**Mutation Testing Research**

PIT Mutation Testing

Mehrnoush Sotoudeh  
Computer Science   
University of Texas at Dallas  
 Dallas Texas United States  
 mxs172630@utdallas.edu

Marian Lusk  
 Computer Science  
University of Texas at Dallas  
 Dallas Texas United States  
 mll150230@utdallas@.edu

ABSTRACT

This paper provides an overview on mutation testing; specifically, on the existing tools such as PIT, which provides gold standard test coverage for Java and JVM. Furthermore, this paper goes over the research that was done over this topic and the future study plan that will take place later this semester. Additionally, this paper will explain the necessary steps for mutation testing, the importance of mutation score and the different categories of mutation operands.

**1 Tools**

There are various mutation testing tools available for java such as Mujava, Mueclipse, PIT, Jester, and etc. However, in this proposal, we will focus on the PIT mutation testing tool. One of the most important features of the PIT tool is that it generates mutants at the bytecode level. Bytecode mutators generate mutations by making changes to compiled bytecode using libraries such as ASM.

PIT tool runs unit tests against the modified code. Since the modified code generates different results, it should cause the unit test to fail. In the case where the unit test does not fail, then there may be issue with the unit test[1].

1.1 Strategies

Since we are using the PIT mutation testing tool, the test selection strategy that is going to be used is Coverage based [2]. This strategy allows for automation testing. Tests are chosen by initially measuring the line coverage, instruction coverage or block coverage. It is worth noting that only those tests that test lines or blocks which has the mutant can be run against it.

**2 Study Plan**

Our initial study plan leads us to the ASM Framework through the ASM documentation available online and ASM tutorials [3]. ASM is a JAVA ByteCode Framework and our project will be manipulating those bytes into mutants of our own design. Prior to submitting our proposal, we started our research on the JavaAgent on-the-fly code instrumentation, which basically allows for byte code to be altered once it is loaded into the JVM. We also worked on researching the Maven build system, which allowed us to have a general overview of how these components will work together to build our test system. Together, we are also analyzing the GitHub implementation of the current mutation operators so that we can better understand how we will implement AOD, ROR, and AOR.

**2.2 Subjects**

While completing our initial research, we selected suitable test subjects. Our test subjects met the minimum code line standard, but also had an adequate amount of relational operators and arithmetic expressions available for mutation testing. In addition to looking for a variety of relational operators and arithmetic expressions, we also looked for a variety of statements in the program. For instance, a program with only if statements or only for loops would allow for us test our mutators, but a program with a combination of the two would allow us to see more fully the effects mutators would have on a program. [5][6][7][8][9]

**3  Implementation**

The first step in mutation testing is applying artificial changes based on mutation operators to generate mutants. It is important to consider that each mutant has to have only one artificial bug. The next step is to run the test suits against mutants to see if it fails or passes. As mentioned before, if a test suite does not fail, it may indicate that the test suite has issues. The last step is to compute the mutation score.

3.1**Mutation Score**

In mutation testing it is considered a fact that faults of mutations are present in the code. Now if the test fails, the fault is captured, and the mutation is killed. However, if the test passes, it means fault was not captured and the mutation lives. Therefore, a high-quality test is the one that has a higher percentage of killed mutants. In our project we will be focusing capturing on the mutation score or our test subjects versus the speed of our algorithm. The mutation score is obtained by calculating the percentage of killed mutants with respect to the total mutants (not including the Equivalent mutants) [10].

3.2**Mutation Operators**

A mutation is a single syntactic change that is made to a statement, for instance changing a plus sign to a minus sign. A mutation operator is a mutation that is defined to affect a particular operator, such as arithmetic or relational. The current PIT tool applies a set a predefined mutation operator to code for testing, however they did not implement AOD, ROR, or AOR. By not implementing these three mutators, developers are missing valuable testing. Developers might test their projects with PIT and have a high MS(T), but since it is missing three important mutations they are unable to find some of their critical bugs/faults. Our project will be filling in the current missing gap and allow testing to be more complete once combined with the already implemented PIT mutators.

**CCS CONCEPTS**

• Software Engineering → Testing and Verification;

**KEYWORDS**

Mutation Testing, PIT, Maven, JAVA ByteCode

**ACKNOWLEDGMENTS**

We would like to thank Professor Lingming Zhang for his guidance.

**REFERENCES**

[1] Real world mutation testing. Retrieved September 19, 2018 from http://pitest.org/

[2] Mutation testing systems for Java compared. Retrieved September 19, 2018 from http://pitest.org/java\_mutation\_testing\_systems/

[3] ASM Framework. Retrieved September 19, 2019 from https://asm.ow2.io/.

[4] Guru99. Retrieved from https://www.guru99.com/mutation-testing.html.

[5] The Extensible Compiler. Retrieved September 19, 2018 from https://github.com/copton/xtc/blob/77b93130afb4c5698332ca125834cc3b1ad0ec47/src/xtc/lang/JavaUnitTests.java.

[6] Random Access File Test. Retrieved September 19, 2018 from https://github.com/sdugit/apache-harmony\_7025/blob/5b53d279675709a4beb54974c72e257c1262da2c/luni/src/test/api/common/org/apache/harmony/luni/tests/java/io/RandomAccessFileTest.java.

[7] Java Testing Example. Retrieved September 19, 2018 from https://github.com/testdouble/java-testing-example.

[8] Alibaba Java Diagnostic tool. Retrieved September 19, 2018 from https://github.com/alibaba/arthas.

[9] Java Design Patterns. Retrieved September 19, 2018 from https://github.com/iluwatar/java-design-patterns.

[10] Lingming Zhang. 2018. Software Testing Verification Validation and Quality Assurance