Automated Mutation Testing Framework

Jay Sharma

The University of Texas at Dallas  
7815 Mccallum Blvd, #1201  
Dallas, Texas, 75252  
(469)213-9126

jms140730@utdallas.edu

Neel Parikh

The University of Texas at Dallas  
7740 Mccallum Blvd, #308  
Dallas, TX, 75252  
(469)831-7407

nvp140230@utdallas.edu

**ABSTRACT**

In this paper, we are proposing an automated mutation testing framework for Java using source code modification technique. This tool will use the JDT library of eclipse for generation of mutants from the existing open source GitHub project and test these mutants against the Junit tests to produce the mutation report. This report will reflect the quality of a test suite. This tool will be extremely useful in real world as manually creating mutants and test each of those is cumbersome and costly.

**Keywords**

Mutants; Mutation Testing; Automated Mutation Testing; MuClipse; Jester; Junit; JDT; Eclipse;

# INTRODUCTION

# Mutation Testing is a fault-based testing technique which provides a testing criterion called the “mutation adequacy score”. The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults. Mutation Testing can be used for testing software at the unit level, the integration level and the specification level. It has been applied to many programming languages as a white box unit test technique [1].

Manual Mutation Testing is time consuming and gets complicated as well. To accelerate this process, we will be developing an automated mutation testing framework which will reduce the time and cost for testing a project.

# PROBLEM ANALYSIS

Mutation testing is an important part of white box testing as it helps to test the quality of test suites by creating artificial bugs called as Mutants inside the code. However, creating mutants and testing them on a project manually is time consuming and cost ineffective. Additionally, while manually creating mutants, there is no surety that all possible mutants would be covered.

Therefore, we propose an automated mutation testing framework that will robotically create mutants for a project and test them against the test cases in suites for the project. At the end of the task, a report will be generated. This report will display the mutation score that will help users to assess the quality of the test cases in suites.

Figure 1 shows the architecture of the mutation testing tool. A Software project is provided as an input to the tool. The Mutant generator uses the available Java mutation operators to generate *Mutants* for the project. These mutants are applied on the test suites which generates a mutation report. This report includes the ratio of mutants killed to total number of mutants also called as Mutation score. This score can be used to determine the quality of the test suites. Higher the mutation score, better the quality of test suites.

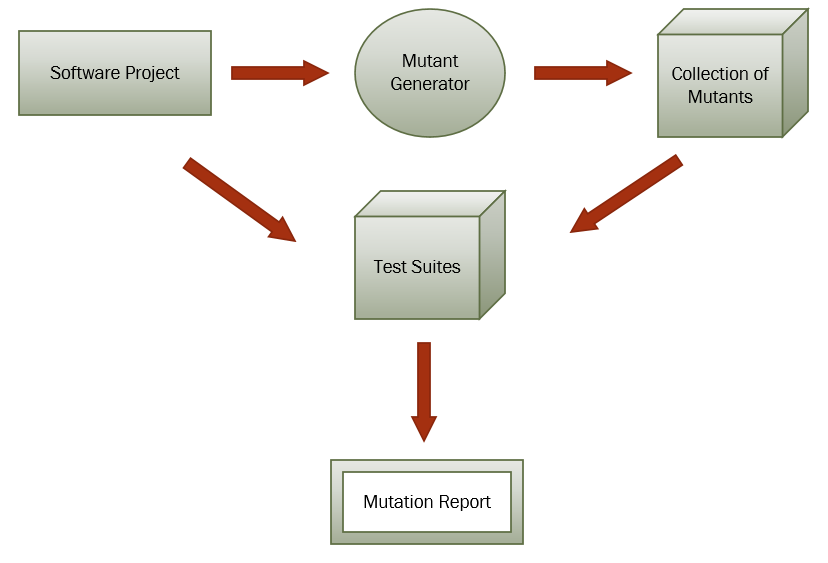


Figure : Tool Design

# BASICS OF EXISTING TECHNIQUES

There are various existing mutation testing tools available for Java that operates on source code level. The two most widely used are MuClipse and Jester.

