage depends on the project and test suite size. Smaller test suites tend to overestimate, whereas larger test suites underestimate. We observe that the quality of the static

[2] `https://github.com/jfree/jfreechart/issues/57`

coverage for the Checkstyle project is significantly different compared to the other projects. Checkstyle is discussed in Section 6.2.2.

Overestimating coverage. The static coverage for the smaller test suites is significantly higher than the real coverage, as measured with dynamic analysis. Suppose a method M_1 has a switch statement that, based on its input, calls one of the following methods, M_2, M_3, M_4. There are three tests, T_1, T_2, T_3, that each call M_1, with one of the three options for the switch statement in M_1 as a parameter. Additionally, there is a Test suite TS_1 that consists of T_1, T_2, T_3. Each test covers M_1 and one of M_2, M_3, M_4, all tests combined in TS_1 cover all 4 methods. The static coverage algorithm does not evaluate the switch statement and detects for each test that 4 methods are covered. This shows that static coverage is not very accurate for individual tests. However, the static coverage for TS_1 matches the dynamic coverage. This example illustrates why the loss in accuracy, caused by overestimating the coverage, decreases as test suites grow in size. The paths detected by the static and dynamic method coverage will eventually overlap once a test suite is created that contains all tests for a given function. The amount of overestimated coverage depends on how well the tests cover the different code paths.

> **Finding 5:** The degree of overestimation by the static method coverage algorithm depends on the real coverage and the amount of conditional logic and inheritance in the function under test.

Underestimating coverage. We observe that for larger test suites the coverage is often underestimated, see Figure 7. Similarly, the underestimation is also visible in the difference between static and dynamic method coverage of the different master test suites as shown in the project results overview in Table 3.

A method that is called through reflection or by an external library is not detected by the static coverage algorithm. Smaller test suites do not suffer from this issue as the number of overestimated methods is often significantly larger than the amount of underestimated methods.

We observe different tipping points between overestimating and underestimating for JFreeChart and JodaTime. For JFreeChart the tipping point is visible for tests suites with a relative size of 81%, whereas JodaTime reaches the tipping point at a relative size of 25%. We assume this is caused by the relatively low "real" coverage of JFreeChart. We notice that many of JFreeChart's methods that were overestimated by the static coverage algorithm are not covered.

We illustrate the overlap between over- and underestimation with a small synthetic example. Given a project with 100 methods and test suite T. We divide these methods into three groups: **1.** Group A, with 60 methods that are all covered by T, as measured with dynamic coverage. **2.** Group B, with 20 methods that are only called through the Java Reflection API, all covered by T similar to Group A. **3.** Group C, with 20 methods that are not covered by T. The dynamic coverage for T consists of the 80 methods in groups A and B. The static method coverage for T also consists of 80 methods. However, the coverage for Group C is overestimated as they are not covered, and the coverage for Group B is underestimated as they are not detected by the static coverage algorithm.

JFreeChart has a relatively low coverage score compared to the other projects. It is likely that the parts of the code that are deemed covered by static and dynamic coverage will not overlap. However, it should be noted that low coverage does not imply more methods are overestimated. When parts of the code base are completely uncovered, the static method coverage might also not detect any calls to the code base.

> **Finding 6:** The degree of underestimation by the static coverage algorithm partially depends on the number of overestimated methods, as this will compensate for the underestimated methods, and on the number of methods that were called by reflection or external libraries.

Correlation between dynamic and static method coverage. Table 4 shows, for JFreeChart and JodaTime, statistically significant correlations that increase from a low correlation for smaller suites to a moderate correlation for larger suites. One exception is the correlation for JFreeChart"s test suites with 1% relative size. We could not find a explanation for this exception.

We expected that the tipping point between static and dynamic coverage would also be visible in the correlation table. However, this is not the case. Our rank correlation test checks whether two variables follow the same ordering, *i.e.*, if one variable increases, the other also increases. Underestimating the coverage does not influence the correlation when the degree of underestimation is similar for all test suites. As test suites grow in size, they become more similar in terms of included tests. Consequently, the chances of test suites forming an outlier decrease as the size increases.

