

Automated Mutation Testing: Review of Methods and Efficacy

Ian Verbeck

Abstract– With the increased demand for continuous integration in the software development process, test driven development and automated testing techniques have become widely employed in the areas of computer science and software engineering since before the year 2000. The need for automated testing systems and techniques have led to many advancements in unit testing frameworks such as JUnit, and have been extended and adapted to parts of the continuous integration pipeline. This paper provides a brief overview of automated mutation testing techniques and its potential use and misuse in the software development process.



1 INTRODUCTION

The areas of software engineering and computer science are filled with specializations which enhance the quality of the end products and applications available to the end user. Software testing falls underneath multiple categories within the software development process, and is an integral part of assuring quality and reliability when an application goes into production. Software systems around the world rely on tests to ensure that when code goes into use it can be executed safely, and detecting faults in these systems can save millions of dollars, hours, and lives. Various testing techniques ensure that at each stage of production, the source code is written correctly and does not exhibit unwanted behavior that may result in the failure of these systems.

Testing Overview

Various testing techniques may be employed to ensure the quality of software and its interaction with other systems, various users, and within the system under test itself. Two software testing paradigms are employed to ensure that testing is efficient

and effective– white box testing and black box testing. White box testing includes tests which are built and ran with knowledge of the underlying source code. techniques in this paradigm include unit testing, integration testing, and mutation testing. Black box testing is the test of inputs and outputs to the system, and include techniques such as acceptance testing, regression testing, and functional testing. This report focuses on the white box testing techniques used to verify the source code and its associated tests.

Mutation Testing

Mutation testing– modification of source code to verify the efficacy of existing testing suites– is inherently interesting in the fields of test-driven development and continuous integration. These two fields have become more prominent since the turn of the century, as unit testing frameworks have advanced to include many automated techniques and adaptations. The answer to the question of how the tests will be tested themselves lies in the methodology behind mutation testing, and may be shown to be as significant as testing the code itself. In the

following sections, a review of mutation testing methodology is examined.

2 PROBLEM STATEMENT

Unit testing frameworks such as jUnit have proliferated into the software development process, and have become an integral part of related automated processes as part of the continuous integration pipeline. Automated unit testing gives the ability to ensure that changes in code are not met with unwanted and unexpected changes in production, and these suites can be executed upon release to take the manual burden off the programmers whose primary concern is writing code that meets the expectations of the end users and requirements of the application being written.

Coverage Metrics

Evaluation of the testing suites associated with particular projects requires metrics which can be measured quantitatively with the purpose of exposing areas of the code which are either not tested properly or fail to perform underneath strenuous testing. This notion then leads to two categories of testing metrics— *those relating to the source code's ability to pass the tests provided, and those which evaluate the test suite's ability to efficiently and effectively test the source code*. Those associated with the former include the number of tests which pass, fail, and exit with error. These are the baseline unit testing metrics which expose which parts of the source code are do not meet the testing suite's standards. Those associated with the latter include test-suite duration, code coverage, and mutation testing metrics. Many independent tools, as well as plugins for IDEs and text editors, have been created to generate these metrics. In this paper, two are of importance— JaCoCo and PITest. JaCoCo generates coverage metrics,

and PITest generates both coverage and mutation-related metrics.

JaCoCo

Code coverage refers to the amount of the source code that is under test, including statements, branches, and conditions which are covered in the testing suite. JaCoCo, used in this context as a plugin for the Eclipse IDE, is a free tool which generates coverage metrics for Java projects. JaCoCo analyzes source code statically and provides an output of coverage-related metrics including line(instruction) coverage, method coverage, and overall project and class coverages. it is representative of coverage tools for other languages, such as Bullseye for C languages and Coverage.py for python. *While traditional coverage metrics generated by these tools provide a look at how the code is performing under existing testing suites, they do not evaluate the quality of the testing suites themselves.* Mutation testing techniques and metrics provide a deeper look into how the testing suites are performing on the source code. In this case, coverage tools measure *testing breadth* while mutation testing tools measure *testing depth*.

PITest

PIT mutation testing is another free tool available as an Eclipse plugin which generates mutated code in order to test the existing testing suites written by the developers and testers. It covers traditional testing metrics, but more interestingly generates mutation metrics which reach granularities of project, package, class, and method. An easy to navigate HTML file is produced which gives the software developers the ability to see where and how the tests are failing the production code, as opposed to how the production code is failing the tests. This is the tool used in this

report to generate and compare mutation metrics across projects and their revisions.

