Automated Mutation Testing Framework Project Proposal

Zengtai Qi

The University of Texas at Dallas  
800 W. Campbell Road,   
Richardson, Texas 75080-3021  
+1 (469) 642-3217

Zengtai.Qi@utdallas.edu

**ABSTRACT**

In this paper, I describe the specifications of the project which is an automated mutation testing framework. The project is the final project of the course SE6367.002 in the University of Texas at Dallas, and the instructor Professor Lingming Zhang.

This paper will include following parts: the problem I want to investigate, the basics of existing techniques, my implementation plan.

**CCS Concepts**

**Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging**

**Keywords**

Mutation testing; weak mutation; strong mutation; JUnit.

# INTRODUCTION

Mutation testing was originally proposed by Richard Lipton as a student in 1971 [1], and first developed and published by DeMillo, Lipton and Sayward [2]. The first implementation of a mutation testing tool was by Timothy Budd as part of his PhD work (titled *Mutation Analysis*) in 1980 from Yale University [3].

Mutation analysis is a well-known approach to assess the quality of test suites or testing techniques [4]. Mutation testing is based on two hypotheses. The first is the *competent programmer* hypothesis. This hypothesis states that most software faults introduced by experienced programmers are due to small syntactic errors [2]. The second hypothesis is called the *coupling effect*. The coupling effect asserts that simple faults can cascade or *couple* to form other emergent faults [5].

# PROBLEM I WANT TO INVESTIGATE

This project is aimed at Java programming language. I plan to implement an automated mutation testing framework, using ASM byte-code manipulation framework.

Then I plan to apply my tool to 1 real-world Java project with JUnit tests to generate mutants and execute mutants for the project. I may also use Java Agent and integrate my tool with the Maven build system.

The purpose of the project is understanding what is mutation testing, why mutation testing does work and how the mutation testing works. Through this project, I plan to learn how to evaluate test suites and then evaluate the effectiveness of a set of test suites.

In addition, I plan to investigate the different effectiveness of different mutation generating methods (mutation engine).

# BASIC OF EXISTING TECHNIQUES

## What is Mutating Testing

Mutation testing [2] is a test criterion that can be used to design test sets or evaluate test sets by generating a set of alternate programs, or *mutants*, and determining how many of these mutants are detected by the test suite. Mutants are typically generated by mutation operators, but the alternate programs are not limited to such approaches.

## Does Mutating Testing Work

In some circumstance, test suite does cover all statements, branches and paths. And the test suite reaches 100% for all control-flow and data-flow coverage. But this doesn’t mean the test suite is perfect and the program is bug-free. The mutation testing can supply an approach to test the test suite, evaluate test suite and measure the effectiveness of the test suite.

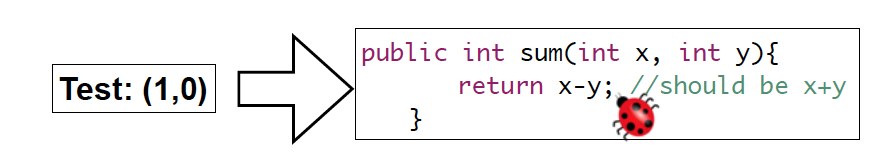


Figure 1. A test suite example.

.

In the above test suite example (see Figure 1), the test suite contains one test case which is (x = 1, y = 0). This test suite does achieve 100% for control-flow and data-flow, but it didn't successfully catch the bug.

Now I mutate the function ‘sum()’ by modifying ‘x - y’ to ‘x \* y’, the test suite will return an opposite result with this program. In this way, we successfully evaluated the test suite with mutating testing.

## How Mutating Testing Works

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be *killed*.

There are many kinds of mutation operations such as mutation operators for imperative languages, mutation operators for object-oriented languages [6], mutation operators for concurrent constructions [7], mutation operators for complex objects like containers [8], etc.

A mutation operator is a rule specifying syntactic changes. The collection of mutation operators is a crucial factor in the effectiveness of mutation testing. The most common mutation operators replace each operand with other syntactically suitable operands. The mutation engine generates a number of variants of the original program called the mutant programs.

Here are some examples of mutation operations for Java programming language in the table 1.

Table 1. Java mutation operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| AOR | Arithmetic Operator Replacement |
| AOI | Arithmetic Operator Insertion |
| AOD | Arithmetic Operator Deletion |
| ROR | Relational Operator Replacement |
| COR | Conditional Operator Replacement |
| COI | Conditional Operator Insertion |
| COD | Conditional Operator Deletion |
| SOR | Shift Operator Replacement |
| LOR | Logical Operator Replacement |
| LOI | Logical Operator Insertion |
| LOD | Logical Operator Deletion |
| ASR | Assignment Operator Replacement |

After mutation testing is done, Mutation Testing Criteria (aka mutation score) is used to measured the effectiveness of test suites. Higher mutation score means the test suite is more effective. The mutation score formulation is:

MS(T) = #KilledMutants / (#AllMutants + #EquivalentMutants)

# IMPLEMENTATION PLAN

## Generate Mutation

As I mentioned in the section 3.3, The collection of mutation operators is a crucial factor in the effectiveness of mutation testing. Mutation generation is the most important part of this project.

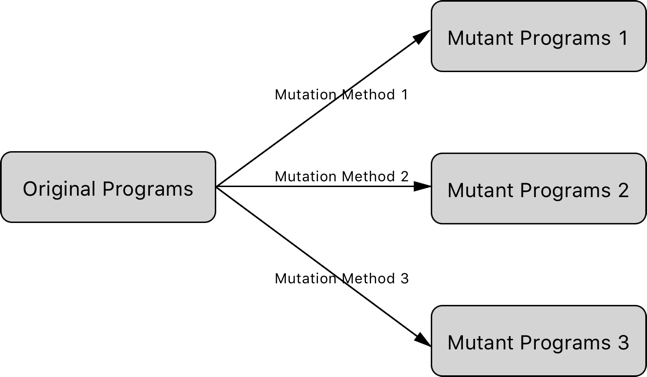


Figure 2. Generate mutations.

.

Since investing different mutation generation methods’ different effectiveness is one of purposes of this project, I plan to use many mutation generation methods (mutation engine) to generate many sets of mutant programs. These sets of mutant programs will be all including in the following investigating.

## Run Test Cases

Assume I have three mutation generation methods and generated three sets of mutant programs. In the second part of the project, I plan to have four running cases.

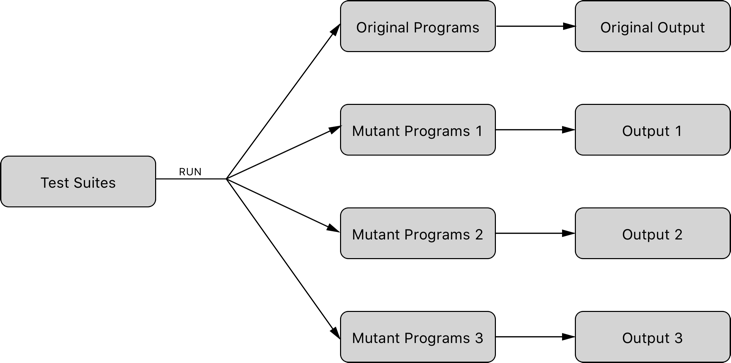


Figure 3. Run test suites.

.

The test suites will be run as input to the original set of programs as well as three mutant sets of programs. After each running cases, the outputs from each set of programs will be noted for the next part of the project.

## Assert and Conclude.

In this part, I will have four sets of outputs coming four sets of programs, one of which comes from original programs and three of which come from three sets of mutant programs.

I will combine them into three comparisons: the original programs outputs with each of three sets of mutant programs. In every comparison, I compare the outputs from original programs with outputs from the mutant programs to calculate the Mutation Testing Criteria (aka mutation score).

Finally, I get three mutation scores. All these scores can be used to evaluate the effectiveness of the test suites. And let’s see what I can assert by comparing the three mutation scores after the implementation.

# ACKNOWLEDGMENTS

My thanks to ACM SIGCHI for allowing us to modify templates they had developed.

# REFERENCES

1. Mutation 2000: Uniting the Orthogonal by A. Jefferson Offutt and Roland H. Untch.
2. Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41. April 1978.
3. Tim A. Budd, Mutation Analysis of Program Test Data. PhD thesis, Yale University New Haven CT, 1980.
4. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34–41, 1978.
5. A. Jefferson Offutt. 1992. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. 1, 1 (January 1992), 5-20.
6. MuJava: An Automated Class Mutation System by Yu-Seung Ma, Jeff Offutt and Yong Rae Kwo.
7. Mutation Operators for Concurrent Java (J2SE 5.0) by Jeremy S. Bradbury, James R. Cordy, Juergen Dingel.
8. Mutation of Java Objects by Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji.