Automated Mutation Testing using JDT and AST

Dhrupad Kaneria
Graduate Student, UT Dallas
800 W Campbell Rd
Richardson TX 75080
dhrupad.kaneria@utdallas.edu

Purva Dahake
Graduate Student, UT Dallas
800 W Campbell Rd
Richardson TX 75080
purva.dahake@utdallas.edu

ABSTRACT

This report provides an introduction to Mutation Testing, Eclipse JDT and AST. It gives an approach to write a program to generate mutants for a given project using JDT while taking the help of AST without making any changes to the actual code. One real word java program (>1000 likes of code) with JUnit tests (>50 tests) is taken into consideration and mutation testing is carried out on this project. The process will generate mutants and the mutants will be executed for the project to determine the quality and correctness of the unit test cases for the project. Depending on the number of test cases passed, the mutants are killed and the mutation score is calculated.

1. INTRODUCTION

1.1 Mutation Testing

Mutation testing [1] (or *Mutation analysis* or *Program mutation*) is used to design new software tests and evaluate the quality of existing software tests. Mutation Testing is a fault-based technique that measures the effectiveness of the test suites for fault localization, based on seeded faults. Mutation testing involves modifying a program in small ways. Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant.

1.2 Eclipse JDT

JDT [2] is a framework that provides provision to manipulate the java source code. Since it is generally attached to Eclipse, it is often called an Eclipse JDT. The JDT is broken down into components. Each component operates like a project unto its own.

1.3 JUnit Tests

JUnit [3] is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit. A research survey across 10,000 java projects hosted on GitHub found that JUnit was most commonly used. It is was used by 30.7% of the projects.

1.4 Abstract Syntax Tree

In computer science, an abstract syntax tree (AST) [7] or just syntax tree, is a tree representation of the abstract syntactic structure or source code written in a programming language. Each node denotes a construct occurring the source code. A class called ASTVisitor [8] provided by AST is used to visit all the nodes in the tree. This helps to visit the given node to perform some arbitrary operation. In this project the arbitrary operation is changing the statement and saving the changes in the form of a mutant.

1.5 Apache Maven

Apache Maven [9] is a build automation tools used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. An XML file (pom file) describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins.

2. PROBLEM

Tests are created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. In order to answer this problem, we tend to undergo the verification of test cases. This verification is done by Mutation Testing.

Each test case is executed on all the mutants that are generated. The outcome of the execution (number of mutants that are still alive) will determine more about the quality and correctness of the test cases written.

3. BASICS OF EXISTING TECHNIQUE

The important part of mutation testing is to induce defects by making small changes in the current program. This is mostly done by changing the operands, or by changing the operands. Assuming the average size of the project is 1000 lines, changing all the operators and then running the program manually is practically impossible. Hence there are certain tools available in the market that generates mutants and gives the result. The widely used mutation testing tools are described and introduced in the below sections.

3.1 Pit

Pit [4] is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It is fast, scalable and integrates with modern test and build tooling. Pit runs the unit tests against automatically modified versions of the application code. Pit is actively developed and supported. As a result, it is one of the widely used tool for mutation testing.

3.2 µJava

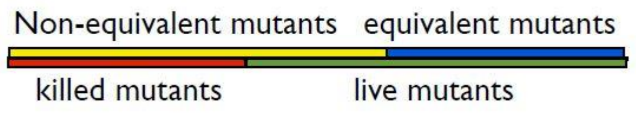
µJava [6] is a mutation system for Java programs. µJava uses two types of mutation operators, class level and method level. It creates object-oriented mutants for Java classes according to 24 operators that are specialized to object-oriented faults. Method level (traditional) mutants are based on the selective operator set. After creating mutants, µJava allows the tester to enter and run tests, and evaluates the mutation coverage of the tests.

4. ANALYSIS PLAN

We intend to analyze a real world Java project with more than 1000 lines of code and more than 50 JUnit tests.

4.1 Mutation Score

The mutants are divided into equivalent and non-equivalent mutants or live and killed mutants. The relation between the four types of mutants is given by the following diagram.



