Code measure Automation Through Mutation Testing

Pritam Rani
M.Tech. Scholar, Dept. of CSE
SGI, Gurgaon.

Dr. Preeti Gera
Associate Professor, Dept. of CSE
SGI, Gurgaon, India.

Abstract- Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support. Mutation testing does not take a path-based approach. Instead, it takes the program and creates many mutants of it, by making simple changes to the program. The goal of testing is to make sure that during the course of testing; each mutant produces an output different from the output of the original program. We have designed a Tool to work Mutation Testing. The mutation testing is performed in terms of some substitutions in terms of operators. In this proposed work we have considered java as the base language to test the code. As the language is object oriented language. In this proposed work we have worked with two major types of operators. One is traditional operators and other is class Operators.

Keywords- Software Testing, Mutation Testing, Object-Oriented Programs.

I. INTRODUCTION

MUTATION TESTING EXAMPLE

Mutation testing of a program P proceeds as follows. First, a set of test cases T is prepared by the tester, and P is tested by the set of test cases in T. If P fails, then T reveals some errors, and they are corrected. If P does not fail during testing by T, then it could mean that either the program P is correct or that P is not correct but T is not sensitive enough to detect the faults in P. To rule out the latter possibility (and therefore, to claim that the confidence in P is high), the sensitivity of T is evaluated through mutation testing and more test cases are added to T until the set is considered sensitive enough for "most" faults. So, if P does not fail on T, the following steps are performed [8].

1. Generate mutants for P. Suppose there are N mutants.
2. By executing each mutant and P on each test case in T, find how many mutants can be distinguished by T. Let D be the number of mutants that are distinguished; such mutants are called dead.
3. For each mutant that cannot be distinguished by T (called a live mutant), find out which of them are equivalent to P. That is, determine the mutants that will always produce the same output as P. Let E be the number of equivalent mutants.
4. The mutation score is computed as D / (N -E).
5. Add more test cases to T and continue testing until the mutation score is 1.

In this approach, for the mutants that have not been distinguished by T, their equivalence with P has to be determined. As determining the equivalence of two programs is un-decidable, this cannot be done algorithmically and will have to be done manually (tools can be used to aid the process). There are many situations where this can be determined easily. For example, if a condition x <= 0 (in a program to compute the absolute value, say) is changed to x < 0, we can see immediately that the mutant produced through this change will be equivalent to the original program P, as it does not matter which path the program takes when the value of x is 0. In other situations, it may be very hard to determine equivalence. One thing is clear: the tester will have to compare P with all the live mutants to determine which are equivalent to P. This analysis can, then, be used to add further test cases to T, in an attempt to kill those live mutants that are not equivalent[1].

Determining test cases to distinguish mutants from the original program is also not easy. In an attempt to form a test case to kill a mutant, a tester will have to examine the mutant (and the original program) and then, reason which test case is likely to distinguish the mutant. This can be a complex exercise, depending on the complexity of the program being tested and the exact nature of the difference between the mutant and the original program. Suppose that a statement at line / of the program P has been mutated to produce the mutant M. The first property that a test case T needs to have to distinguish M and P is that the test case should force the execution to reach the statement at /. Clearly, without this, M and P will not behave differently. The test case T should also be such that after execution of the statement at /, different states are reached by P and M. Before reaching /, the state while executing the programs P and M will be the same as the programs are same until/. If the test case is such that after executing the statement at /, the execution of the programs P and M either takes a different path or the values in the state are different, then there is a possibility that this difference will be manifested in output being different. If the state after executing the statement at / continues to be the same in P and M, we will not be able to distinguish P and M. Finally, T should be such that when P and M terminate, their states are different (assuming that P and M output their complete state at the end only).

MUTATION OPERATORS

In this presented work we are dealing two broad categories of the mutants

1. Traditional Operators
2. Class Method Operators

Here the Traditional operators represent all the method operators. These operators include Arithmetic operators, Conditional operators, Logical operators etc. This category includes both the unary and the binary operators. Another categorization is the class operators. The class operators represent the operators used in object oriented programs. These operators include the operators related to polymorphism, inheritance etc. Here in given table all the operators are defined

TRADITIONAL OPERATORS

To investigate the presence of some relationships among the set of test cases that kill the relational operator mutants, we looked at three Java programs. All the three programs were comprised of a sequence of Class operator based statements it basically includes the arithmetic operators, Relational Operators and Logical Operators. The bodies of the conditional statements were assignment statements that decided the result returned by the program. The programs accepted two or more integer inputs, went through a series of decision statements and came up with a single integer value as its result. The operational profiles of all the three programs were closed sets of integer combinations.

In each program, the faults were introduced one at a time. Since our primary aim was to study the nature of relational operators, the induced faults were confined to the body of the conditional statements, while the guard statements were not changed. Test cases in the operational profile that could find the fault in the program were identified. For each faulty version of the program, five mutants were generated by replacing the relational operator, Arithmetic operators and logical operators in the conditional statement that had the fault in its body, with all other possible relational operators. The relational operators considered were '<', '< =' '>', '> =', '= =' and '! =', Arithmetic operators include +,-,*, /, % and Logical Operators includes &&, ||, Corresponding to each mutant a set of test cases was developed that consisted of test cases that killed the mutant. For each set, the percentage of test cases in them that found the fault in the original program was noted.

The study revealed that definite relationships exist between the test suites that kill the relational operator mutants. These relationships when depicted in the form of Venn diagrams looked similar for all the mutants. Each oval in the Venn diagram represent the test cases that kill the corresponding mutant. The Venn diagram describing the relationship between various relational operator mutants formed by replacing '<' in a conditional statement is shown in figure. The Venn diagram looked similar for other relational operators with the test suites changing positions. From the Venn diagram it can be seen that there are a couple of inclusion relationships between the test suites that kills mutants, which implicitly means that we could have potentially removed some of the mutants without effecting the performance of the test suites resulting from mutation analysis.

Table – 2 Details of Table Studied

No	Program	Description	Number of Inputs
1	diff	Computes an integer Based on the Difference of inputs.	2
2	triangle	Accepts the dimensions of three sides of a triangle and determines its type	3
3	sum	Computes an integer based on the sum of the inputs.	4

CLASS LEVEL MUTATION OPERATORS

This will briefly describe the class-level operators implemented in MuJava. The operators considered for MuJava have been divided in four categories according to their usage in Object Oriented programming. The first 3 groups target features common to all OO languages. The last group includes features that are specific to Java. These groups are:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Java-specific Features

Here the Vann Diagram of proposed operator substitution is given

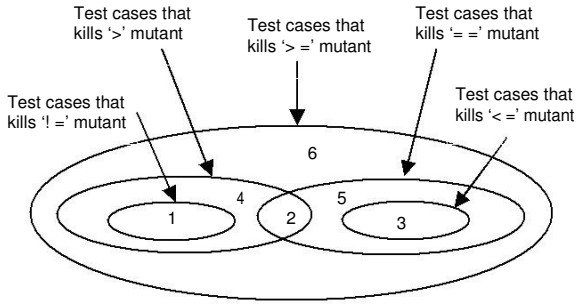


Figure 1. Venn diagram showing relationship between test suites that kill relational operator mutants generated by replacing '<' operato