*MuClipse* [2] is an eclipse based plugin for mutant generation and for running JUnit test suites. It is based on MuJava. MuClipse supports 15 traditional operators and 28 class level mutation operators. It provides GUI, mutant viewer and test case executor. GUI support helps in selecting the mutation operator and generating the mutants. Mutant viewer is used to view the mutants and test case executor for execution of mutants using test set. MuClipse automatically generates mutants by selecting the class to mutate and run JUnit test suites on each mutant. It accounts the total number of live as well as killed mutants along with the mutation score for that test suite. For mutant creation, it first generates the parse tree of the subject program and then generates the mutants. Mutation operates on the source code, and thus, it is necessary to compile the mutants before running JUnit test suite. This process makes it quite slow. It also ignores the equivalent mutants [4].

*Jester* [3] is the JUnit test tester. It modifies the source code as well as the test code and verifies whether the change is detected by the test. The test is erroneous if it is not modified and also does not fail. It offers a strategy to extend the traditional set of mutation operators. It repeats the process of creating, executing, testing and reporting for each mutant during analysis; thus, very slow to work with. The cost of using ‘Jester’ is very high in terms of time that includes machine time and developer time to interpret the result. It also doesn’t provide the interface to select the mutation operators applied for the programs under test [4].

# IMPLEMENTATION/STUDY PLAN

The tool will be developed using Java programming language in Eclipse IDE. JDT Library will be used to manipulate the source code to generate mutants. GitHub will be used as a version control system. The generated mutants will be tested on Junit tests of the project, based on which a mutation report will be produced. This report will display the mutation score that will help to determine the quality of a test suite.

The tentative development plan for the tool is as follows:

Table 1: Implementation Plan

|  |  |  |
| --- | --- | --- |
| **Task** | **Start Date** | **End Date** |
| Project Proposal | 09/24/2016 | 10/01/2016 |
| Study Eclipse JDT Framework | 10/02/2016 | 10/08/2016 |
| Develop prototype | 10/09/2016 | 10/15/2016 |
| Implementation of visitor classes | 10/16/2016 | 10/21/2016 |
| Midterm progress report | 11/28/2016 | 10/30/2016 |
| Implement logic for mutant generator | 11/01/2016 | 11/10/2016 |
| Test Mutant Generator | 11/11/2016 | 11/16/2016 |
| Generate Execution report | 11/16/2016 | 11/25/2016 |
| Review and Update code | 11/26/2016 | 12/01/2016 |
| Final Project Report | 12/02/2016 | 12/07/2016 |

# DESIGN OF OUR TECHNIQUE

We have implemented Visitor Object Oriented design pattern as the architecture of the project. We have chosen to use this pattern ahead of other design patterns because it provides separation of algorithm from an object structure it operates on. Thus it provides an ability to add new operations (Operator Replacement classes in this case) to existing object structures without modifying those structures. This pattern will generate the set of mutants based on the operator replacement classes and will use these mutants to generate mutation score based on the test cases.

We will be focusing on the following Java based operators for mutant generation [5]:

* Logical Operator Replacement (LOR)
* Relational Operator Replacement (ROR)
* Arithmetic Operator Replacement (AOR)
* Shift Operator Replacement (SOR)
* Assignment Operator Replacement (ASR)

The mutation strategy for each of the above operators is explain as below:

1. **Logical Operator Replacement (LOR)**

Replace binary logical operators with other binary logical operators.

The mutants will be generated by using the logical operators like AND, OR, and XOR.

Table 2: Logical Operator Replacement

|  |
| --- |
| Operators |
| AND |
| OR |
| XOR |

1. **Relational Operator Replacement (ROR)**

Replace relational operators with other relational operators, and replace the entire predicate with true and false.

The mutants will be generated by using the relational operators like:

Table 3: Relational Operator Replacement

|  |
| --- |
| Operators |
| == |
| != |
| > |
| >= |
| < |
| <= |

1. **Arithmetic Operator Replacement (AOR)**

Replace short-cut arithmetic operators with other unary arithmetic operators.

The mutants will be generated by using the arithmetic operators like:

Table 4: Arithmetic Operator Replacement

|  |
| --- |
| Operators |
| + |
| - |
| \* |
| / |
| % |

1. **Shift Operator Replacement (SOR)**

Shift Operator Replacement Replace shift operators with other shift operators.

The mutants will be generated by using the shift operators like:

Table 5: Shift Operator Replacement

|  |
| --- |
| Operators |
| >> |
| << |

1. **Assignment Operator Replacement (ASR)**

Replace short-cut assignment operators with other short-cut operators of the same kind.

The mutants will be generated by using the assignment operators like:

Table 6: Assignment Operator Replacement

|  |
| --- |
| Operators |
| += |
| -= |
| \*= |
| /= |
| %= |

# DESIGN AND EXPERIMENTAL EVALUATION

The mutants are generated from an open source GitHub project having more than 1000 lines of code. The tool will produce a quantitative result on the quality of the test suite for that project.

For the purpose of testing our mutation tool we have used “Cucumber-reporting” project.

<https://github.com/damianszczepanik/cucumber-reporting>

The process we followed for development of this tool is as follow:

1. Created a GIT Repository for creating a collaborative working environment

<https://github.com/sharmajaym2007/MutationTestingFramework>

1. Analyzed the Eclipse JDT Framework to go through the various set of classes which could be used for development of our tool
2. Analyzed best practices for building an architecture for visitor design pattern
3. Implemented visitor design pattern with various visitor classes for the mutation operators as specified in the previous sections

The structure of the project is a hierarchical view of the java files which inherits the AST Visitor classes from the JDT framework.

The project structure currently looks like below:

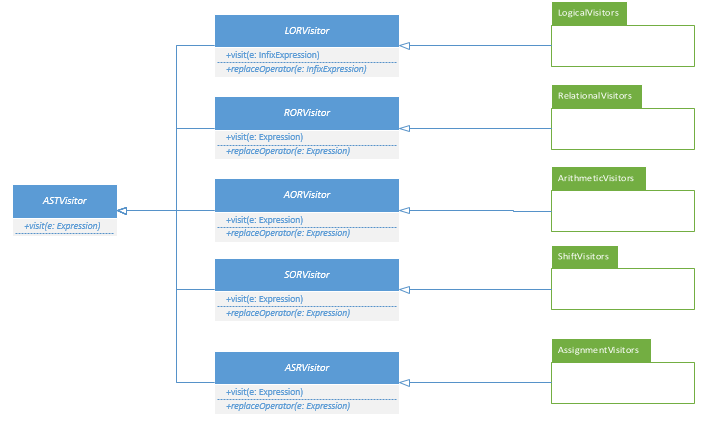


Figure : Visitor Design Classes

The visitor classes are contained in their respective packages as below:

* LORVisitor:
* Contains all the logical operators replacement java classes
* *visit(InfixExpression exp)*
* RORVisitor:
* Contains all the relational operators replacement java classes
* *visit(InfixExpression exp)*
* SORVisitor:
* Contains all the shift operators replacement java classes
* *visit(InfixExpression exp)*
* AORVisitor:
* Contains all the arithmetic operators replacement java classes
* *visit(InfixExpression exp)*

1. Developed a driver program to generate the mutants by manipulating the source code for the existing project.
2. Create Python script to run the jar file for the generator tool to create a dynamic testing environment to run on any project under consideration.
3. At last, we tested the quality of our tool by running it against various other GitHub projects that has more than 1000 LOC

# ALGORITHM DESIGN

We have implemented the code such that all the possible mutant operators will be covered wherever applicable. Initially we had an issue with less number of mutants generated due to some conditional issue in the logic which we eventually fixed it in the final version of the tool.

The basic algorithm for mutant generation and execution works as below:

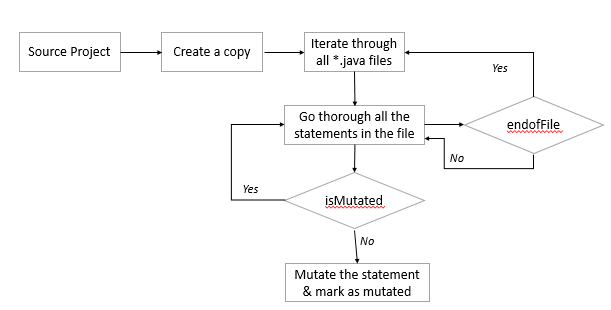


Figure : Mutant Generation

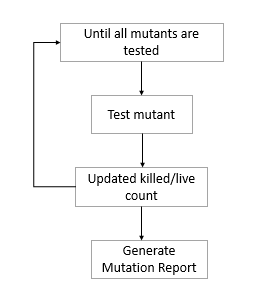


Figure : Mutant Execution

* Scan through the source code files
* Search for a statement which can be mutated
* Using the visitor classes, apply all the possible mutation operators
* Create a copy of the original source code for all the mutants generated
* Create a data structure to store the mutation operator, line number and file name information
* The above step will reduce any redundant mutants
* Once all the mutants are generated, run the mutants against the test suite
* Create the mutation report to test the effectiveness of the test suite

# EXECUTION STEPS

Our tool is developed as a standalone executable jar file along with a Python script to automate the pre and post execution steps such as directory clean-up and loading of source code from the path specified. The tool will run the maven tests for the existing Junit tests for the project against all the mutants and at the end will provide a mutation report in the end with all statistics of live mutants and killed mutants. The basic steps to run the tool is:



Figure : Execution flow

**Pre-requisites:**

* Must have Java8 and Python 3 on your system

**Steps:**

1. Place the ***mutation.jar*** file and ***testscript.py*** file in some directory on your machine
2. Inside command prompt, run the python script as:

***testscript.py***

1. Provide the project path
2. Then the python script will start the execution of generating the mutants in the same directory of your project directory with name ***“mutants”***
3. The script will execute all the mutants and create a mutation report which you can see on the command prompt as well as in the ***report.txt*** file

# TOOLS AND TECHNOLOGIES

For this project to be successful we have utilized various tools and technologies as below:

**Java** – Implementation of mutant generation

**Maven** – Execution of Junit tests

**Log4j** – Logging framework for the tool which is useful in case of debugging

**Python** – Automation script for execution of test suite against each mutant and provide a mutation report in the end

**Eclipse** - IDE for Java development

**GitHub** - Collaboration tool for Version Control

# CHALLENGES

The idea of mutation testing is pretty interesting but at the same time, it is also challenging to perform on all the projects especially when your project is very complex. There were few challenges that we faced during the course of project development such as:

* **Handling the ‘+’ operator replacement** – As the ‘+’ operator is used for concatenation of the string along with the arithmetic operations, we had to consider this case in our code. We did this using the JDT class ***InfixExpression*** which allows the identify the left and right operand of a statement, so when both the operands were strings we ignored the replacement
* **Time Consuming** – This is a known limitation of mutation testing that it takes a long time to execute. In order to reduce that, we used the selective mutation and random mutation strategy which aims to run mutation testing only of selected operators. This reduced the running time but not as effective as it was expected.

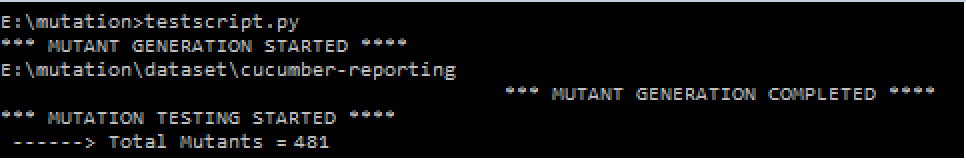
# EXPERIMENTAL RESULTS

We used the existing GitHub project to run our tool and carry out the experimental results. Apart from this, we also tried to run the code with couple of other projects from GitHub to compare the results and check the correctness of our tool.

Below are few screenshots of our results:

1. **Mutant Generation**

The screenshot shows the mutant as a project copy with a change in the file for the mutant. These mutants are created using the ***GenerateMutants.java*** file inside the ***mutants.jar*** executable file.



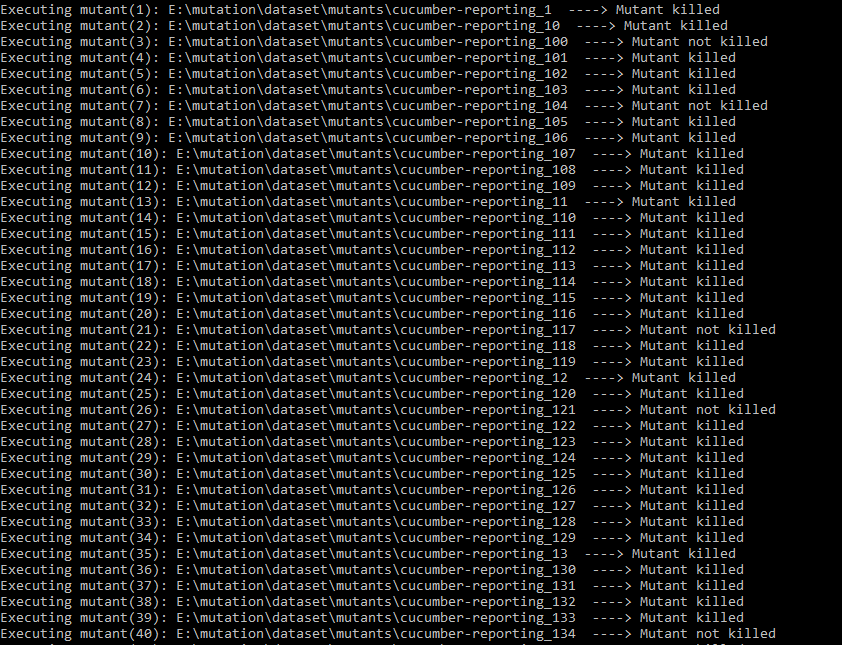


Figure : Mutant Generation Screenshot

1. **Log4j logging**

This screenshot shows the logs generated for the code by the log4j framework for mutant generation and execution.



Figure : Log4j Screenshot

1. **Test failure by Maven**

The below screenshot shows the test which is failed by maven for the mutant project when “mvn test” command is executed by the python script in order to run the Junit test cases.

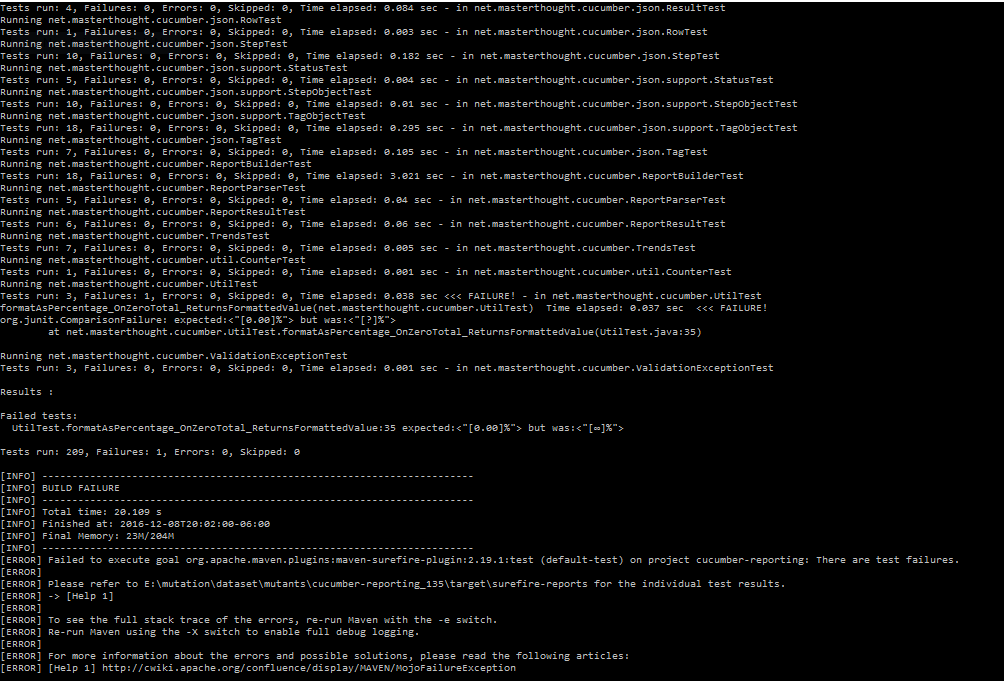


Figure : Maven Screenshot

1. **Mutation Report**

The final screenshot shows the mutation report that will be generated by the Python script once all the mutants are executed against the test suite. This indicates the number of live and killed mutants in the project.

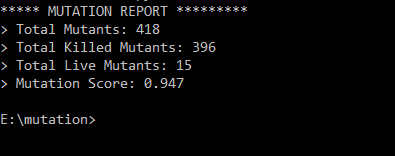


Figure : Mutation Report

1. **Comparison graph**

We ran the tool, against 2 other projects for the correctness and efficiency of our tool. The graph below shows the result of all the 3 projects which indicates that more the number of mutants, better the chances of finding the effectiveness of a test suite.

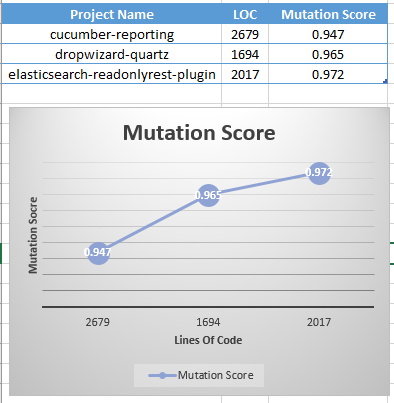


Figure : Comparison Chart

# FUTURE SCOPE

For this project, we have implemented the basic working functionality of mutant generation and execution using the Operator based mutation testing. There are few updates that we are planning to do in future:

* Adding more set of mutant operators to prove that our tool is efficient
* Use the existing mutation testing tools to compare the results with our tool and perform necessary improvements in case of adding operators, refining code logic in order to improve performance
* Open source the tool to GitHub community of users to get ideas from various developers all over the world interested in mutation testing which can marginally improve the quality of the tool in the near future
* Currently, the tool is making a copy of the project inserting the mutated statement in one of the file, this seems to be less effective as a degrades the performance
* In order to improve the performance, we plan to refine our code such that, only the mutated file is replaced in the original project without making a copy of the project

# REFERENCES

1. Yue Jia and Mark Harman, “An Analysis and Survey of the Development of Mutation Testing”, IEEE Transactions on Software Engineering.
2. B. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION,'07, 2007, pp. 193-202.
3. I. Moore, "Jester-a Junit test tester," in 2nd International Conference on Extreme Programming and Flexible Processed in Software Engineering, Italy, 2001.
4. Shweta Rani, Bharti Suri and Sunil Kumar Khatri, “Experimental Comparison of Automated Mutation Testing Tools for Java”.

[5] Yu-Seung Ma, “Description of muJava’s Method-level Mutation Operators”.