4367 Project

1st Phase: Mid-term Report

Sagar Moozhikulam
University of Texas at Dallas
Dallas TX USA
sbm170130@utdallas.edu

Matthew Dias
University of Texas at Dallas
Dallas TX USA
mcd140030@utdallas.edu

ABSTRACT

This document will outline our project proposal and mid-term progress report for investigating mutation testing using the PIT tool.

CCS CONCEPTS

• Software verification and validation → Software defect analysis → Software testing and debugging

KEYWORDS

Mutation testing, PIT

PROJECT PROPOSAL

1 Problem

For this project the problem we want to investigate is the effectiveness of mutation testing on real-world projects. We also want to assess the original set of operators implemented for the PIT mutation testing tool, and determine what differing results may appear after augmenting the tool by adding some traditional operators that have not been implemented.

2 Techniques

The existing techniques for mutation testing consist of mutating code using defined operators that are known to be effective in producing bugs. However, not all of the traditional operators have been implemented in the PIT testing tool. For our project we will implement a few of these traditional operators. We will also select some real world GitHub projects with substantial codebases and testing suites, and run our augmented version of the PIT testing tool against them.

3 Implementation Plan

Before starting our experiment, we will need to understand how the tools we are using and modifying work. This includes the ASM bytecode engineering framework, the JAVAAgent on-fly code instrumentation, and the Maven build system. Next we will extend the PIT mutation testing tool with the following mutation operators:

- AOD: Replaces an arithmetic expression by each one of the operand
- ROR: Replaces the relational operators with each of the other ones
- AOR: Replaces an arithmetic expression by each of the other ones

After this we will run test suites from 5 real-world applications from GitHub against mutants generated by the original PIT tool, and the augmented PIT tool. Finally, we will compare the results from both runs, and compare the efficacy of each in terms of mutants killed.

4 Evaluation Subjects

We will run test suites from 5 real world GitHub projects against the mutants generated by our augmented PIT mutation testing tool. These projects are all widely used and have codebases in excess of 1000 lines and test suites in excess of 50 cases. To work with PIT they are required to all be Java projects that use JUnit for their test suites.

- RxJava - https://github.com/ReactiveX/RxJava
  This project provides functionality for event-driven programming and observables for the JVM.
- MyBatis - https://github.com/mybatis/mybatis-3
  This is a simple SQL mapper for Java objects that maps to stored procedures and SQL statements unlike traditional ORMs.
- Guava - https://github.com/google/guava
  This is a library which adds collections and other utilities for common needs to Java.
- fastjson - https://github.com/alibaba/fastjson
  This library can be used to deserialize and serialize between Java objects and JSON documents.
- Jenkins - https://github.com/jenkinsci/jenkins
  This project is an open source automation server that is mostly used for background builds and continuous integration.

MID-TERM PROGRESS

1 Idea

The first part of the project is to fork the PIT from GitHub. Since this is a copy of the code, we will have the freedom to modify however we see fit without affecting the released code on GitHub. Once we have the clone of the code, we have to add mutators to

program. These mutators that will be added define ways in which the code will be changed to attempt to generate fake detectable bugs. We will attempt to create these new mutators by copying the structure of the existing mutators that come with the PIT project. We will then modify them according to the specifications laid out in the implementation plan. After we create the mutators we will add them to the list of mutators in the project. We will then compile the augmented plugin. We will then add the augmented plugin to each of the real world projects, and run it on their test suites, comparing the results with and without our added mutators.

2 Design

The overall design of the project revolves around 3 mutation operators. The AOD mutation operator replaces an arithmetic operation by each operand. The ROR mutation operator replaces the relational operator with each of the other ones. The AOR mutation operator replaces an arithmetic expression by each of the other ones. For each category of mutators we will have a group of mutator classes. An instruction visitor will accompany each mutator. These visitors will define which instructions must be visited, and what changes will be made to them. Visitors contain either a list of the prospective changes, or a visit instruction method that will be called against all instructions.