> **Finding 7:** As test suites grow, the correlation between static and dynamic method coverage increases from low to moderate.

6.2.2 Checkstyle

Figures 6 and 7 show that the static coverage results for Checkstyle's test suites are significantly different from JFreeChart and JodaTime. For Checkstyle, all groups of test suites with a relative size of 49% and lower are split into three subgroups that have around 30%, 70% and 80% coverage. In the following subsections, we analyse the quality of the static coverage for Checkstyle and the predictability of test suite effectiveness.

Quality of static coverage algorithm. To analyse the static coverage algorithm for Checkstyle we compare the static coverage with the dynamic coverage for individual tests (Figure 9a), and inspect the distribution of the static coverage among the different tests (Figure 9b).

We regard the different groupings of test suites in the static coverage spread as a consequence of the few tests with high static method coverage.

Checker tests. Figure 9b shows 1104 tests scoring 30% to 32.5% coverage. Furthermore, dynamic coverage only varied between 31.3% and 31.6% coverage and nearly all tests are located in the `com.puppycrawl.tools.checkstyle.checks` package. We call these tests *checker tests*, as they are all focussed on the checks. A small experiment where we combined the coverage of all 1104 tests, resulted in 31.8% coverage, indicating that all these checker tests almost completely overlap.

Listing 1 shows the structure typical for checker tests: the logic is mostly located in utility methods. Once the configuration for the checker is created, `verify` is called with the files that will be checked and the expected messages of the checker.

```
@Test
public void testCorrect() throws Exception {
 final DefaultConfiguration checkConfig =
     createCheckConfig(
     AnnotationLocationCheck.class);
 final String[] expected = CommonUtils.
     EMPTY_STRING_ARRAY;
 verify(checkConfig, getPath("
     InputCorrectAnnotationLocation.java"),
     expected);
}
```

Listing 1: Test in `AnnotationLocationCheckTest`

> **Finding 8:** Most of Checkstyle's tests are focussed on the checker logic. Although these tests vary in effectiveness, they cover an almost identical set of methods as measured with the static coverage algorithm.

Coverage subgroups and outliers. We notice three vertical groups for Checkstyle in Figure 7 starting around 31%, 71% and 78% static coverage and then slowly curving to the right. These groupings are a result of how test suites are composed and the coverage of the included tests.

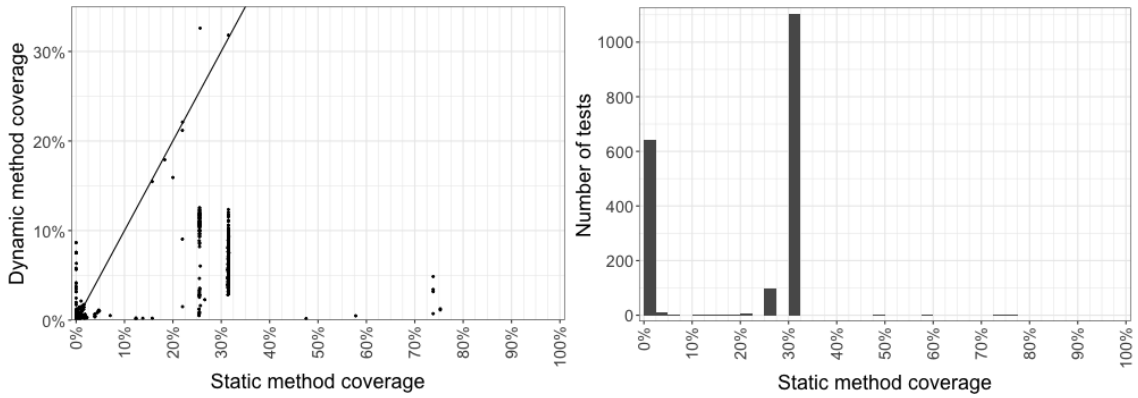
The coverage of the individual tests is shown in Figure 9a. We notice a few outliers at 48%, 58%, 74% and 75% coverage. We construct test suites by randomly selecting tests. A test suite's coverage is never lower than the highest coverage among its individual tests. For example, every time a test with 74% coverage is included, the test suite's coverage will jump to at least that percentage. As test suites grow in size, the chances of including a positive outlier increases. We notice that the outliers do not exactly match with the coverage of the vertical groups. The second vertical for Checkstyle in Figure 7 starts around 71% coverage. We found that if the test with 47.5% coverage, `AbstractChecktest.testVisitToken`, is combined with a 30% coverage test (any of the checker tests), it results in 71% coverage. This shows that only 6.5% coverage is overlapping between both tests. We observe that all test suites in the vertical group at 71% include at least one checker test and `AbstractCheckTest.testVisitToken` and that they do not include any of the other outliers with more than 58%. The most right vertical group starts at 79% coverage. This coverage is achieved by combining any of the tests with more than 50% coverage with a single checker test.

The groupings in Checkstyle's coverage scores are a consequence of the few coverage outliers. We show that these outliers can have a significant impact on a project's coverage score. Without these few outliers, the static coverage for Checkstyle's master test suite would only be 50%

Test suites with low coverage. Figure 9b shows that more than half of the tests have at least 30% coverage. Similarly, Figure 7 shows that all test suites cover at least 31% of the methods. However, there are 763 tests with less than 30% coverage, and no test suites with less than 30% coverage. We explain this using probability theory. The smallest test suite for Checkstyle has a relative size of 1% which are 19 tests. The chance of only including tests with less than 31% coverage $\frac{763}{1875} * \frac{763-1}{1875-1} * \ldots * \frac{763-18}{1875-18} \approx 3 * 10^{-8}$. These chances are negligible, even without considering that a combination of the selected tests might still lead to a coverage above 31%.

Missing coverage. We found that `AbstractCheckTest.testVisitToken` scores 47.5% static method coverage, although it only tests the `AbstractCheck.visitToken` method. Therefore any test calling the visitToken method will have at least 47.5% static method coverage.

160 classes extend `AbstractCheck`, of which 123 override the `visitToken` method. The static method coverage algorithm includes 123 virtual calls when `AbstractCheck.visitToken` is



(a) Static and dynamic method coverage of individual tests. Static coverage of tests below the black line is overestimated, above is underestimated.

(b) Distribution of the tests over the different levels of static method coverage.

Figure 9: Static method coverage scores for individual tests of Checkstyle.

called.The coverage of all `visitToken` overrides combined is 47.5%. Note that the static coverage algorithm also considers constructor calls and static blocks as covered when a method of a class is invoked. We found that only 6.5% of the total method coverage overlaps with `testVisitToken`.

This large overlap between both tests suggests that `visitToken` is not called by any of the check tests. However, we found that the `verify` method indirectly calls `visitToken`. The call `process(File, FileText)`, is not matched with `AbstractFileSetCheck.process(File, List)`. The parameter of type `FileText` extends `AbstractList` which is part of the `java.util` package. During the construction of the static call graph, it was not detected that `AbstractList` is an implementation of the `List` interface because only Checkstyle's source code was inspected. If these calls were detected the coverage of all checker tests would increase to 71%, filling the gap between the two right-most vertical groups in the plots for Checkstyle in both Figures 6 and 7.

> **Finding 9:** Our static coverage algorithm fails to detect a set of calls in the tests for the substantial group of checker tests due to shortcomings in the static call graph. If these the calls were correctly detected, the static coverage for test suites of the same size would be grouped more closely possibly resulting in a more significant correlation.