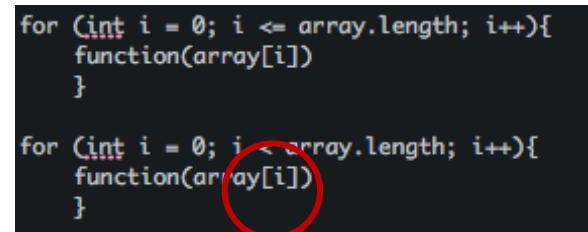
3 METHODOLOGIES

Mutation testing is an interesting, and relatively new, technique in the field of software engineering. It is often not thought of as a typical part of the continuous integration process (as is its unit testing counterpart), though it may prove to be just as beneficial as other testing techniques. The following is a brief overview of mutation testing, the PIT process, and the types of mutation tests involved in this report.

Description and Overview

Mutation testing is based on a simple process. Parts of the source code under test are changed in various ways to produce segments of code which should cause existing tests to fail—mutants. If they pass when they should have failed, it is an indicator that the tests are not fully testing the intended functionality of the source code which has been changed, and are said to let the mutant survive. If a test fails when it should have, that test is said to kill off that particular mutant, and indicates it is testing the intent of the source code correctly. The total number of mutants killed divided by the total number of mutants generated is the mutant score (mutant coverage), the main metric in mutation testing. PIT provides mutation coverage metrics for projects, packages, and classes. A common example of a mutation test is a conditionals boundary test. This will change the boundary of a conditional to see if the test case covering it fails when the code is running beyond the intended point. If the test does not fail, and a parameter is passed to the encompassing method in production which allows for an extra and unwanted iteration, this code may fail in production even though it passed its

automated testing procedure. The following picture demonstrates the difference between the mutated code and the non-mutated code.



```
for (int i = 0; i <= array.length; i++){  
    function(array[i])  
}  
  
for (int i = 0; i < array.length; i++){  
    function(array[i])  
}
```

As the red circle indicates, the mutated code changes the boundary from `<=` to `<`. In the case where the last element of the array passed to this function is one that is not intended to be passed to the inner function as a parameter, it is important to ensure that the unit test for this for loop fails upon mutation. If it does not, the corresponding unit test is allowing for unwanted behavior to occur in production by passing (and indicating working source code) even when the final iteration is executed. This is a great example of how mutation testing is really a way of testing the test suites, rather than the source code itself.

Mutation Testing Types

The types of mutation testing activated for the PIT tests ran in this report are the default setting for the PIT Eclipse plugin:

- Conditionals Boundary Mutator
- Increments Mutator
- Void Method Call Mutator
- Invert Negatives Mutator
- Return Value Mutator
- Math Mutators
- Negate Conditionals Mutator

In the following section, the summaries for three open source projects and their coverage and mutation metrics are discussed, as well as a deeper look into how mutants can be killed by writing new tests or modifying existing tests.

4 RESULTS

In order to evaluate the efficacy of the different testing metrics mentioned in previous sections, three different open source projects were taken from Github.com and imported into eclipse where JaCoCo and PIT were run to provide insight into their testing suites. Additionally, a new test class was written for one project in an attempt to kill mutants associated with that project, and provide additional insight into the use of mutation testing to find issues with existing testing suites.

Overview of Metrics for 3 Projects

The first of three projects evaluated in this report is Moshi, a JSON library for Java that enables easier parsing of JSON data into Java objects. Moshi has approximately 39,564 instructions in its source code, 89.7 % of which is covered by unit tests. The PIT testing tool generated 1,703 mutation tests, 123 of which did not have coverage and 199 of which survived, resulting in a mutant killing rate of 87.4%. This was the most exhaustively tested project, as well as the largest, and is a great example of the PIT tool. The two mutations with the most survivors for Moshi were Return Values Mutator and Math Mutators. The following picture shows the example breakdown of Moshi's mutation coverage for its main package.

Breakdown by Class

Name	Line Coverage	Mutation Coverage
AdapterMethodsFactory.java	95% 126/132	93% 64/69
ArrayJsonAdapter.java	86% 18/21	92% 11/12
ClassFactory.java	33% 11/33	33% 3/9
ClassJsonAdapter.java	81% 38/47	92% 11/12
CollectionJsonAdapter.java	87% 20/23	88% 7/8
JsonAdapter.java	85% 35/41	95% 21/22
JsonReader.java	92% 67/73	93% 28/30
JsonScope.java	83% 10/12	100% 5/5
JsonUtf8Reader.java	97% 617/637	90% 384/425
JsonUtf8Writer.java	97% 202/208	82% 116/142
JsonValueReader.java	99% 202/205	98% 149/152
JsonValueWriter.java	96% 150/156	79% 94/119
JsonWriter.java	98% 55/56	83% 25/30
LinkedHashMap.java	80% 314/394	53% 132/251
MapJsonAdapter.java	89% 25/28	91% 10/11
Moshi.java	90% 134/149	92% 76/83
StandardJsonAdapters.java	72% 50/69	88% 22/25
Types.java	84% 113/134	86% 77/90

This is exemplary of PIT's output readability and usability, with each of the classes available for further look-in to see where tests aren't failing and which types of mutants are living. Overall coverage and mutation coverage within 3% indicates that the tests written for Moshi cover depth nearly to the same extent as breadth, and that it has a test suite with health similar to its overall coverage.

The second project evaluated in this report is Dagger, a framework for dependency injection started by Google. Dagger has approximately 11,122 instructions in its source code, 81.4 % of which is covered by unit tests. The PIT testing tool generated 1557 mutation tests, 163 of which did not have coverage and 41 of which survived, resulting in a mutant killing rate of 89.6%. The two mutations with the most survivors for Dagger were Void Method Call Mutator and Negate Conditional Mutator. A mutation coverage rate over 8% higher than the overall code coverage rate indicates that the testing suite for this project is healthy in terms of depth. As this is an open source project started by google, it is to be expected the tests created have been re-written over time to increase performance and reliability in the software. It also exemplifies the fact that *increased coverage does not always*

lead to increased testing depth and efficiency, as it has less overall coverage than Moshi but still has higher mutation coverage.

The final project evaluated in this report is ScribeJava, an OAuth client library. In order to evaluate the ability to kill mutants in this code, an extra testing class was written and run after the initial metrics were taken. ScribeJava's initial version before the extra test class addition had approximately 12,948 instructions in its source code, 59.3 % of which is covered by unit tests. The PIT testing tool generated 1,145 mutation tests, 641 of which did not have coverage and 199 of which survived, resulting in a mutant killing rate of 80.2%. The two mutations with the most survivors were the Negate Conditionals Mutator and Math Mutator. As this project had the lowest mutation coverage of the three projects, additional tests were written in order to increase the overall coverage as well as the mutation coverage and to compare the results after those tests were incorporated into the PIT process.

Overview of Revised Metrics

The additional tests added in order to kill off mutants in the ScribeJava testing suite is comprised of 19 new unit tests spanning 287 new lines of code. Mutants to kill were picked randomly, and were taken from a variety of different classes. With the 19 new unit tests, 30 additional mutants were killed, resulting in a new mutant coverage of 87.1% and a new overall coverage of 60.4%. This is an important discovery in terms of how unit tests can improve mutation coverage at a higher rate than overall coverage. An increase in mutation coverage of nearly 6.9%, compared to an increase in overall coverage of 1.1%, shows how mutation

testing may be used to increase a testing suite's depth efficiently by changing tests or adding tests to kill mutants. This is possible because *one new unit test may not increase coverage at all, but at the same time may kill off multiple mutants*. List of mutants killed:

- Increments Mutant: 2
- Void Method Call Mutant: 6
- Return Values Mutant: 5
- Math Mutants: 4
- Negate Conditionals Mutant: 13

Tests used to kill off mutants generally fall into two categories: those which are useful, and those which are naive. One of each will be discussed here.

Testing Examples

While tests may be written to kill all kinds of mutants, the question remains whether or not some of these tests are necessary for a testing suite, and whether the increased complexity and overhead is worth it. In examining this question, two examples of the additional tests written for ScribeJava are as follows: The first is a test written to kill a Void Method Call mutant generated by removing the method which throws an exception when the function `apikey()` is passed an empty string. I argue that this test is effective, as it ensures that an empty value is not passed into a function which may operate falsely if that is the case. In this case, it is important the developers can be assured the rest of the `apikey()` function can be correctly executed.

```
@Test //KILLS 1 MUTANT... assures exception is called when apikey() is passed empty string
public void line42RemovedCall() {
    Exception ex = null;
    ServiceBuilder sBuilder = new ServiceBuilder("APIKEY");
    try {
        ServiceBuilder sBuilderReturned = sBuilder.apiKey("");
    } catch (Exception e) {
        ex = e;
    }
    if (ex.equals(null)) {
        Assert.fail();
    }
}
```

An example of a test written to kill a mutant out of naivety, or with the purpose of killing a mutant rather than testing the suite itself,

is a test written to ensure the hash of a parameter does not equal 0.

```
@Test //KILLS MUTANT.. Assumes hash doesn't == 0, which it shouldn't (would make hash very guessable)
public void line45ReplacedIntegerReturnWithTernary() {
    OAuthAccessToken oToken = new OAuthAccessToken("Random", "String");
    int actual = oToken.hashCode();
    assertEquals(0, actual);
}
```

While the hashing algorithm invoked on this object should not return 0, this tests the imported hashing algorithm itself rather than the test upon the source code of the function under test. This test, while increasing mutation coverage, does not increase the quality of the testing suite.

5 RELATED WORK

Much work has been done to evaluate the efficacy of mutation testing techniques. Three works discussed here, each covering different topics, are: *Predictive Mutation Testing* (published by IEEE), *Should Software Testers Use Mutation Analysis to Augment a Test Set?* by Ben H. Smith and Lauren Williams at North Carolina State University, and *Investigating the Effectiveness of Mutation Testing Tools in the Context of Deep Neural Networks* by Nour Chetouane, Lorenz Klampfl, and Franz Wotawa.

The first work examines the issue of the efficiency of mutation testing, as it is naturally an exhaustive process which may add too much overhead to the continuous integration pipeline to be worth its effects. The paper argues that predictive mutation testing may be a solution to this issue, and that by creating a classification model which predicts the kill-ability of mutants without executing the tests to kill them, mutation testing can be expedited without compromising efficiency. The study found that predictive mutation testing techniques can improve efficiency at rates up to and surpassing 150 times the usual mutation

testing efficiency with minor losses in accuracy.

The second work examines mutation testing in a broader view and in terms of efficacy rather than efficiency. The study found that mutation testing can indeed be used effectively in order to improve coverage metrics, as well as provide a layout for building new and improved testing suites to test the source code of various projects. The findings of this report are therefore in congruence with the findings of this one in that mutation testing is indeed effective. In addition, this report finds that the tools used in executing automated mutation tests play a role in the reliability of the results evaluated after testing.

The third report looks at mutation testing through the lens of machine learning, particularly deep neural networks. The injection of faults into other projects, not those only comprising of traditional software applications, is an interesting look into how techniques from one area of computer science may be applied to others in hopes of increasing their accuracy and efficiency. In this paper, it is found that creating a mutation testing framework for deep learning networks can increase the quality of test data used in machine learning applications. This can then be used to increase the quality of deep learning and other machine learning applications to produce results with a higher level of insight into large amounts of data.

6 CONCLUSIONS

In this report, three different open source projects taken from the Github public repositories were evaluated using JaCoCo and PIT mutation testing to provide insight

into the relationship between traditional coverage metrics and mutation coverage metrics. It was found that these two metrics, and their variance, are not necessarily correlated, and rather can be used in conjunction to evaluate the efficacy of an application's testing suite. Additionally, a new testing class was written comprising 19 new unit tests for ScribeJava, which increased the mutation coverage of this project substantially while leaving its overall coverage metrics nearly the same. Insight into the results showed that killing mutants can be done with effectiveness in mind or out of naivety, similar to writing unit tests with the sole purpose of increasing traditional coverage metrics.

Mentioned briefly in this report is the concern of the natural exhaustive-ness of mutation testing and the increased overhead it brings, especially in the context of continuous integration. Metrics obtained using the PIT plugin for Eclipse showed overall run times of 127, 54, and 52 seconds for projects of length 39,564 lines, 11,122 lines, and 12,948 lines, respectively. This results in an average of 255 lines per second to run the PIT mutation testing tool. This metric would likely be increased in a testing environment with efficient hardware and software systems in place, but further research should be done in order to compare the efficiency of mutation testing with traditional unit testing in the presence of large scale software systems to determine the probability of mutation testing's integration into continuous environments.

REFERENCES

Bullseye
<https://www.bullseye.com/>

Coverage.py
<https://coveage.readthedocs.io/en/v4.5.x/>

PITest
<https://pitest.org/>

JaCoCo
<https://www.eclemma.org/jacoco/>

Eclipse
<https://www.eclipse.org/>

GitHub
<https://github.com/>

Moshi
<https://github.com/square/moshi>

Dagger
<https://github.com/google/dagger>

ScribeJava
<https://github.com/scribejava/scribejava>

Referenced Work 1
J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia and L. Zhang, "Predictive Mutation Testing," in *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898-918, 1 Sept. 2019.

Referenced Work 2
Smith, Ben & Williams, Laurie. (2009). Should software testers use mutation analysis to augment a test set?. *Journal of Systems and Software*. 82. 1819-1832. 10.1016/j.jss.2009.06.031.

Referenced Work 3
Chetouane N., Klampfl L., Wotawa F. (2019) Investigating the Effectiveness of Mutation Testing Tools in the Context of Deep Neural Networks. In: Rojas L., Joya G., Catala A. (eds) *Advances in*

**Computational Intelligence. IWANN
2019. Lecture Notes in Computer Science,
vol 11506. Springer, Cham**