3 Implementation

For the ROR mutators we extended the provided `AbstractJumpMutator` visitor since we were dealing with conditional statements and only wanted to mutate jump instructions. The `AbstractJumpMutator` requires a method `getMutations` that returns a `Map` with opcodes as keys, and `Substitutions` as values. Since a `Map` cannot contain the same key multiple times, we separated the mutators into different classes by the instruction that would replace the existing instruction. Each class contains all the possible conditional instructions aside from the one using as the instruction to replace the old one. For 6 different relational operators, 6 mutator classes were written, each containing 5 mutators, 1 for each of the replacement candidate instructions.

```
static {
    MUTATIONS.put(Opcodes.IF_ICMPLT, new
Substitution(Opcodes.IF_ICMPEQ, "Replaced int
less than with equal"));
    MUTATIONS.put(Opcodes.IF_ICMPGT, new
Substitution(Opcodes.IF_ICMPEQ, "Replaced int
greater than with equal"));
    MUTATIONS.put(Opcodes.IF_ICMPLE, new
Substitution(Opcodes.IF_ICMPEQ, "Replaced int
less than or equal with equal"));
    MUTATIONS.put(Opcodes.IF_ICMPGE, new
Substitution(Opcodes.IF_ICMPEQ, "Replaced int
greater than or equal with equal"));
    MUTATIONS.put(Opcodes.IF_ICMPNE, new
Substitution(Opcodes.IF_ICMPEQ, "Replaced int
not equal with equal"));
}
```

Since there are an additional set of relational operator instructions for comparing to 0, another set of 6 mutator classes was required for those instructions. Additionally, there are 2 relational operators for checking if two addresses are equal, these required an additional mutator.

For the AOR mutators a very similar implementation was required. However, since for this mutator category we want to alter plain instructions rather than conditionals, we extend the `AbstractInsnMutator` instead. This requires a `getMutations` method that returns a Map of `InsnSubstitutions` instead. These have the same format as regular `Substitutions`. Since there are 4 different primitive number types in Java with accompanying typed JVM instructions, namely, `int`, `float`, `long`, and `double`, mutators for each of these types must be created. For each of the 4 types, each mutator class must have 4 mutators to cover the 4 source instructions that will be replaced.

```
static {
    MUTATIONS.put(Opcodes.ISUB, new
InsnSubstitution(Opcodes.IADD, "Replaced int sub
with add"));
    MUTATIONS.put(Opcodes.IMUL, new
InsnSubstitution(Opcodes.IADD, "Replaced int mul
with add"));
    MUTATIONS.put(Opcodes.IDIV, new
InsnSubstitution(Opcodes.IADD, "Replaced int div
with add"));
    MUTATIONS.put(Opcodes.IREM, new
InsnSubstitution(Opcodes.IADD, "Replaced int mod
with add"));
    MUTATIONS.put(Opcodes.LSUB, new
InsnSubstitution(Opcodes.LADD, "Replaced long
sub with add"));
    MUTATIONS.put(Opcodes.LMUL, new
InsnSubstitution(Opcodes.LADD, "Replaced long
mul with add"));
    MUTATIONS.put(Opcodes.LDIV, new
InsnSubstitution(Opcodes.LADD, "Replaced long
div with add"));
    MUTATIONS.put(Opcodes.LREM, new
InsnSubstitution(Opcodes.LADD, "Replaced long
mod with add"));
...
```

For the AOD mutators a different method was required. Because some of the changes would require more than one operation we needed to directly extend the `MethodVisitor` rather than use the "static" visitors as before. For this class we were required to override a `visitInsn` method which would be called for each instruction that is visited as part of the code traversal for mutation. This method takes in an integer corresponding to the opcode for the current instruction, and in the body of the method the parent classes `visitInsn` must be called with the opcode for the replacement instruction in order to change the code. To replace the expressions with the left operand we chose to visit the instruction with the opcode for the POP instruction. This instruction removes and produces the value from the top of the stack. Longs and doubles take up two stack values since they are

larger, so for those we used POP2 instead to get the top two values from the stack.
The right operand replacement proved to be much trickier. In order to get the value of the right operand without inserting the left operand first, we had to use `SWAP` to swap the places of the top two values of the stack. After this we could perform a `POP` instruction to get the right operand.

```
mv.visitInsn(Opcodes.SWAP);
mv.visitInsn(Opcodes.POP);
```

