

Mutation Testing of Source Code

Software Testing[CS - 731]

Koustubh Tadanki

IMT2018048

Nipun Goel

IMT2018052

Omkar Lavangad

IMT2018053

1. Abstract

This report consists of the experiments and their descriptions for Mutation Testing on a Codebase. We have used a codebase written in Java and PITclipse for performing and analysis of mutation testing.

2. Introduction

Many software fails due to a lack of testing various aspects of the program. Software Testing is a very crucial phase for Software Development. Software testing is important as it verifies the customer's reliability and their content with the application. In this project, we focus on Mutation Testing of source code is a white box technique.

Mutation testing, also known as code mutation testing, is a form of white box testing in which testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes. Changes introduced to the software are intended to cause errors in the program. Each mutated version is called a mutant and it tests, detects and rejects the mutants by causing the behaviour of the mutated version to differ from the original. This is called killing the mutant.

By the percentage of mutants that they kill, test suites are measured. Mutants are based on well-defined mutation operators that either mimic typical programming errors (such as using the variable name or wrong operator) or force the creation of valuable tests (such as dividing each expression by zero). Thus helping the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom accessed during execution.

We developed a suitable code base and tried mutation testing on it. And also analysed the results from the test.

2.1 CodeBase

The Codebase chosen for this experiment was 1000+ lines of code with class `ArrayFunctions` which has a lot of basic methods that are very commonly and widely used on arrays. These include 10 sorting methods. `ArrayFunctions.java` is the source code for which different mutation operators are used and `ArrayFunctionsTest.java` is the JUnit test case java class. We also generated random tests by writing a script `generateArray.cpp`.

2.2 Steps to do mutation testing :

1. We introduce a few faults into the source code by creating many versions called mutants. And then each mutant is obtained by using one of the compatible mutation operators in source code.
2. Now as we have introduced mutants, they should be killed by introducing some test cases. And the test cases should pass for the original program and strongly kill the mutant (i.e. propagating the error which results in incorrect output).
3. Mutant is said to be strongly killed if the output of the source program and mutated program differ from each other in a particular test case. Otherwise, it is said to have survived.

2.3 Mutation Testing tool

We explored tools for mutation testing for java and identified PIT testing as the fastest and easiest to use. Further in this report, we will explain how it was used for mutation inferences.

About PIT

PIT is one of the most popular mutation testing tools for Java. According to the PIT official website. PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the JVM. It's fast, scalable and integrates with modern test and build tooling. For mutation coverage, Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lives. The quality of your test cases can be gauged from the percentage of mutations killed.

By default the operators belonging to `DEFAULT` are active. However, there isn't much support to actually select the mutation operators though the documentation provides a few complex workarounds. Hence for the current task, we chose the *DEFAULTS* group.

For Example, `Negate Conditionals Mutator` is a class of mutator operations where the conditional statements are mutated in the following method :

The negate conditionals mutator will mutate all conditionals found according to the replacement table below.

Original conditional	Mutated conditional
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

3. Inferences and Results

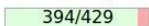
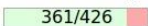
Following are the mutation operators which PIT applied on our codebase:

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Initially, we obtained a mutation coverage of **85%**

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
1	92%  394/429	85%  361/426

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
com.softwareTesting.ArrayFunctions	1	92%  394/429	85%  361/426

Report generated by [PIT](#) 1.4.11

We tried to improve this mutation coverage by analysing the mutants that survived and then designed test cases to improve the mutation coverage percentage. We also randomised some of the test case generations to improve the coverage.

After doing so, we obtained mutation coverage of **90%**.

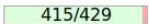
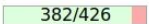
Final Results :

```
=====
- Statistics
=====
>> Generated 426 mutations Killed 382 (90%)
>> Ran 429 tests (1.01 tests per mutation)
```

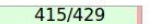
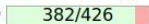
Pit Test Coverage Report

Package Summary

com.softwareTesting.ArrayFunctions

Number of Classes	Line Coverage	Mutation Coverage
1	97%  415/429	90%  382/426

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ArrayFunctions.java	97%  415/429	90%  382/426

Report generated by [PIT](#) 1.4.11

```
-----
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalBoundaryMutator
>> Generated 88 Killed 62 (70%)
> KILLED 62 SURVIVED 25 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
-----
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 69 Killed 68 (99%)
> KILLED 67 SURVIVED 0 TIMED_OUT 1 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
-----
```

Most of the surviving mutants belonged to the Conditional Boundary Mutator family.

On further analysis, we realised that many of the surviving mutants are equivalent mutants. So, no test case could kill them.

```
//get Maximum of Elements of an Array
static int getMax(int arr[])
{
    int n = arr.length;
    int res = arr[0];
3   for (int i = 1; i < n; i++)
2   {
        res=arr[i];
    }
1   return res;
}
```

1. getMax : changed conditional boundary → SURVIVED
2. getMax : negated conditional → KILLED

```
//get Maximum of Elements of an Array
static int getMax(int arr[])
{
    int n = arr.length;
    int res = arr[0];
3   for (int i = 1; i < n; i++)
    {
2       if(res<arr[i])
        {
            res=arr[i];
        }
    }
1   return res;
}
```

In the above screenshot, the function `getMax(int arr[])` returns the max element of the array and so replacing "`res<arr[i]`" by "`res<=arr[i]`" won't affect the state of the program at any time.

4. Contributions

All of us installed eclipse and PITclipse (eclipse plugin for PITest) in our individual laptops. Due to a lack of a suitable existing codebase, we decided to write a new java code suitable for mutation testing. A suitable code should have lots of functions with arithmetic operations and inputs and outputs, so we decided to write a java library class for various functions which are commonly and widely used on arrays. We divided the functions among us and merged our individual codes to create a java file with 1000+ lines of code.

The test suite was basically divided into 3 parts:

- Analysis and TestCases for the Sorting methods (Nipun).
- Analysis and TestCases for primitive return type methods (Omkar).
- Analysis and TestCases for the remaining methods (Koustubh).

We also wrote a script for generating random test cases and corresponding expected output.

5. References

- "PIT Testing Home". URL: <http://pitest.org/>.
- Bouchaib Falah and Soukaina Hamimoune. "Mutation Testing Techniques: A Comparative Study". In: Nov. 2016. doi: 10.1109/ICEMIS.2016.7745368.
- PITest medium article
<https://medium.com/geekculture/mutation-testing-for-maven-project-using-pitest-f9b8fef03a05>
- What is Mutation Testing? <https://www.youtube.com/watch?v=wZeZMtqVmck&t=533s>
- PITclipse GitHub repo <https://github.com/pitest/pitclipse>
- Junit asserts documentation <https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html>