High reflection usage. Checkstyle applies a visitor pattern on an AST for the different code checks. The `AbstractCheck` class forms the basis of this visitor and is extended by 160 checker classes. These classes contain the core functionality of Checkstyle and consist of 2090 methods (63% of all methods), according to SAT. Running our static coverage algorithm on the master test suite missed calls to 328 methods. Of these methods, 248 (7.5% of all methods) are setter methods. Further inspection showed that checkers are configured using reflection, based on a configuration file with properties that match the setters of the checkers. This large group of methods missed by the static coverage algorithm partially explains the difference between static and dynamic method coverage of Checkstyle's master test suite.

> **Finding 10:** The large gap between static and dynamic method coverage for Checkstyle is caused by a significant amount of setter methods for the checker classes that are called through reflection.

Relation with effectiveness. Checkstyle is the only project for which there is no statistically significant correlation between static method coverage and test suite effectiveness.

We notice a large distance, regarding invocations in the call hierarchy, between most checkers and their tests. There are 9 invocations between `visitToken` and the much used `verify` method.

In addition to the actual checker logic, a lot infrastructure is included in each test. For example, instantiating the checkers and its properties based on a reflection framework, parsing the files and creating an AST, traversing the AST, collecting and converting all messages of the checkers.

These characteristics seem to match those of integration tests. Zaidman *et al.* studied the evolution of the Checkstyle project and arrived at similar findings: "Moreover, there is a thin line between unit tests and integration tests. The Checkstyle developers see their tests more as I/O integration tests, yet associate individual test cases with a single production class by name" [43].

Directness. We implemented the directness measure to inspect whether it would reflect the

presence of mostly integration like tests. The directness is based on the percentage of methods that are directly called from a test. The master test suites of Checkstyle, JFreeChart and JodaTime cover respectively 30%, 26% and 61% of all methods directly. As Checkstyle's static coverage is significantly higher than that of JFreeChart we observe that Checkstyle covers the smallest portion of methods directly from tests. Given that unit tests should be focused on small functional units, we expected a relatively high directness measure for the test suites.

Finding 11: Many of Checkstyle's tests are integration-like tests that have a large distance between the test and the logic under test. Consequently, only a small portion of the code is covered directly.

To make matters worse, the integration-like tests were mixed with actual tests. We argue that integrations tests have different test properties compared to unit tests: they often cover more code, have less assertions, but the assertions have a higher impact, *e.g.*, comparing all the reported messages. These differences can lead to a skew in the effectiveness results.

6.2.3 Dynamic method coverage and effectiveness

We observe in Figure 8 that, within groups of test suites of the same size, test suite with more dynamic coverage are also more effective. Similarly, we observe a moderate correlation between dynamic method coverage and normal effectiveness for all three projects in Table 8.

When comparing test suite effectiveness with static method coverage, we observe a low to moderate correlation for JFreeChart and JodaTime when accounting for size in Table 6, but no statistically significant correlation for Checkstyle. Similarly, only the Checkstyle project does not show a statistically significant correlation between static and dynamic method coverage, as shown in Table 7. We believe this is a consequence of the integration like test characteristics of the Checkstyle project. Due to the large distance between tests and code and the abstractions used in-between, the static coverage is not very accurate.

The moderate correlation between dynamic method coverage and effectiveness suggests there is a relation between method coverage and normal effectiveness. However, the static method coverage does not show a statistically significant correlation with normal effectiveness for Checkstyle. We state that our static method coverage metric is not accurate enough for the Checkstyle project.

6.2.4 Method coverage as a predictor for test suite effectiveness

We found a statistically significant, low correlation between test suite effectiveness and static method coverage for JFreeChart and JodaTime. We evaluated the static coverage algorithm and found that smaller test suites typically overestimate the coverage (Finding 5), whereas for larger test suites the coverage is often underestimated (Finding 6). The tipping point depends on the real coverage of the project. We also found that static coverage correlates better with dynamic coverage as test suite increase in size (Finding 7).

