An Analysis of Mutation Operator in Mutation Testing

Mr. Bajirao Baban Kondbhar[1], Prof. Dr. Emmanuel M[*2]

[1,2] *Department of Information Technology*
Pune Institute of Computer Technology,
SPPU University of Pune, India.

Abstract— Normal observation of Mutation testing is it costs more in case of test design and execution. Number of mutation operators like condition operator, arithmetic operator etc. Every mutation operator generates equivalent mutants, and cost to design test cases and execution of test cases are more in case of equivalent mutant. This paper aims at to reduce equivalent mutants and test cases related to them. In mutation testing, mutant generation is one of the tasks before mutant execution for killed and live mutant. In this paper, authors have proposed an algorithm to test equivalent mutant and reduce the work of test design for identified equivalent mutant. The goal of the paper is to improvise the effectiveness of mutation testing by applying quality test suite and analysing mutation score. Sample java program and muJava mutation system [6] is used to analyse mutation operator and mutation score.

***Keywords*— Mutation Testing, Mutation Operator, Test case generation and Execution**.

I. INTRODUCTION

Mutation testing is fault based testing. Aim of the Tester is to reveal the software defect or fault. Black box test case design technique deals with what is specified. Alternatively, it checks for functional requirements, whether mapped with expectations i.e. comparison with actual outcomes and expected outcomes. In white box test case design technique internal structure of the system under test is evaluated. Mutation testing is one of the white box method because tester deals with code details, tester seed defect in given program and test behaviour of the given program whether it's behave correctly or not? Authors have used java program for analysis of mutation operators and muJava mutation system. The traditional process of mutation analysis is illustrated in Figure 1. In mutation analysis, original code is input for testing, original program generates versions of code by changing statement, logical, arithmetic and conditional operator that is called mutation operator [1]. Overall task is to generate mutant and then design test suite which consists of number of test cases. Execute test suite and check for killed and live mutant. If live mutant is there then analyse equivalent mutant. Redundancy or cyclic nature for equivalent mutant in traditional mutation analysis is checked. Our aim is to reduce the mentioned problem and improve upon the test suite quality and mutation score. Mutation testing is applied for testing a program. While other software testing techniques applied on the correct functionality of the program. The main idea is to create good test suite of test cases rather than trying to detect the faults.

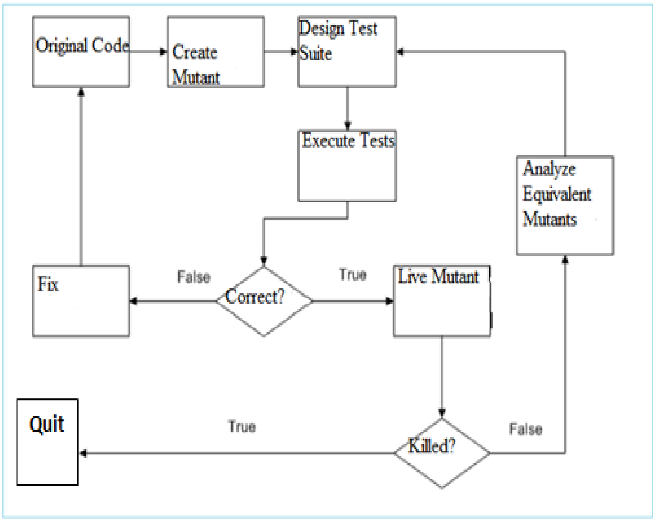


Figure.1 Mutation Process

Good test cases are those which are having high probability to detect uncovered defect. A good test case will detect all the possible faults in a software program.

II. RELATED WORK

Following are the overall steps used for mutation testing.

1. Consider original code or program C.
2. Assume mutation operator like conditional, arithmetic, logical, relational etc. and produce the versions of the original program i.e. mutants C'.
3. Design test case for the given mutant C' and original program C.
4. Analyses outcomes or results of C and C'.
5. If the results are not equivalent or equal that mutant is killed mutant i.e. K
6. If the results are same there may be two reasons:
   i. Mutants are difficult to kill. Design another test case to kill the mutant.
   ii. Mutants are having same behavior i.e. there is equivalent mutant [4]

It is necessary to achieve high mutation score on # of killed mutants and # of non equivalent mutants. Practically value of mutation score is 1. If MS is mutation score then 0 < = MS < = 1 need to evaluate [1]. $MS = \frac{K}{M_i - E}$ Where

K = # of killed mutants, M_i is total # of mutants and E is # of equivalent mutants [2].
Lin Deng, Jeff Offutt, Nan Li [5] worked on statement deletion operator (SDL), they observed that SDL perform well than other mutation operator. Mutation score of SDL mutant is 92% and 40% of equivalent mutants has been discovered.
Pedro Reales Mateo, Macario Polo Usaola [3] implemented Bacterio tool for mutation testing, Bacterio tool perform better in mutant design and execution, many tasks of mutation analysis is atomized by Bacterio tool.
In MuJava, [6] method level mutant and class level mutants are generated, It record mutation score as well as calculate computational time to generate and execute the mutants. Many tools are implemented for mutation testing like mujava, Bacterio, Jester, etc. to reduce the cost and effort of software testing.

III. PROBLEMS IN MUTATION TESTING

Mutation testing is effective to detect faults in code or program. Drawback is amount of human effort to measure the correctness of originality of code i.e. human oracle problem to analyses original code and mutant code outcomes with each test case [1]. Another major problem of equivalent mutants is how to minimize the effort of test case design on equivalent mutants.
Problem of input options for test case i.e. input domain of test cases designed for test suite. Which inputs want to select to test case is one of the question? Every test case wants to execute on valid and invalid inputs.
Although it is impossible to resolve these issues completely, we can automate the mutation testing process and improve scalability.

IV. PROPOSED ARCHITECTURE

Proposed architecture is in figure 2 includes the following concepts regarding Mutation Analysis.

1. P is original program which is input for Mutation Analysis.
2. By observing original program architecture generates n-mutants by making some change in original code i.e. apply mutation operator.
3. Module execution mutant generates execution history of all generated Mutants.
4. If original code output and mutant code output is not mapping i.e. the mutant is killed mutant, keep as it in Test Suite (don't remove corresponding Test case).
5. If original code output and mutant code output is mapping i.e. the mutant is Equivalent mutant and remove that Mutant from Test Suite (remove corresponding Test case). Use proposed algorithm and static testing aspects.
6. Execute all mutants (corresponding Test cases) in given Test Suite, calculate Mutation Score and computational time to execute all mutants.
7. Compare the accuracy with existing Mutation Testing Tools like MuJava.[6]

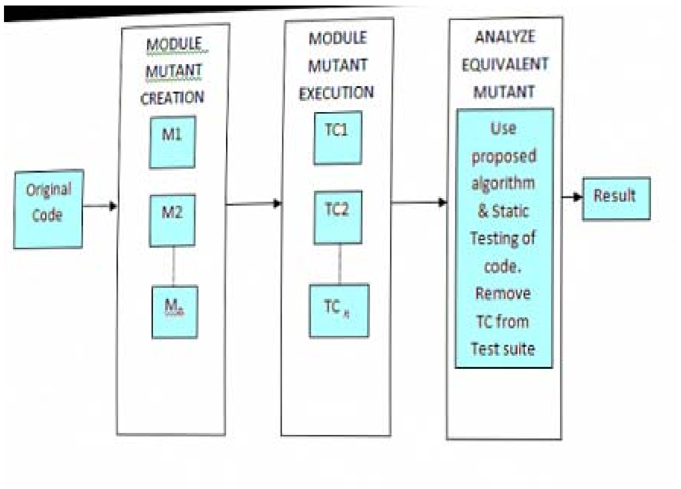

Original Code
MODULE MUTANT CREATION
M1
M2
Mn
MODULE MUTANT EXECUTION
TC1
TC2
TCn
ANALYZE EQUIVALENT MUTANT
Use proposed algorithm & Static Testing of code. Remove TC from Test suite
Result

Figure.2 Proposed Architecture.

V. PROPOSED ALGORITHM

```
Begin
Select Mutation Operator
K = Number of Mutation operator
For n = 0 to K .
Generate Mutant()
Execute Mutant()
Compare original code output with Mutant code output.
If Mutant = Killed Mutant
  ∀ Mutant assign
Calculate_Mutation_Score()
Measure test suite quality ()
Else if Mutan t≠ Killed Mutant
Remove ( )
//It is equivalent mutant; remove the test case related to mutant from test suite.
Else
Add ( )
// new test case in test suite if valid and invalid inputs detects killed mutants.
End
```

VI. MATHEMATICAL MODEL

Let P is original program and M is mutant generated from original program. T_s is test suite which is given by following equation.

$$T_s = \sum_{n=0}^{k} T_{c_i} \quad \text{.................} \qquad (1)$$

Where T_{c_i} is number of test cases in test suite. This is given by following equation.

$$T_c = \{M_1, M_2, ...M_k\} \text{..................} \qquad (2)$$

$$MS = \frac{K}{M_i - E} \text{...........................} \qquad (3)$$

Where K = # of killed mutants, M_i is total # of mutants and E is # of equivalent mutants. To analyse the effectiveness of mutation score following equation is used. MS is mutation score.

$$MS = \begin{cases} 1 \ldots\ldots\ldots\ldots\ldots practical\ value \\ \approx 1 \ldots\ldots\ldots Expected\ Results \end{cases} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (4)$$

VII. Experimental Setup

We used MuJava mutation system which is available on website https://cs.gmu.edu/~offut/mujava/ and simulate the given problems and de-merits of mutation testing. [6] We used java program gdMultClass.java. Original input code for gdMultClass is given below in figure 3.

Arithmetic and logical mutation operator are considered to generate mutants for gdMultClass.java program; total 28 numbers of mutants are generated. In second module of proposed architecture mutant execution is there i.e. design test cases for given mutants and keep all test cases in test suite or test set. In third module, analysis of equivalent mutant is there. We applied our proposed algorithm and compared mutant execution history with MuJava mutation system.

Step 1: Mutant Generation.

Sample mutant generated by MuJava mutation system [6] as per given figure 3 i.e. first module in our proposed architecture.

Input:

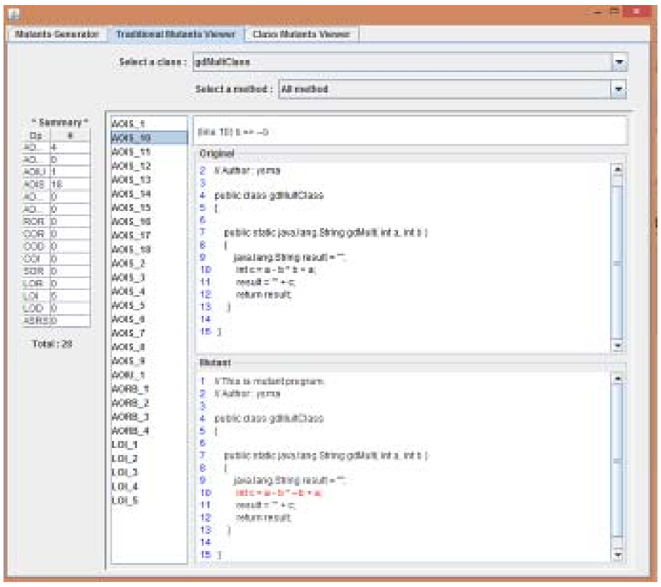
Original (input) code.

Processing:

Apply mutation operator (Arithmetic, Relational, Logical and Conditional). Prepare versions of original code.

Output:

Created Mutant i.e. Mutant Code.



Select a class : gdMultClass

Select a method : All method

* Summary *

Op	#
AO...	4
AO...	0
AOIU	1
AOIS	18
AO...	0
AO...	0
ROR	0
COR	0
COD	0
COI	0
SOR	0
LOR	0
LOI	5
LOD	0
ASRS	0

Total : 28

AOIS_1, AOIS_10, AOIS_11, AOIS_12, AOIS_13, AOIS_14, AOIS_15, AOIS_16, AOIS_17, AOIS_18, AOIS_2, AOIS_3, AOIS_4, AOIS_5, AOIS_6, AOIS_7, AOIS_8, AOIS_9, AOIU_1, AORB_1, AORB_2, AORB_3, AORB_4, LOI_1, LOI_2, LOI_3, LOI_4, LOI_5

(line 10) b => --b

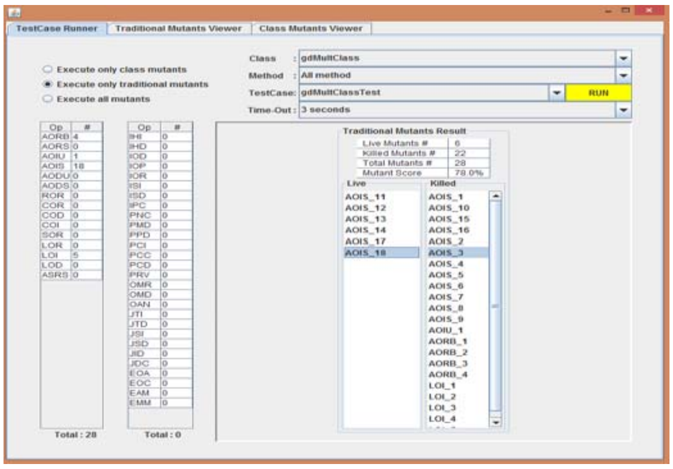
Original
```
// Author : yema

public class gdMultClass
{

   public static java.lang.String gdMult( int a, int b )
   {
      java.lang.String result = "";
      int c = a - b * b + a;
      result = "" + c;
      return result;
   }

}
```

Mutant
```
// This is mutant program.
// Author : yema

public class gdMultClass
{

   public static java.lang.String gdMult( int a, int b )
   {
      java.lang.String result = "";
      int c = a - b * --b + a;
      result = "" + c;
      return result;
   }

}
```

Figure 3: Mutant Generation.

Step 2: Mutant Execution

Sample mutant executed by MuJava mutation system [6] as per given snapshot i.e. second module in our proposed architecture.



○ Execute only class mutants
● Execute only traditional mutants
○ Execute all mutants

Class : gdMultClass
Method : All method
TestCase: gdMultClassTest
Time-Out : 3 seconds

Op	#
AORB	4
AORS	0
AOIU	1
AOIS	18
AODU	0
AODS	0
ROR	0
COR	0
COD	0
COI	0
SOR	0
LOR	0
LOI	5
LOD	0
ASRS	0

Total : 28

Op	#
IHI	0
IHD	0
IOD	0
IOP	0
IOR	0
ISI	0
ISD	0
IPC	0
PNC	0
PMD	0
PPD	0
PCI	0
PCC	0
PCD	0
PRV	0
OMR	0
OMD	0
OAN	0
JTI	0
JTD	0
JSI	0
JSD	0
JID	0
JDC	0
EOA	0
EOC	0
EAM	0
EMM	0

Total : 0

Traditional Mutants Result

Live Mutants #	6
Killed Mutants #	22
Total Mutants #	28
Mutant Score	78.0%

Live: AOIS_11, AOIS_12, AOIS_13, AOIS_14, AOIS_17, AOIS_18

Killed: AOIS_1, AOIS_10, AOIS_15, AOIS_16, AOIS_2, AOIS_3, AOIS_4, AOIS_5, AOIS_6, AOIS_7, AOIS_8, AOIS_9, AOIU_1, AORB_1, AORB_2, AORB_3, AORB_4, LOI_1, LOI_2, LOI_3, LOI_4

Figure 4: Mutant Execution

Step 3: Design of Test-cases, analysis of Mutation operator & Mutation Execution.

Input code:
gdMultClass.java

```
public class gdMultClass
{
      public static String gdMult(int a,int b)
      {
            String result = "";
            int c = a - b * b + a;
            result = "" + c;
            return result;
      }
 }
```

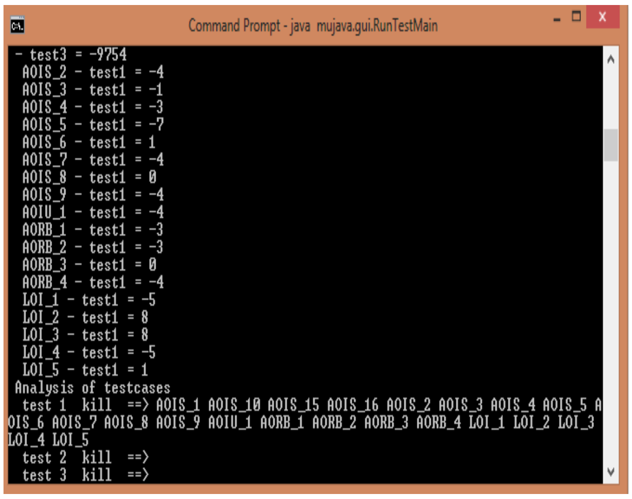
Test-case Design:

gdMultClassTest.java

```
public class gdMultClassTest
{
public String test1()
{
      String result = "";
      gdMultClass obj = new gdMultClass();
      result = obj.gdMult(1,2);
      return result;
}
public String test2()
{
      String result = "";
      gdMultClass obj = new gdMultClass();
      result = obj.gdMult(11,12);
      return result;
}
public String test3()
{
    String result = "";
    gdMultClass obj = new gdMultClass();
    result = obj.gdMult(123,100);
    return result;
}
}
```

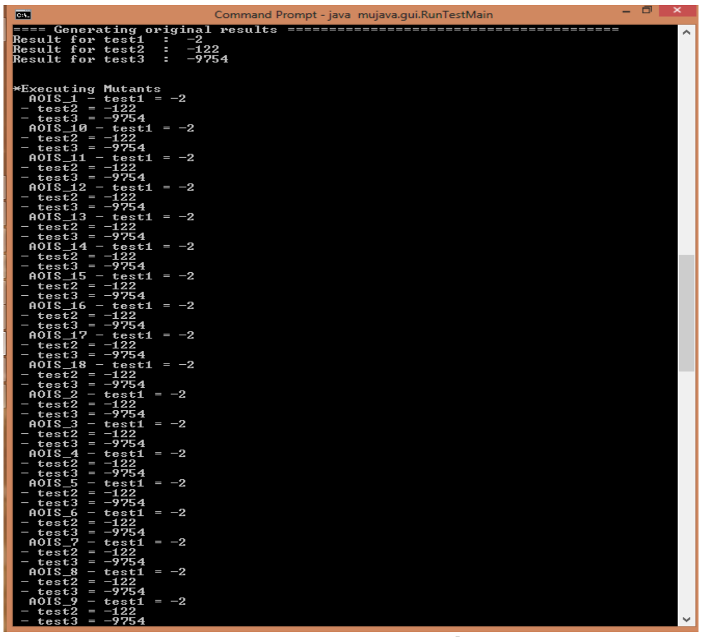
Mutation Log:

```
AORB_1:10:java.lang.String_gdMult(int,int):a - b * b + a
=> (a - b * b) * a
AORB_2:10:java.lang.String_gdMult(int,int):a - b * b + a
=> (a - b * b) / a
AORB_3:10:java.lang.String_gdMult(int,int):a - b * b + a
=> (a - b * b) % a
AORB_4:10:java.lang.String_gdMult(int,int):a - b * b + a
=> a - b * b - a
AOIU_1:10:java.lang.String_gdMult(int,int):a => -a
AOIS_1:10:java.lang.String_gdMult(int,int):a => ++a
AOIS_2:10:java.lang.String_gdMult(int,int):a => --a
AOIS_3:10:java.lang.String_gdMult(int,int):a => a++
AOIS_4:10:java.lang.String_gdMult(int,int):a => a--
AOIS_5:10:java.lang.String_gdMult(int,int):b => ++b
AOIS_6:10:java.lang.String_gdMult(int,int):b => --b
AOIS_7:10:java.lang.String_gdMult(int,int):b => b++
AOIS_8:10:java.lang.String_gdMult(int,int):b => b--
AOIS_9:10:java.lang.String_gdMult(int,int):b => ++b
AOIS_10:10:java.lang.String_gdMult(int,int):b => --b
AOIS_11:10:java.lang.String_gdMult(int,int):b => b++
AOIS_12:10:java.lang.String_gdMult(int,int):b => b--
AOIS_13:10:java.lang.String_gdMult(int,int):a => a++
AOIS_14:10:java.lang.String_gdMult(int,int):a => a--
AOIS_15:11:java.lang.String_gdMult(int,int):c => ++c
AOIS_16:11:java.lang.String_gdMult(int,int):c => --c
AOIS_17:11:java.lang.String_gdMult(int,int):c => c++
AOIS_18:11:java.lang.String_gdMult(int,int):c => c--
LOI_1:10:java.lang.String_gdMult(int,int):a => ~a
LOI_2:10:java.lang.String_gdMult(int,int):b => ~b
LOI_3:10:java.lang.String_gdMult(int,int):b => ~b
LOI_4:10:java.lang.String_gdMult(int,int):a => ~a
LOI_5:11:java.lang.String_gdMult(int,int):c => ~c
```



```
 - test3 = -9754
  AOIS_2 - test1 = -4
  AOIS_3 - test1 = -1
  AOIS_4 - test1 = -3
  AOIS_5 - test1 = -7
  AOIS_6 - test1 = 1
  AOIS_7 - test1 = -4
  AOIS_8 - test1 = 0
  AOIS_9 - test1 = -4
  AOIU_1 - test1 = -4
  AORB_1 - test1 = -3
  AORB_2 - test1 = -3
  AORB_3 - test1 = 0
  AORB_4 - test1 = -4
  LOI_1 - test1 = -5
  LOI_2 - test1 = 8
  LOI_3 - test1 = 8
  LOI_4 - test1 = -5
  LOI_5 - test1 = 1
 Analysis of testcases
  test 1  kill  ==> AOIS_1 AOIS_10 AOIS_15 AOIS_16 AOIS_2 AOIS_3 AOIS_4 AOIS_5 A
OIS_6 AOIS_7 AOIS_8 AOIS_9 AOIU_1 AORB_1 AORB_2 AORB_3 AORB_4 LOI_1 LOI_2 LOI_3
LOI_4 LOI_5
  test 2  kill  ==>
  test 3  kill  ==>
```

Figure 5: Mutant Execution Result Analysis.



```
==== Generating original results =========================================
Result for test1   :   -2
Result for test2   :   -122
Result for test3   :   -9754


*Executing Mutants
  AOIS_1 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_10 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_11 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_12 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_13 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_14 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_15 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_16 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_17 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_18 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_2 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_3 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_4 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_5 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_6 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_7 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_8 - test1 = -2
 - test2 = -122
 - test3 = -9754
  AOIS_9 - test1 = -2
 - test2 = -122
 - test3 = -9754
```

Figure 6: Mutation Operator Analysis.

VIII. CONCLUSION

In this paper, proposed work has applied various mutation operators like arithmetic, logical, conditional and relational operator. It has been observed that arithmetic operator is having high probability to generate equivalent mutant than conditional, logical and relational operator. Hence there is need to focus more on arithmetic operator. By using proposed algorithm, in some cases of arithmetic operator, mutation score efficiency is increased.
Future scope is to execute mutants in parallel by using nVIDIA processors to increase the performance of computational time.

REFERENCES

[1] Yue Jia & Mark Harman, "An analysis and survey of the Development of Mutation Testing", *IEEE transactions on software engineering, vol. 37, no. 5, september/october 2011*

[2] Mustafa Bozkurt, Mark Harman and Youssef Hassoun "Testing and verification in service oriented architecture: a survey" *software testing, verification and reliability Softw. Test. Verif. Reliab. (2012) Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1470*

[3] Pedro Reales Mateo, Macario Polo Usaola. "Bacterio: Java Mutation Testing Tool" *2012 28th IEEE International Conference on Software Maintenance (ICSM)*

[4] Pedro Reales Mateo, Macario Polo Usaola, and Jose Luis Fernandez Aleman. "Validating Second-Order Mutation at System Level." *IEEE Transactions on Software Engineering, VOL. 39, no. 4, april 2013.*

[5] Lin Deng, Jeff Offutt, Nan Li" Empirical Evaluation of the Statement Deletion Mutation Operator" *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), March 2013*

[6] Mujava tool available through Website: http://cs.gmu.edu/~offut/mujava/.