**Automated Mutation Testing Framework**

**Project Final Report**

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 will describe the specific details of how this Automated Mutation Testing Framework was developed. Mainly about these topics: brief introduction to mutation testing, details about mutators embedded in the framework, how the mutators are implemented by ASM, how the logic throughout the framework is implemented.

**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; Maven; github; shell script.

# INTRODUCTION

The Mutation testing is conceptually quite simple1.

Faults (or mutations) are automatically seeded into code, then the tests are run. If the tests fail then the mutation is killed, if the tests pass then the mutation lived.

The quality of your tests can be gauged from the percentage of mutations killed.

To put it another way – the automated mutation testing framework runs a program’s unit tests against automatically modified versions of the program. When the program code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite.

Let’s talk about some specific details about the framework of this project.

The framework generally works as the above descripted. Firstly, the framework compiles the whole target project getting the origin program byte-code (class files), these byte-code is not mutated, I would call it clean program (mutated program would be called dirty program in opposite). Test on this clean program is supposed to have a successful result.

For every mutator, the framework inserts mutator to the clean program generating a dirty program. Then run the tests cases/suites with dirty program. If tests cases/suites turn out to be fail, then the we say this mutator is killed. Being killed means the test has a good quality.

Every time a mutator is inserted and tests have run, the framework will record the test and assert whether this mutator is killed or not. After that, the dirty program will be restored to the clean program which was backed up before. The clean program will be manipulated with the next mutator.

# MUTATORS

Mutations are performed on the byte code generated by the compiler rather than on the source files2. This approach has the advantage of being generally much faster and easier to incorporate into a build, but it can sometimes be difficult to simply describe how the mutation operators map to equivalent changes to a Java source file.

The five mutators I introduced in this section, are those embedded in the framework of this project, they are Conditionals Boundary Mutator, Negate Conditionals Mutator, Math Mutator, Increments Mutator and Invert Negatives Mutator.

## Conditionals Boundary Mutator

The conditionals boundary mutator replaces the relational operators <, <=, >, >=.

with their boundary counterpart as per the table below.

Table 1. Conditionals Boundary Mutation

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| < | <= |
| <= | < |
| > | >= |
| >= | > |

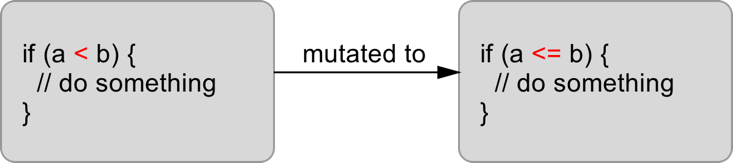


Figure 1. Conditionals Boundary Mutator Example

## Negate Conditionals Mutator

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

**Table 2. Negate Conditionals Mutation**

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| ＝＝ | != |
| != | == |
| <= | > |
| >= | > |
| < | >= |
| > | <= |

## Math Mutator

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

**Table 3. Math Mutation**

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| + | - |
| - | + |
| \* | / |
| / | \* |
| % | \* |
| & | | |
| | | & |
| ^ | & |
| << | >> |
| >> | << |
| >>> | << |

## Increments Mutator

The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

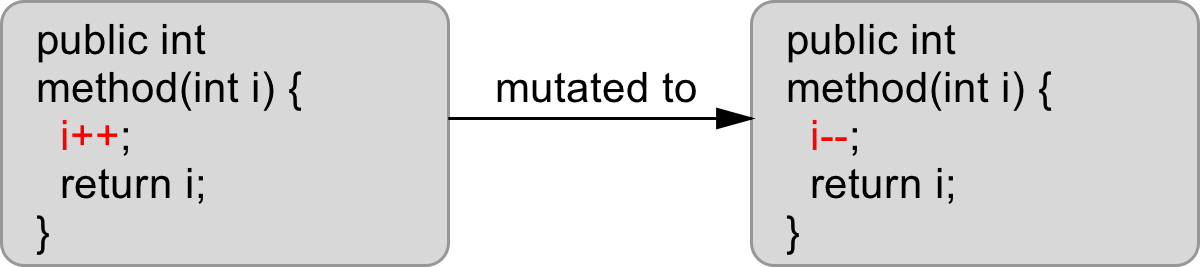


Figure 3. Increments Muator Example

## Invert Negatives Mutator

The invert negatives mutator inverts negation of integer and floating point numbers.

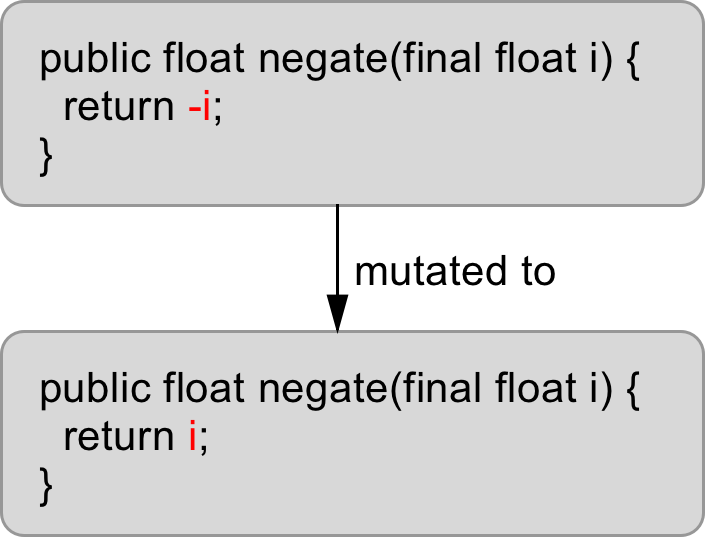


Figure 4. Invert Negatives Mutator Example

# MUTANT GENERATION

The approach of mutant generation in the framework of this project is manipulating the compiled byte-code. This will be down using the third party library ASM.

Manipulating byte code, is generally much faster and generally easier to integrate into a build, can potentially create mutants without access to source files and same mutation operators can in theory work for other JVM languages.

However, it is more difficult to develop mutation operators this way. Also, mutations are divorced from the source code and may not be representative of errors a programmer would make. And it can be hard to describe / explain the mutations that must work around intricacies of the JVM.

The manipulating done by the framework is a statement-level manipulating. By saying ‘statement-level’, I mean the manipulating target is not a method, nor a class. The manipulating target is a computing operator or a logical operator which is in a line of statement.

When the byte-code runs, it is a ClassVisitor for every class in the runtime, for every method in a specific class, the ClassVisitor will initialize an instance of MethodVisitor, visiting the method. And the MethodVisitor would have different function in it, dealing with different instrument.

So what the framework would do with ASM is firstly having an customized ClassVisitor. In this customized ClassVisitor, the function that initialize the MethodVisitor will be override, replace the default MethodVisitor with a customized MethodVisitor. In the customized MethodVisitor, functions will be overridden according to the mutator’s requirements.

## Conditional Boundary Mutator

Conditional boundary operators are visited in the function ‘visitJumpInsn(int opcode, Label label)’, so in order to generate conditional boundary mutator, the framework is going to override the function ‘visitJumpInsn(int opcode, Label label)’.

The opcode parameter in the function indicates the conditional boundary operator. The Opcodes interface defines the JVM opcodes, access flags and array type codes.

The framework has a hashtable, storing which operator opcode should be mutated to which operator (mapping relationship is discussed in the previous section).

When the function ‘visitJumpInsn(int opcode, Label label)’ is called, the framework would check whether the opcode is contained in the hashtable mentioned in the previous paragraph. If so, call the ‘super.visitJumpInsn(int opcode, Label label)’ with the opcode that should be mutated to as the opcode parameter. If not contained in the hashtable, call the ‘super.visitJumpInsn(int opcode, Label label)’ with the origin opcode as the opcode parameter.

Figure 5 below shows how the hashtable is initialized and populated. Figure 6 below is a kind of snippet of pseudocode helps explaining what’s going on when Conditional Boundary Mutator is generated.

Notice that the code in figures is incomplete, tons of logic code controlling the mutator insertion is not shown in these snippets of code.

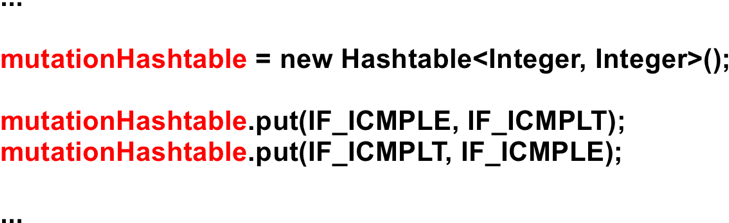


Figure 5. Initialize and Populate Mutation Hashtable

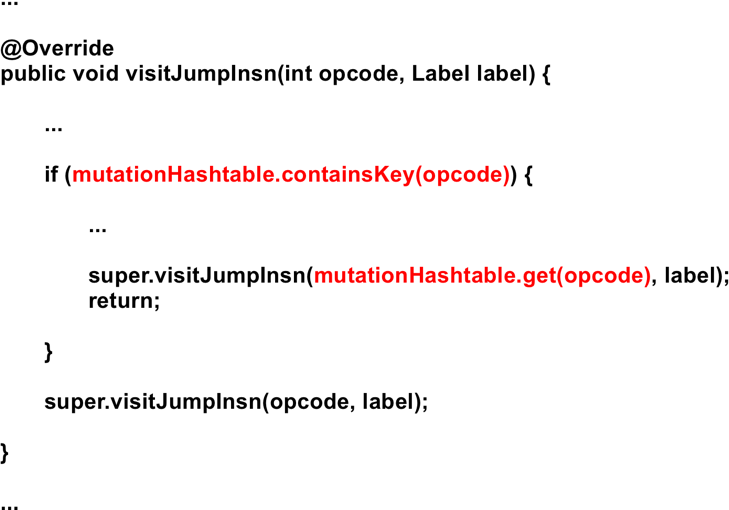


Figure 5. Generate Conditional Boundary Mutator

## Math Mutator

Math operators are visited in the function ‘visitInsn(int opcode)’, so in order to generate math mutator, the framework is going to override the function ‘visitInsn(int opcode)’.

The framework also has a hashtable in this mutator generating, storing which operator opcode should be mutated to which operator.

When the function ‘visitInsn(int opcode)’ is called, the framework would check whether the opcode is contained in the hashtable. If so, call the ‘super.visitInsn(int opcode)’ with the opcode that should be mutated to as the opcode parameter. If not contained in the hashtable, call the ‘super. visitInsn(int opcode)’ with the origin opcode as the opcode parameter.

Figure 7 below is a kind of snippet of pseudocode helps explaining what’s going on when Math Mutator is generated.



Figure 7. Generate Math Mutator

## Other Mutators

The other three mutators that we haven’t discussed are: Negate Conditionals Mutator, Increments Mutator and Invert Negatives Mutator.

Negate Conditionals Mutator is very similar to Conditional Boundary Mutator. Both of two manipulate condition operators to generate mutation.

Increments Mutator and Invert Negatives Mutator are very similar to Math Mutator because they override the function ‘visitInsn(int opcode)’ to generate mutation.

Readers can refer the previous two subsection knowing about the other three mutators’ generating.

Notice that the code in figures is incomplete, tons of logic code controlling the mutator insertion is not shown in these snippets of code.

# MUTANT INSERTION

## Insertion Approach

The insertion approach is about how the framework find the class files to apply the customized ClassVisitor and customized MethodVisitor. The approach is consisted of two parts, one is Shell-script and the other is Java.

Shell-script’s duty is jumping to the correct directory where all he class files are stored, and then run the Java program. Because maven project has an fixed file structure, class files are always stored in ‘./target/classes’, so Shell-script uses ‘cd target/classes’ command to go into the class files directory.

Before run java command to insert mutators, the Shell-scipt would copy the clean program to for restoration in the future. Java command has the build-path configuration, which connect the ASM library jar file to make sure the Java program works.

## Insertion Strategy

The insertion strategy controls which lines of code should be mutated by the framework. Strategy is implemented in the java insertion program.

I had made a mistake in this step before. At first, my strategy for inserting mutator is a kind of greedy strategy. For example, let’s say I’m inserting Math Mutator, there would be many math operators in a method and even more math operators in a class. What I did is mutating every math operator in the class. This is not correct.

One of the mutation testing purpose is simulating bugs caused by programmer’s mistake. Since even the worst programmer wouldn’t make mistake in every math operator, mutating every math operator obviously obeys the purpose of mutation testing.

The correct insertion strategy should be only mutating a reasonable number of math operations, which is the framework does after being fixed.

The framework has logic code controlling the mutator insertion, which make sure number of changes inserted is reasonable. To be specific, in the framework, at most 1 mutation is inserted into one method, and at most 5 methods are mutated in a class.

# MUTANT ASSERTION

Numbers of killed mutators and live mutators should be recorded and reported at last, evaluating the quality of tests. The framework can do the tests quality assertion somehow.

When tests are executed with a dirty program, a report will be generated into the ‘report’ directory. And the framework will assert whether this mutator is killed or not according to the report of this mutator.

After all the mutators are tested, the framework will give a summary showing numbers of killed mutators and live mutators.

# ACKNOWLEDGMENTS

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

# REFERENCES

1. *Mutation testing systems for Java compared.* (2016, March). Retrieved  from http://pitest.org/java\_mutation\_testing\_systems/
2. *Mutation operators.* (2016, March). Retrieved  from http://pitest.org/quickstart/mutators/
3. *Package java.lang.instrument.* (2016, May). Retrieved  from http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/package-summary.html