An exception to these observations is Checkstyle, the only project without a statistically significant correlation between static method coverage and both, test suite effectiveness and dynamic method coverage. Most of Checkstyle's tests have nearly identical coverage results (Finding 8) albeit the effectiveness varies. The SAT could calculate static code coverage, however it is less suitable for more complex projects. The large distance between tests and tested functionality (Finding 11) in the Checkstyle project in terms of call hierarchy led to skewed results as some of the must used calls were not resolved (Finding 9). This can be partially mitigated by improving the call resolving.

We consider the inaccurate results of the static coverage algorithm a consequence of the quality of the call graph and the frequent use of Java reflection(Finding 10). Furthermore, the unit tests for Checkstyle show similarities with integration tests.

RQ 2: To what extent is static coverage a good predictor for test suite effectiveness?

First, we found a moderate to high correlation between dynamic method coverage and effectiveness for all analysed projects which suggests that method coverage is a suitable indicator. The projects that showed a statistically significant correlation between static and dynamic method coverage also showed a significant correlation between static method coverage and test suite effectiveness. Although the correlation between test suite effectiveness and static coverage was not statistically significant for Checkstyle, the coverage score on project level provided a relatively good indication of the project's real coverage. Based on these observations we consider coverage suitable as a predictor for test effectiveness.

6.3 Practicality

A test quality model based on the current state of the metrics would not be sufficiently accurate.

Although there is evidence of a correlation between assertion count and effectiveness, the assertion count of each project's master test suite

did not map to the relative effectiveness of each project. Each of the analysed projects had on average a different number of assertions per test. Further improvements to the assertion count metric, *e.g.*, including the strength of the correlation, are needed to get more usable results.

The static method coverage could be used to evaluate effectiveness to a certain extent. We found a low to moderate correlation for two of the project between effectiveness and static method coverage. Furthermore, we found a similar correlation between static and dynamic method coverage. The quality of the static call graph should be improved to better estimate the real coverage.

We did not investigate the quality of these metrics for other programming languages. However, the SAT supports call graph analysis and identifying assertions for a large range of programming languages, facilitating future experiments.

We encountered scenarios for which the static metrics gave imprecise results. If these sources of imprecision would be translated to metrics, they could indicate the quality of the static metrics. An indication of low quality could suggest that more manual inspection is needed.

6.4 Internal threats to validity

Static call graph. We use the static call graph constructed by the SAT, for both metrics. We found several occurrences where the SAT did not correctly resolve the call graph. We fixed some of the issues encountered during our analysis. However, as we did not manually analyse all the calls, this remains a threat to validity.

Equivalent mutants. We treated all mutants that were not detected by the master test suite as equivalent mutants, an approach often used in literature [35, 24, 45]. There is a high probability that this resulted in overestimating the number of equivalent mutants, especially for JFreeChart where a large part of the code is simply tested. In principle, this is not a problem as we only compare the effectiveness of sub test suites. However, our statement on the order of the master's tests suite effectiveness is vulnerable to this threat as we did not manually inspect each mutant for equivalence.

Accuracy of analysis. We manually inspected large parts of the Java code of each project. Most of the inspections were done by a single person with four years of experience in Java. Also, we did not inspect all the tests. Most tests were selected on a statistic driven-basis, *i.e.*, we looked at tests that showed high effectiveness but low coverage, or tests with a large difference between static and dynamic. To mitigate this, we also verified randomly selected tests. However, the chances of missing relevant source of imprecision remains a threat to validity.

6.5 External threats to validity

We study three open source Java projects. Our results are not generalisable to projects using other programming languages. Also, we only included assertions provided by JUnit. Although JUnit is the most popular testing library for Java, there are testing libraries possibly using different assertions [44]. We also ignored mocking libraries in our analysis. Mocking libraries provide a form of assertions based on the behaviour of units under test. These assertions are ignored by our analysis, albeit they can lead to an increase in effectiveness.