However, this only worked for integers and floats that take up one stack value each. For longs and doubles we need a different method to swap the values. So we use `DUP2_X2`.

```
{value4, value3}, {value2, value1} → {value2,
value1}, {value4, value3}, {value2, value1}
```

As illustrated above, this instruction takes the top 4 values off the stack (which represent the top two longs or doubles), and inserts them back underneath a copy of the top 2 values. After that we can perform `POP2` twice to get the right operand.

```
mv.visitInsn(Opcodes.DUP2_X2);
mv.visitInsn(Opcodes.POP2);
mv.visitInsn(Opcodes.POP2);
```

Each mutator implemented the `MethodMutatorFactory` which requires a `create` method which returns an instance of the visitor that accompanies the mutator. After creating all the mutators, they had to be added to the PIT mutator suite by instantiating them in the mutators config. This entailed calling an `add` method with the static values for the mutators that implemented the abstract mutators for their visitors, and calling the `add` method with new instances of the mutator classes for those that implemented `MethodVisitor` for their visitors.

4 Challenges

As most of this project was unfamiliar territory, we ran into many problems along the way. Initially when integrating the augmented PIT tool with another project, we changed the version identifier of our version of the tool. For some unknown reason, maven would not detect the project with the modified name. It also caused some errors related to the subprojects of the PIT tool. We resolved this by just using the name of the project as it was when we cloned from GitHub. The reason this worked is that maven uses the local repository to resolve plugin dependencies before going to the online repository. So by installing our augmented plugin in the local repository, the project was able to pick up our plugin rather than the official one.

Since there weren't any obvious examples for creating multiple mutators based on a single source pattern, we ran into a problem implementing this functionality. At first we tried to imitate one of the default mutators that had a `makeMutators` method. This worked by taking in a property for the mutator class on instantiation which it would use to evaluate which visitors should be returned from the `create` method. The makeMutators method would then be called when adding the mutators to the suite. This would loop through all the possible source values that we wanted to consider, and would return a list of visitors each with a different source value to change. While this functionality worked (we even wrote a test for it), it didn't prove fruitful as the visit instructions were not considered when running the tool against another projects test suite. To solve this we switched to a simpler method of just creating multiple mutator classes for each replacement instruction type, and adding them all individually to the mutators configuration. While this means that there is more duplicated code and the configuration of the plugin must be longer, it actually makes the code more readable since it is closer to a declarative structure. Plus, it actually works.

The first challenge we ran into was trying to find real world projects that were suitable for evaluating our augmented plugin. As of the writing of this report, we have not yet found any that use Maven, have enough lines of code and tests, and have all passing tests. Surprisingly, many of the projects that purport to have passing builds do not actually pass successfully when running locally. I presume this is due to the way that build status is shown on many GitHub projects. Perhaps the branch is not taken into account, so while the project may be passing for whichever branch was last tested on the Continuous Integration server, it is not passing for the main branch of the repository. We will continue our search for acceptable evaluation subjects and compile our results for the presentation.

Our implementation is hosted at
https://github.com/matthewdias/pitest

5 Plan

For phase 2 of this project we are electing to choose Option 3. This seems like a very interesting application of our knowledge of mutation testing and the PIT tool. Additionally it will help us to become more familiar with how more modern testing frameworks are used. Our plan is to add functionality to PIT for the Spek testing framework. While this tool is written in Kotlin, in theory it should also work for Java projects since PIT works at the bytecode level, and since Java code can be called directly within Kotlin code.

REFERENCES

[1] GitHub: https://github.com/
[2] ASM: http://asm.ow2.org/
[3] Java Agent: https://docs.oracle.com/javase/6/docs/api/java/lang/instrument/packagesummary.html
[4] PIT: http://pitest.org/; Source code repo: https://github.com/hcoles/pitest
[5] Maven: https://maven.apache.org/
[6] Mutation testing: https://en.wikipedia.org/wiki/Mutation_testing

[7] RxJava: https://github.com/ReactiveX/RxJava
[8] MyBatis: https://github.com/mybatis/mybatis-3
[9] Guava: https://github.com/google/guava
[10] fastjson: https://github.com/alibaba/fastjson
[11] jenkins: https://github.com/jenkinsci/jenkins
[12] JVM bytecode instructions:: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings