

Developing a simple mutation analyzer for the Major Mutation Framework

Group 52

**Divyesh Harit, Rishi Mody, Thuan Binh and
Utkarsh Srivastava**

1. Overview

MAJOR is a mutation analysis framework that divides the mutation analysis process into two steps:

1. Generate and embed mutants during compilation.
2. Running the mutation analysis.

To carry out these functions MAJOR has two components:

1. A compiler-integrated mutator
2. An analysis backend that extends Apache Ant's Junit task

The advantage of using MAJOR is that it can be used for efficient and scalable mutation analysis. It also enables for both strong and weak mutation analysis.

The mutation analyzer has been created as a standalone java program. It takes two arguments as inputs, path to a JAR file and path to the mutants.log file. The JAR file is created from the .class files that are a result of compiling using MAJOR javac and XMutator, which generates and embeds mutants in code. The mutants.log file is generated upon build and lists all the mutants produced from the source code.

The analyzer, as an output, reports information about mutation kill rate and summary of killed mutants, this information, along with runtime of the code, is presented in the form of a Graphical User Interface. Because the number of mutant IDs not killed could be too large, upon compilation of the program, the list of these mutant IDs are stored in a newly created file called unKilledMutants.log.

2. Input and Output Specification:

The aim of the project is to develop a standalone java program for mutation analysis (as a part of the major mutation framework).

Inputs:

1. Source and test files using MAJOR javac and XMutator to generate mutants.log
2. A JAR file, that is created from the source and test class files
3. The mutation analyzer takes two arguments: path to the JAR file and path to the mutants.log file

Outputs:

1. GUI displaying: mutation kill rate, summary of killed mutants, and runtime of the code.
2. unKilledMutants.log that contains the list of mutant IDs not killed.

3. Installation and Execution:

The submitted zip contains (This is the parent directory):

1. Main Source Code for Mutation Analyser: <mutation_analyser>
2. Test Code including Source Code to be Mutated and Test Suite :
 - a. TestCode (our own example suite with 3 classes in 3 corresponding test classes)
 - b. Triangle (code taken from Prof. Rene Just's mutation_results.zip)
 - c. Numerics4j (code taken from Prof. Rene Just's mutation_results.zip)
3. A readMe.txt file describing constituents of the code and steps to Run.

Steps to run:

1. This code assumes that "major" folder is kept in the parent directory.
2. Each TestCode includes a "build.xml" file and a custom script called "addMutants.sh". The build.xml contains the path for javac from the "major" folder present in the parent directory. As our code depends on an input in the form of a "JAR", run the following from the path of the chosen test code folder:
 - a. `chmod +x addMutants.sh`
 - b. `./addMutants.sh`This will generate .class files with embedded mutants and a separate "mutants.log" file. The packed Jar comprising all .class files will be placed in the "Jars" folder in the parent path directory as "Mutants.log". It should be noted that "junit.jar" is a dependency for code compilation and is kept in lib folder in the current directory.
3. Now, shift to the main source code for Mutation Analyser and compile it using:
 - a. `ant clean compile`Note that both the dependencies "junit.jar" and "config.jar" are present in the lib folder.
4. To run the code, use the command:
 - a. `java -cp bin:lib/junit.jar:lib/config.jar: main.Main <arg1> <arg2>`
 - b. Here program arg1: Full Qualified Path to the created Mutants.jar stored in "Jars" directory, for e.g.: `"/home/utkarsh1404/cs620Code/Jars/Mutants.jar"`
 - c. And program arg2: Full Qualified Path to the created mutants.log file of the chosen Test Code, for e.g.: `"/home/utkarsh1404/cs620Code/Triangle/mutants.log"`
5. The final output is the GUI and generated unKilledMutants.log file with unKilled mutant summary.
6. Alternative: To run it from EclipseIDE, directly add program arguments as mentioned above through "Run Configurations" and run the code.

****Note**:** Inspect the build.xml file for any "path" inconsistencies if the code has errors for path upon compilation/run.

4. Code Documentation:

****Note**:** The code has been commented extensively to provide specifications in the code.

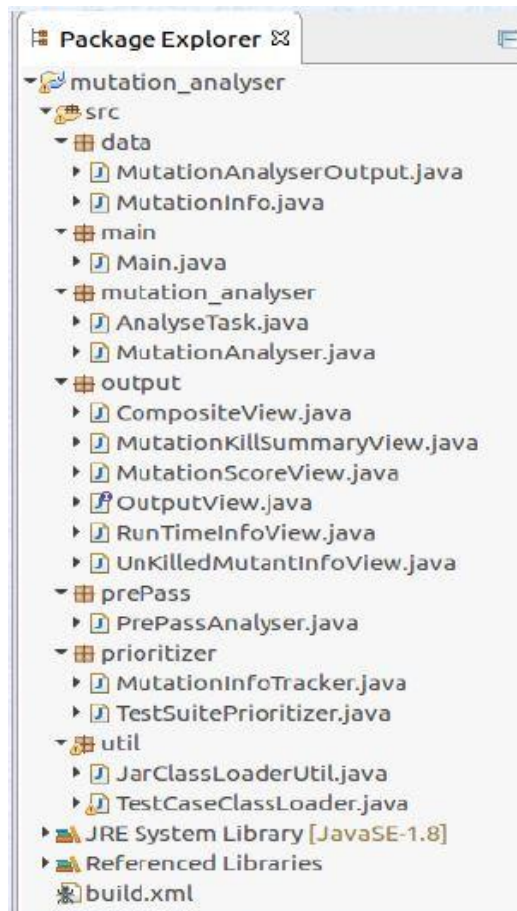
The written code tries to simulate the working of the Mutation Analyser present as a part of the Major Mutation Framework. The program aims to take a test suite and a list of mutants as input and generates a final mutation score with associated mutation summary. For the same, the code has been separated based on individual functionalities of each part and the order in which they appear in the code flow. The documentation is divided into 3 parts:

1. Basic Idea and Code Flow
 2. Design Choices - Software Engineering Practice
 3. Block Diagram to define each logical code component
 4. Libraries and Data Structures Used
-
1. After receiving the input JAR of compiled mutant files and a list of generated mutants in a log, as a pre-processing step, the code handled:
 - a. Extract out and load all the class files that correspond to “Test” files in the Jar.
 - b. Extract coverage information of mutants and run-time for each test case in each test class and store in a Map to use later for test suite-prioritization.
 2. For each test class, prioritize the test functions based on a metric that favors test cases that take relatively less time to run and cover more mutants at the same time. Also store map of test classes against list of mutants covered by them for use in optimization later (call it TestClass-MutationMap).
 3. Now to analyze the mutant kill rate, each test case from each test class is run on each mutated code. It's failure/timeout status is stored to determine whether it was killed or not. Timeout is handled via the use of ExecutorService (Tunable Timeout). Optimization used in this step include:
 - a. Making sure to run faster test cases and that cover more mutants first in each iteration.
 - b. “TestClass-MutationMap” helps by not running the iteration for a mutant which is not covered by any test case for a given test class.
 - c. The iteration is updated with “break” once even a single test case fails in the presence of the mutant. This would ensure not running further un-necessary iterations of test cases and test classes as the mutant has already been killed. This comes from the fact that each mutated source code enables a single mutation in the source code.
 4. The output object stores the mutation info-retrieved by running the full Mutation Analysis in the last step. This is further used to be displayed individually in a UI with unkilld mutant information getting stored in “unKilledMutants.log” file.

****Note**:** If any exception is caught, we simply display the stack trace and not the particular exception.

5. Design Choices- Software Engg. Practice:

- 1) One of the critical design choices we made was to properly separate distinct operations, a concept commonly known as separation of concerns. In our implementation, in the following three phases, we have created separate packages according to their functionality, thus allowing future extensions while not requiring modifications to the original code:
 - A. Pre-pass: Record run-time and coverage of each test case on each mutant.
 - B. Prioritization: Prioritize test cases according to their pre-pass information, and create a hash-map of test classes and the test case names.
 - C. Execution and analysis: Run test cases on all mutants, thus producing mutant kill information.

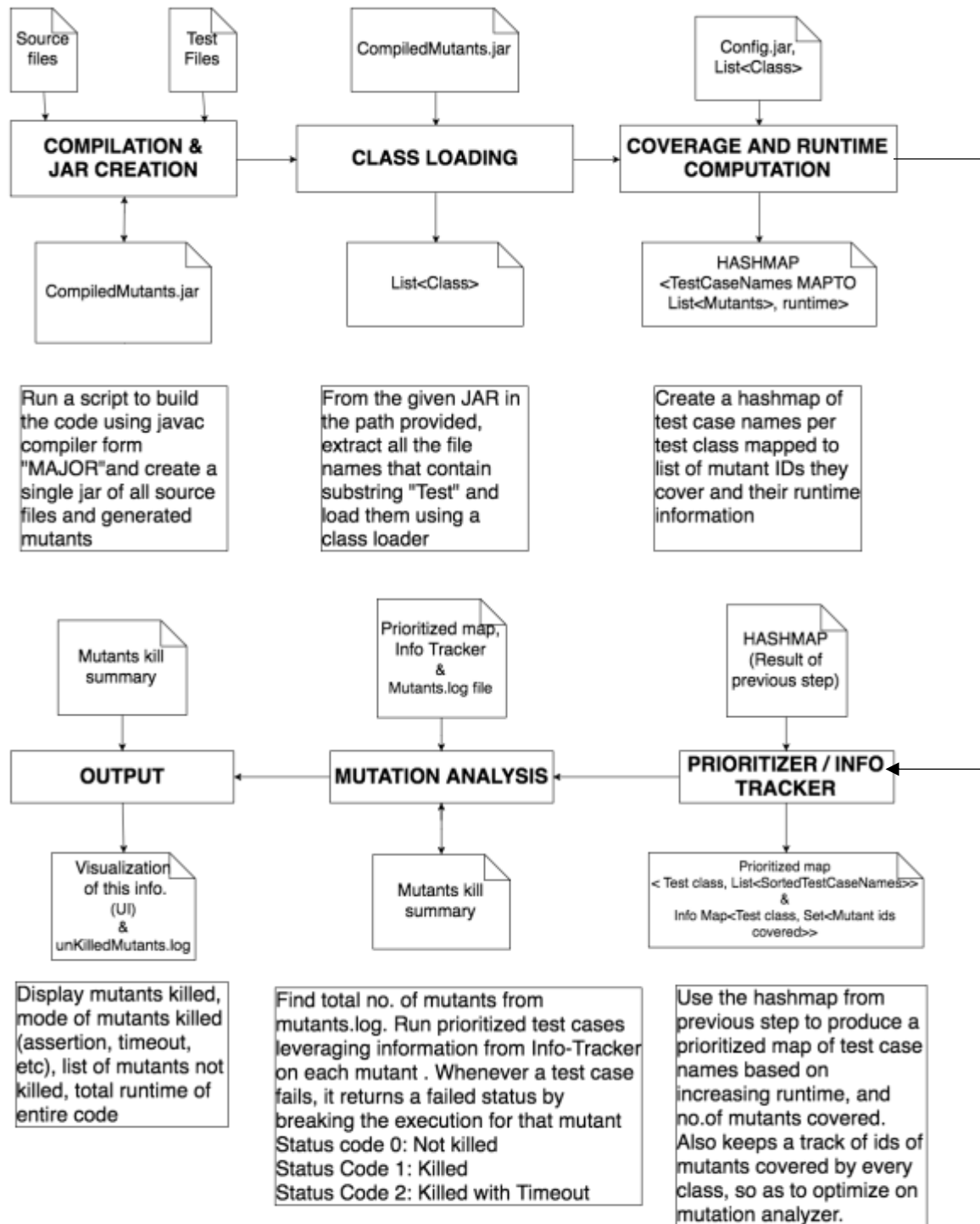


The screenshot above displays the package structure of the design, which includes:

- A. **data**: Loader for the visualization (MutationAnalyserOutout), and list of IDs of covered mutants and runtime of each covered mutant in each test class (MutationInfo)
- B. **main**: Used to trigger the operations sequentially
- C. **output**: GUI that displays the mutant kill summary, mutation score, total run-time of the code and summary of the unkilled mutants.

- D. **prePass**: Perform the pre-pass.
 - E. **prioritizer**: Prioritize the test cases on the basis of run-time and coverage information.
 - F. **util**: Loads the single jar file of source and test classes, and use that to extract classes that contain "Test" substring
- 2) Usage of **composite pattern** for the visualization, with the aim being a more flexible output.
 - 3) Usage of **iterator pattern** for cycling through multiple functions (test cases) for the test classes.

6.Block Diagram: Logical Components



7. Libraries and Utilities used:

1. **InputStreamReader**: Read contents from mutants.log file
2. **URLClassLoader**: To load contents of mutants.jar file
3. **JarEntry**: To iterate over elements in the JAR
4. **JUnitCore, Request, Result**: To create a request, run and record the results of a test case in a test file
5. **JFrame**: GUI output
6. **ExecutorService, Future, Callable**: For handling timeouts and Thread Executions

8. Data Structures used:

1. **MutationInfoClass**: Holds information from the pre-pass phase
2. **MutationAnalyzerOutput**: Holds mutant kill information to display output
3. **Map <String, List<String>> PrioritizedTests**: Creates a map of class names to a prioritized list of ordered test cases.
4. **Map <String, Set<Integer>> MutationInfoPerClass**: Creates a map of class names to a set of IDs of covered mutants.

9. Optimizations:

In order to make the program execute faster and be more efficient, we have performed the following optimizations:

1. In order to reduce the overall run-time of the code, we use run-time and coverage information for each test case on each mutant, and use it to prioritize test case execution by creating a prioritized hash-map of test case names and the classes they belong to.
2. We save additional run-time by killing a mutant as soon as a test case for that iteration fails. This results in not calling any further unnecessary test case executions.
3. In order to avoid wasting time being stuck on infinite loops, we use an executor service with a timeout of 1 second (configurable).

10. Limitations:

We realized that our project suffers from the following limitations:

1. We have implemented the analyzer for strong mutation analysis, and thus it takes run-time and coverage information, but does not consider program state information so that weak mutation analysis can be performed.
2. A single JAR file as required as an initial input. For someone not willing to use the program this way, the analyzer simply won't work.

3. We have assumed that the test class names end with a “Test” substring, so that we can load them from the JAR file. While an extremely common practice, it is possible that a developer follows some other naming criteria for their test classes, in which case, the analyzer would fail.

11. Example Runs:

Note: Contents of unKilledMutants.log file have not been listed here.

1. **TestCode:**

Number of mutants generated: 106
Number of mutants killed: 94
Mutation score: 88.7%
Total number of mutants killed due to timeout: 2
Total number of unKilled mutants: 12
Total runtime: 2.38 seconds (System Time)

2. **Triangle:**

Number of mutants generated: 139
Number of mutants killed: 130
Mutation score: 93.53%
Total number of mutants killed due to timeout: 0
Total number of unKilled mutants: 9
Total runtime: 1.10 seconds (System Time)

3. **Numerics4J:**

Number of mutants generated: 16708 (-XMutator = ALL)
Number of mutants killed: 1037
Mutation score: 64.26%
Total number of mutants killed due to timeout: 228
Total number of unKilled mutants: 5971
Total runtime: ~828 seconds (System Time)

12. Code Screenshots:

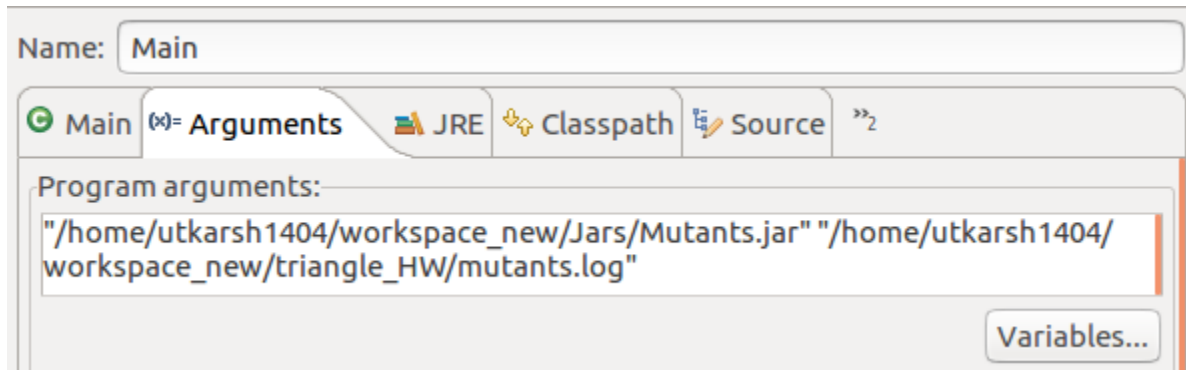


Figure 1 Setting the path for jar and log file under arguments for Eclipse IDE



Figure 2 System output during program execution

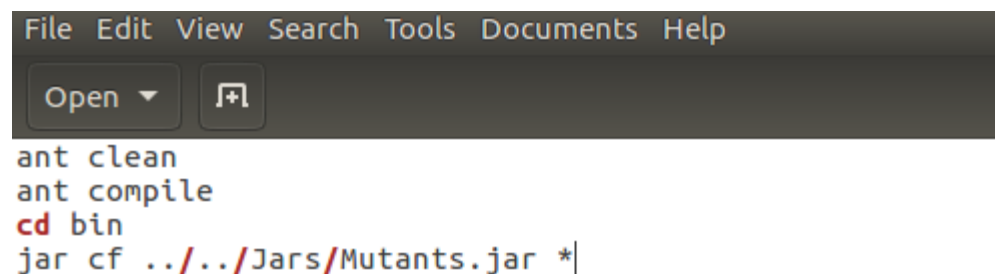


Figure 3 Code of the script which generates the initial jar file

nts.log (-/workspace_new/triangle_HWJ) - gedit


```
Open: 
1: LVR:0:POS:triangle.Triangle@classify:20:0 |==> 1
2: LVR:0:NEG:triangle.Triangle@classify:20:0 |==> -1
3: ROR:==(int,int):<(int,int):triangle.Triangle@classify:20:a <= 0 |==> a < 0
4: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:20:a <= 0 |==> a == 0
5: ROR:==(int,int):TRUE(int,int):triangle.Triangle@classify:20:a <= 0 |==> true
6: LVR:0:POS:triangle.Triangle@classify:20:0 |==> 1
7: LVR:0:NEG:triangle.Triangle@classify:20:0 |==> -1
8: ROR:==(int,int):<(int,int):triangle.Triangle@classify:20:b <= 0 |==> b < 0
9: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:20:b <= 0 |==> b == 0
10: ROR:==(int,int):TRUE(int,int):triangle.Triangle@classify:20:b <= 0 |==> true
11: COR:||(boolean,boolean):!(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 |==> a <= 0 != b <= 0
12: COR:||(boolean,boolean):LHS(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 |==> a <= 0
13: COR:||(boolean,boolean):RHS(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 |==> b <= 0
14: COR:||(boolean,boolean):TRUE(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 |==> true
15: LVR:0:POS:triangle.Triangle@classify:20:0 |==> 1
16: LVR:0:NEG:triangle.Triangle@classify:20:0 |==> -1
17: ROR:==(int,int):<(int,int):triangle.Triangle@classify:20:c <= 0 |==> c < 0
18: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:20:c <= 0 |==> c == 0
19: ROR:==(int,int):TRUE(int,int):triangle.Triangle@classify:20:c <= 0 |==> true
20: COR:||(boolean,boolean):!(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 || c <= 0 |==> (a <= 0 || b <= 0) != c <= 0
21: COR:||(boolean,boolean):LHS(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 || c <= 0 |==> a <= 0 || b <= 0 || c <= 0
22: COR:||(boolean,boolean):RHS(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 || c <= 0 |==> c <= 0
23: COR:||(boolean,boolean):TRUE(boolean,boolean):triangle.Triangle@classify:20:a <= 0 || b <= 0 || c <= 0 |==> true
24: LVR:0:POS:triangle.Triangle@classify:23:0 |==> 1
25: LVR:0:NEG:triangle.Triangle@classify:23:0 |==> -1
26: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:24:a == b |==> a <= b
27: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:24:a == b |==> a >= b
28: ROR:==(int,int):FALSE(int,int):triangle.Triangle@classify:24:a == b |==> false
29: LVR:0:POS:triangle.Triangle@classify:25:1 |==> 0
30: LVR:0:NEG:triangle.Triangle@classify:25:1 |==> -1
31: AOR:+=(int,int):%(int,int):triangle.Triangle@classify:25:trian + 1 |==> trian % 1
32: AOR:+=(int,int):*(int,int):triangle.Triangle@classify:25:trian + 1 |==> trian * 1
33: AOR:+=(int,int):-(int,int):triangle.Triangle@classify:25:trian + 1 |==> trian - 1
34: AOR:+=(int,int):/(int,int):triangle.Triangle@classify:25:trian + 1 |==> trian / 1
35: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:27:a == c |==> a <= c
36: ROR:==(int,int):>=(int,int):triangle.Triangle@classify:27:a == c |==> a >= c
37: ROR:==(int,int):FALSE(int,int):triangle.Triangle@classify:27:a == c |==> false
38: LVR:0:POS:triangle.Triangle@classify:28:2 |==> 0
39: LVR:0:NEG:triangle.Triangle@classify:28:2 |==> -2
40: AOR:+=(int,int):%(int,int):triangle.Triangle@classify:28:trian + 2 |==> trian % 2
41: AOR:+=(int,int):*(int,int):triangle.Triangle@classify:28:trian + 2 |==> trian * 2
42: AOR:+=(int,int):-(int,int):triangle.Triangle@classify:28:trian + 2 |==> trian - 2
43: AOR:+=(int,int):/(int,int):triangle.Triangle@classify:28:trian + 2 |==> trian / 2
44: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:30:b == c |==> b <= c
45: ROR:==(int,int):>=(int,int):triangle.Triangle@classify:30:b == c |==> b >= c
46: ROR:==(int,int):FALSE(int,int):triangle.Triangle@classify:30:b == c |==> false
47: LVR:0:POS:triangle.Triangle@classify:31:3 |==> 0
48: LVR:0:NEG:triangle.Triangle@classify:31:3 |==> -3
49: AOR:+=(int,int):%(int,int):triangle.Triangle@classify:31:trian + 3 |==> trian % 3
50: AOR:+=(int,int):*(int,int):triangle.Triangle@classify:31:trian + 3 |==> trian * 3
51: AOR:+=(int,int):-(int,int):triangle.Triangle@classify:31:trian + 3 |==> trian - 3
52: AOR:+=(int,int):/(int,int):triangle.Triangle@classify:31:trian + 3 |==> trian / 3
53: LVR:0:POS:triangle.Triangle@classify:33:0 |==> 1
54: LVR:0:NEG:triangle.Triangle@classify:33:0 |==> -1
55: ROR:==(int,int):<=(int,int):triangle.Triangle@classify:33:trian == 0 |==> trian <= 0
56: ROR:==(int,int):>=(int,int):triangle.Triangle@classify:33:trian == 0 |==> trian >= 0
57: ROR:==(int,int):FALSE(int,int):triangle.Triangle@classify:33:trian == 0 |==> false
58: AOR:+=(int,int):%(int,int):triangle.Triangle@classify:34:a + b |==> a % b
```

Figure 4 Contents of mutants.log file, which lists all the mutants



Figure 5 GUI, listing the mutant kill summary and total code run-time



Figure 6 GUI, listing the mutant kill summary and total code run-time for Numerics4J