The relation between these mutants is given by the mutation score. The mutation score [5] is defined as the percentage of killed mutants with the total number of mutants.

$$MS(T) = \frac{\#KilledMutants}{\#AllMutants - \#EquivalentMutants}$$

Since it is very difficult to find the number of equivalent mutants, we have made a silent change in the formula and is given by:

$$MS\ (T) = \frac{\#Killed\ Mutants}{\#\ All\ Mutants}$$

Though this is not the exact estimation, this helps in providing a decent knowledge about the quality of the test cases written.

4.2 Mutation Strategy

The mutants can be generated based on different changes. The original program code is changed only at one line. If more than one change is induced, it is called Higher Order Mutation Testing. After going through many articles and papers on mutation testing, we have come up with the following strategies.

Old Operator	New Operator
<	<=
<=	<
>	>=
>=	>
!=	=
=	!=
||	&&
&&	||
*	-
-	*

The mutation strategy is not implemented on all the statements in the project. We have targeted only the expressions that in Infix format. Expressions in postfix and prefix format is not taken into consideration. Specifically the mutation strategy is applied to Infix expressions in the expressions of **If** statements and **While** statements only.

5. EXPERIMENTAL EVALUATION SUBJECTS

The experimental subject on which the mutation testing is being tested out is selected form the list of the projects provided. The project is called parse4j. It consists of 92 test cases and over 1000 lines of code spread across different packages and files. Out of the 92 test cases, 9 test cases had failed the execution and 11 test cases are errored. After understanding the implementation of the test cases and scenario being tested, we learnt that these are the test cases for testing the negative impact i.e. these test cases were written to test the exception handling for invalid test inputs.

6. PSEUDO CODE

This section will have the pseudo code and the algorithm used in implementing the project. The entire project is divided into mainly three different stages namely generating mutant, running the test cases and calculating mutation score.

6.1 Approach

After selecting the experimental specimen (the master project – parse4j), it was examined for the given specifications. The test cases were executed for the given project using maven. The observations returned were noted down for future analysis.

Stage 1:

After completing the initial analysis and execution, the java code to generated mutants was implemented. In this code, the copies of actual project is made for the mutants. In each copy, the changes are made and mutants are generated. By using ASTVisitor class provided by the AST the ifStatement and whileStatements are visited and their expressions are changed. From the criteria discussed previously, the operators are changed and saved. These changes are also stored and verified against before making the change. If the change is already applied, the next line is changed.

Stage 2:

Once all the mutants are generated, the next step was to execute the JUnit test cases on these mutants. This execution is done using maven. Since the master project is a maven project, the unit tests should be executed using maven itself. There are a few changes inserted in the pom file to support faster execution of the test cases. The changes include multithreading execution of test cases on both method level and class level. This has helped in faster execution of the test suite.

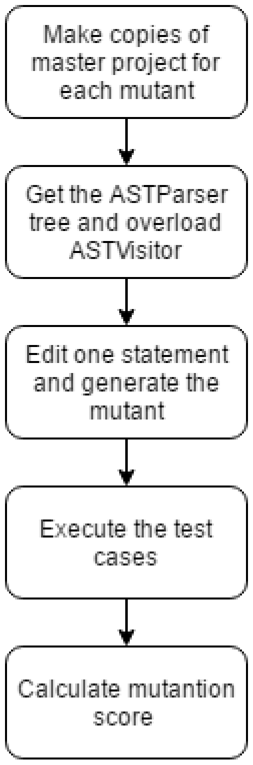
Stage 3:

After the test cases are executed on the mutants, another java program is executed to check the killed mutants and live mutants and thereby calculate the mutation score. The output generated by executing the test cases on the mutants is read through and compared with the number of passed test cases, failed test cases and error test cases. If there is a mismatch in the numbers, the mutants are killed and the count is increased. Once all the mutants undergo the check, the mutation score is calculated based on the formula given previously.

Stage 4:

This entire process of the above explained three stages are automated with batch file. On executing the batch file using command prompt, the user should input the number of mutants to be created. Once the count is entered, the java program to create the mutants is executed. On successfully creating the mutants, the test cases are executed on these mutants using maven. The mutation score is then calculated with the help of the java program.

6.2 Flowchart


Make copies of master project for each mutant
Get the ASTParser tree and overload ASTVisitor
Edit one statement and generate the mutant
Execute the test cases
Calculate mutantion score

6.3 Code Sample

Stage 1:

```
Generate copies of the master project for each mutant
For each mutant
{
Randomly find a file to change
Generate the AST parse tree
For the compilation unit
Unit.accept(new ASTVisitor())
{
//Override visit method
Boolean visit(final IfStatement statement)
{
Get the expression in the statement
If the mutant is not already created, get the mutation strategy and make the change
Save the file if the same change is not already done
}
Boolean visit(final WhileStatement statement)
{
Do the same things as mentioned above
}
}
}
```

Stage 2:

```
Execute the test cases on the mutants
Mvn test
```

Stage 3:

```
For all the mutants
{
Go through the target files and get the count of the failed tests and errored test cases
}
If the count doesn't match the count from the master project
{
Kill the mutant and update the count
}
Calculate the mutation score
```

Stage 4:

```
This is the batch file to automate the previous stages
Enter the number of mutants to be generated
Compile the first java code
Execute the code to generate the mutants
In a loop for all mutants
{
        Go to the respective directory
        Execute mvn test //to execute the test cases on the mutant
}
Compile the java code to calculate mutation score
Execute and calculate the mutation score
```

7. OBERVATIONS

For example observation, we consider creating _ number of mutants. This can be specified when the batch file is run. The batch file contains the automated code to compile the java files and run JUnit tests on the mutants. The only input the user is required to give is the number of mutants he/she wishes to create for performing mutation testing.

For _ number of mutants, our testing framework gives the following observations which are stored in an output.txt file. It logs the test statistics namely, the number of JUnit tests, how many of them failed, how many of them threw errors and how many of them passed. Then we applied the software testing concept of mutation testing and inferred how many of them were actually killed. If the mutant is killed it contributes in increasing the mutation score. Higher the mutation score, greater is the quality of the testing framework.

Following is one of our observed samples:

```
Enter num of mutants: 4
```

```
Master Project
Test Run : 92
Failed : 9
Error : 11
Skipped : 0

For Mutant 0
Test Run : 92
Failed : 14
Error : 14
Skipped : 0
Mutant 0 is killed

For Mutant 1
Test Run : 92
Failed : 11
Error : 11
Skipped : 0
Mutant 1 is killed

For Mutant 2
Test Run : 92
Failed : 9
Error : 11
Skipped : 0
Mutant 2 is alive

For Mutant 3
Test Run : 92
Failed : 9
Error : 11
Skipped : 0
Mutant 3 is alive

Final Statistics : Killed Mutants : 0
Alive Mutants : 2
Total Mutants : 2
Mutation Score : 50.0%
```

8. CONCLUSION

We have thus achieved the project aim satisfying all the requirements and criteria. Developing an automated software testing framework was indeed a challenge but very interesting at the same time. We learnt how the basic concepts of software testing can be applied on real world projects.

Mutation testing seems powerful and some research indicates that mutation score is a better predictor of real fault detection rate than code coverage. As one might guess, creating mutations and executing tests against those mutations is not a lightweight process and can take quite a lot of time.

Automating the mutation testing process has greatly increased the scope to perform much more complex testing and has opened a new window for software testing, verification and validation.

9. REFERENCES

[1] Mutation Testing, https://en.wikipedia.org/wiki/Mutation_testing

[2] Eclipse JDT, https://eclipse.org/jdt/

[3] JUnit, https://en.wikipedia.org/wiki/JUnit

[4] PIT, http://pitest.org/

[5] Mutation Score, http://www.guru99.com/mutation-testing.html

[6] µJava, http://cs.gmu.edu/~offutt/mujava/

[7] Abstract Syntax Tree, https://en.wikipedia.org/wiki/Abstract_syntax_tree

[8] ASTVisitor, http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html

[9] Apache Maven, https://en.wikipedia.org/wiki/Apache_Maven