6.6 Reliability

Tengeri *et al.* compared different instrumentation techniques and found that JaCoCo produces inaccurate results especially when mapped back to source code [39]. The main problem was that JaCoCo did not include coverage between two different sub-modules in a Maven project. For example, a call from sub-module A to sub-module B is not registered by JaCoCo because JaCoCo only analyses coverage on a module level. As the projects analysed in this thesis do not contain sub-modules, this JaCoCo issue is not applicable to our work.

7 Related work

We group related work as follows: test quality models, standalone test metrics, code coverage and effectiveness, and assertions and effectiveness.

7.1 Test quality models

We compare the TQM [18] we used, as described in Section 2.2 with two other test quality models. We first describe the other models, followed by a motivation for the choice of a model.

STREW. Nagappan introduced the Software Testing and Reliability Early Warning (STREW) metric suite to provide "an estimate of post-release field quality early in software development phases [34]." The STREW metric suite consists of nine static source and test code metrics. The metric suite is divided into three categories: Test quantification, Complexity and OO-metrics, and Size adjustment. The test quantifications metrics are the following: **1.** Number of assertions per line of production code. **2.** Number of tests per line of production code. **3.** Number of assertion per test. **4.** The ratio between lines of test code and production code, divided by the ratio of test and production classes.

TAIME. Tengeri *et al.* introduced a systematic approach for test suite assessment with a focus on code coverage [38]. Their approach, Test Suite Assessment and Improvement Method (TAIME), is intended to find improvement points and guide

the improvement process. In this iterative process, first, both the test code and production code are split into functional groups and paired together. The second step is to determine the granularity of the measures, start with coarse metrics on procedure level and in later iterations repeat on statement level. Based on these functional groups they define the following set of metrics:

Code coverage calculated on both procedure and statement level.

Partition metric "The Partition Metric (PART) characterizes how well a set of test cases can differentiate between the program elements based on their coverage information [38]".

Tests per Program how many tests have been created on average for a functional group.

Specialisation how many tests for a functional group are in the corresponding test group.

Uniqueness what portion of covered functionality is covered only by a particular test group.

STREW, TAIME and TQM are models for assessing aspects of test quality. STREW and TQM are both based on static source code analysis. However, STREW lacks coverage related metrics compared to TQM. TAIME is different from the other two models as it does not depend on a specific programming language or xUnit framework. Furthermore, TAIME is more an approach than a simple metric model. It is an iterative process that requires user input to identify functional groups. The required user input makes it less suitable for automated analysis or large-scale studies.

7.2 Standalone test metrics

Bekerom investigated the relation between test smells and test bugs [41]. He built a tool using the SAT to detect a set of test smells: Eager test, Lazy test, Assertion Roulette, Sensitive Equality and Conditional Test Logic. He showed that classes affected by test bugs score higher on the presence of test smells. Additionally, he predicted classes that have test bugs based on the eager smell with a precision of 7% which was better than random. However, the recall was very low which led to the conclusion that it is not yet usable to predict test bugs with smells.

Ramler *et al.* implemented 42 new rules for the static analysis tool PDM to evaluate JUnit code [37]. They defined four key problem areas that should be analysed: Usage of the xUnit test framework, implementation of the unit test, maintainability of the test suite and testability of the SUT. The rules were applied to the JFreeChart project and resulted in 982 violations of which one-third was deemed to be some symptom of problems in the underlying code.

7.3 Code coverage and effectiveness

Namin *et al.* studied how coverage and size independently influence effectiveness [35]. Their experiment used seven Siemens suite programs which varied between 137 and 513 LOC and had between 1000 and 5000 test cases. Four types of code coverage were measured: block, decision, C-Use and P-Use. The size was defined by the number of tests and effectiveness was measured using mutation testing. Test suites of fixed sizes and different coverage levels were randomly generated to measure the correlation between coverage and effectiveness. They showed that both coverage and size independently influence test suite effectiveness.

Another study on the relation between test effectiveness and code coverage was performed by Inozemtseva and Holmes [24]. They conducted an experiment on a set of five large open source Java projects and accounted for the size of the different test suites. Additionally, they introduced a novel effectiveness metric, normalized effectiveness. They found moderate correlations between coverage and effectiveness when size was accounted for. However, the correlation was low for normalized effectiveness.

The main difference with our work is that we used static source code analysis to calculate method coverage. Our experiment set-up is similar to that of Inozemtseva and Holmes except that we chose a different set of data points which we showed as more representative.

7.4 Assertions and effectiveness

Kudrjavets *et al.* investigated the relation between assertions and fault density [28]. They measured the assertion density, *i.e.*, number of assertions per thousand lines of code, for two components of Microsoft Visual Studio written in C and C++. Additionally, real faults were taken from an internal bug database and converted to fault density. Their result showed a negative relation between assertion density and fault density, *i.e.*, code that had a higher assertion density has a lower fault density. Instead of assertion density we focussed on the assertion count of Java projects and used artificial faults, *i.e.*, mutants.

Zhang and Mesbah [45] investigated the relationship between assertions and test suite effectiveness. They found that, even when test suite size was controlled for, there was a strong correlation between assertion count and test effectiveness. Our results overlap with their work as we both found a correlation between assertion count and effectiveness for the JFreeChart project. However, we showed that this correlation is not always present as both Checkstyle and JodaTime showed different results.

8 Conclusion

We analysed the relation between test suite effectiveness and metrics, assertion count and static method coverage, for three large Java projects, Checkstyle, JFreeChart and JodaTime. Both metrics were measured using static source code analysis. We found a low correlation between test suite effectiveness and static method coverage for JFreeChart and JodaTime and a low to moderate correlation with assertion count for JFreeChart. We found that the strength of the correlation depends on the characteristics of the project. The absence of a correlation does not imply that the metrics are not useful for a TQM.

Our current implementation of the assertion count metric only shows promising results when predicting test suite effectiveness for JFreeChart. We found that simply counting the assertions for each project gives results that do not align with the relative effectiveness of the projects. The project with the most effective master test suite had a significantly lower assertion than the other projects. Even for sub test suites of most project, the assertion count did not correlate with test effectiveness. Incorporating the strength of an assertion could lead to better predictions.

Static method coverage is a good candidate for predicting test suite effectiveness. We found a statistically significant, low correlation between static method coverage and test suite effectiveness for most analysed projects. Furthermore, the coverage algorithm is consistent in its predictions on a project level, *i.e.*, the ordering of the projects based on the coverage matched the relative ranking in terms of test effectiveness.

8.1 Future work

Static coverage. Landman *et al.* investigated the challenges for static analysis of Java reflection [30]. They identified that is at least possible to identify and measure the use of hard to resolve reflection usage. Measuring reflection usage could give an indication of the degree of underestimated coverage. Similarly, we would like to investigate whether we can give an indication of the degree of overestimation of the project.

Assertion count. We would like to investigate further whether we can measure the strength of an assertion. Zhang and Mesbah included assertion coverage and measured the effectiveness of different assertion types [45]. We would like to incorporate this knowledge into the assertion count. This could result in a more comparable assertion count on project level.

Deursen *et al.* described a set of test smells including the eager tests, a test the verifies too much functionality of the tested function [42]. We found a large number of tests in the JodaTime project that called the function under test several times. For example, JodaTime's `test_wordBased_pl_regEx` test checks 140 times if periods are formatted correctly in Polish. These eager tests should be split into separate cases that test the specific scenarios.

8.2 Acknowledgements

We would like to thank Prof. Serge Demeyer for his elaborate and insightful feedback on our paper.

References

[1] Checkstyle. `https://github.com/checkstyle/checkstyle`. Accessed: 2017-07-15.

[2] Checkstyle team. `http://checkstyle.sourceforge.net/team-list.html`. Accessed: 2017-11-19.

[3] Code cover. `http://codecover.org/`. Accessed: 2017-07-15.

[4] JaCoCo. `http://www.jacoco.org/`. Accessed: 2017-07-15.

[5] JFreeChart. `https://github.com/jfree/jfreechart`. Accessed: 2017-07-15.

[6] JodaTime. `https://github.com/jodaorg/joda-time`. Accessed: 2017-07-15.

[7] JUnit. `http://junit.org/`. Accessed: 2017-07-15.

[8] MAJOR mutation tool . `http://mutation-testing.org/`. Accessed: 2017-07-15.

[9] muJava mutation tool. `https://cs.gmu.edu/~offutt/mujava/`. Accessed: 2017-07-15.

[10] PIT+. `https://github.com/LaurentTho3/ExtendedPitest`. Accessed: 2017-07-15.

[11] PIT fork. `https://github.com/pacbeckh/pitest`. Accessed: 2017-07-15.

[12] PIT mutation tool . `http://pitest.org/`. Accessed: 2017-07-15.

[13] R's Kendall package. `https://cran.r-project.org/web/packages/Kendall/Kendall.pdf`. Accessed: 2017-07-15.

[14] SLOCCount. `https://www.dwheeler.com/sloccount/`. Accessed: 2017-07-15.

[15] TIOBE-Index. `https://www.tiobe.com/tiobe-index/`. Accessed: 2017-07-15.

[16] Tiago L. Alves and Joost Visser. Static estimation of test coverage. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 55–64, 2009.

[17] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 21–30, 2014.

[18] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans. Software Eng.*, 40(11):1100–1125, 2014.

[19] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[20] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 85–103, 2007.

[21] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, pages 30–39, 2007.

[22] Ferenc Horváth, Bela Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely, and Tibor Gyimóthy. Test suite evaluation using code coverage based metrics. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, October 9-10, 2015.*, pages 46–60, 2015.

[23] David C Howell. *Statistical methods for psychology*. Cengage Learning, 2012.

[24] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 435–445, 2014.

[25] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[26] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665, 2014.

[27] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 147–156, 2016.

[28] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, pages 204–212, 2006.

[29] Tobias Kuipers and Joost Visser. A tool-based methodology for software portfolio monitoring. In *Software Audit and Metrics, Proceedings of the 1st International Workshop on Software Audit and Metrics, SAM 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, pages 118–128, 2004.

[30] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection: literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 507–518, 2017.

[31] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. Assessing and improving the mutation testing practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 430–435, 2017.

[32] András Márki and Birgitta Lindström. Mutation tools for java. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1364–1415, 2017.

[33] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[34] Nachiappan Nagappan. *A Software Testing and Reliability Early Warning (Strew) Metric Suite*. PhD thesis, North Carolina State University, 2005.

[35] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 57–68, 2009.

[36] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365, 2016.

[37] Rudolf Ramler, Michael Moser, and Josef Pichler. Automated static analysis of unit test code. In *First International Workshop on Validating Software Tests, VST@SANER 2016, Osaka, Japan, March 15, 2016*, pages 25–28, 2016.

[38] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, David Havas, and Tibor Gyimóthy. Beyond code coverage - an approach for test suite assessment and improvement. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–7, 2015.

[39] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Negative effects of bytecode instrumentation on java

source code coverage. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 225–235, 2016.

[40] Paco van Beckhoven. Assessing test suite effectiveness using static analysis. Master's thesis, University of Amsterdam, 2017.

[41] Kevin van den Bekerom. Detecting test bugs using static analysis tools. Master's thesis, University of Amsterdam, 2016.

[42] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[43] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*, pages 220–229, 2008.

[44] Ahmed Zerouali and Tom Mens. Analyzing the evolution of testing library usage in open source java projects. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 417–421, 2017.

[45] 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 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 214–224, 2015.

[46] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.