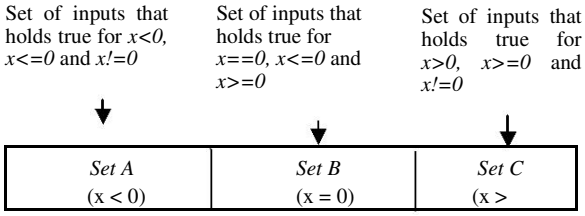


Figure 2. Inherent relationship between relational operators

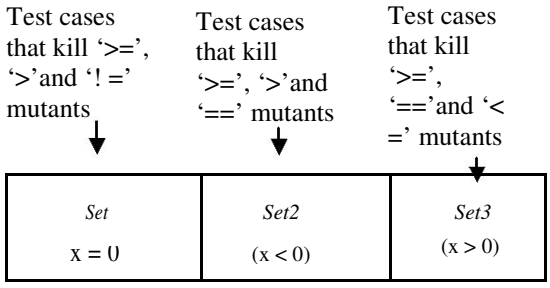


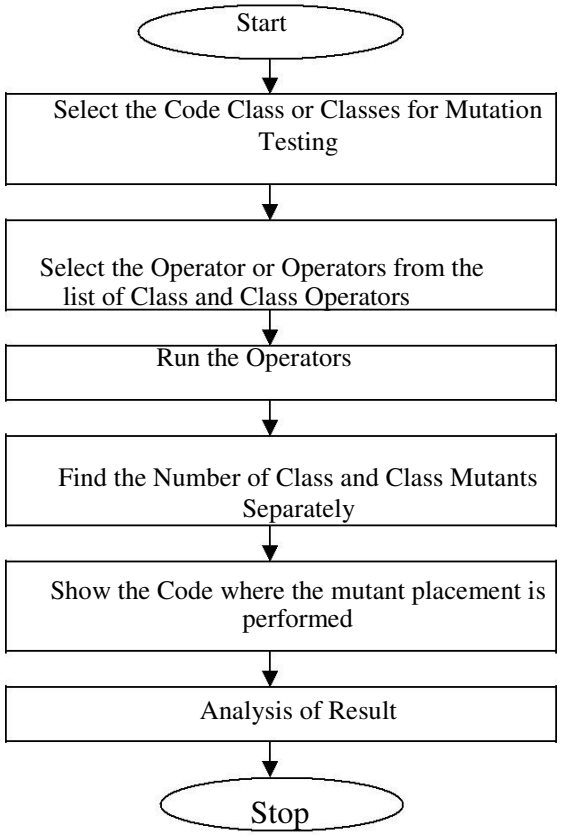
Figure 3. Redrawn Venn diagram from Fig.1

Table - 3 Classification of test cases based on mutants killed

Mutated operator	Mutants killed by test cases in the set		
	Set1	***Set2***	***Set3***
<	>=, >, !=	>=, >, ==	>=, ==, <=
<=	>, >=, ==	>, >=, !=	>, !=, <
>	<=, ==, >=	<=, ==, <	<=, <, !=
>=	<, <=, ==	<, <=, !=	<, !=, >
==	!=, <, <=	!=, <, >	!=, >, >=
!=	==, >=, >	==, >=, <=	==, <=, <

II. RESULTS AND ANALYSIS

We have designed a Tool to work Mutation Testing. The mutation testing is performed in terms of some substitutions in terms of operators. In this proposed work we have considered java as the base language to test the code. As the language is object oriented language. In this proposed work we have worked with two major types of operators. One is Class operators and other is Class method Operators.



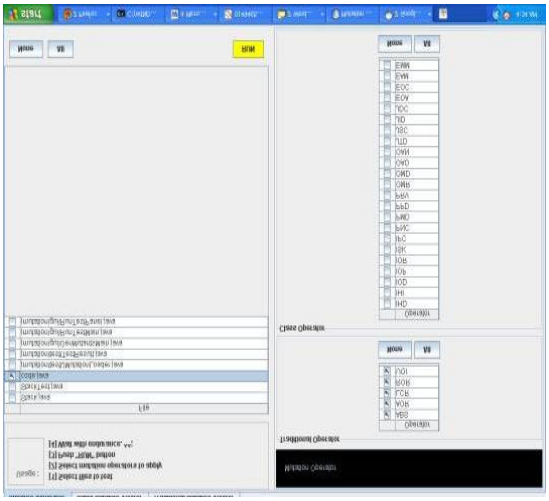


Figure 4. Test Case Generation

```
public class code
{
  public int calc(int z)
     {
                int    result;
                result  =  1;
                if(z > -10)
                        result = 2;
                if(z > 0)
                        result = 3;
                if(z > 10)
                        result = 4;
                return result;
     }
     public int triangle(int a,int b,int c)
      {
                int result;
                result = 1;             // scalene
                if(a == b)              // isosceles
                        result = 2;
                if(b == c)
                        result = 2;
                if(a == c)
                        result = 2;
                if(a == b) // equilateral if(a ==
                        c)
                        result = 3;
                if(a >= b+c) // not a triangle
                        result = 4;
                if(b >= a+c)
                        result = 4;
                if(c >= b+a)
                        result = 4;
                if(a <= 0) // bad input result =
                        5;
                if(b <= 0)
                        result = 5;
                if(c <= 0)
                        result = 5;
          return 1;
        }
    public int calcfour(int a, int b, int c, int d)
       {
          int result,z;
             z = a + b + c + d;

               if(z <0 && b>0)
                        result = 1;
                 if(z <-10 || a<0)
                        result = 2;
```

```
            if(z >=0)
                    result = 3;
            if(z >10)
                    result = 4;
            return result;
        }
    }
```

The above defined code is tested using mutation Class test operators.

Figure 5. Class and Operator Selection Window for Mutation Generation

Here we presented the work performed by taking some example code. We are considering the Class methods operators and the class operators separately. The complete working of the presented tool is given in the form of a flow chart presented here. First Step of the proposed work is defined in terms of Mutant Generation. In very first screen of the proposed work we have a user friendly environment with following properties. User can select one or All Classes on which the mutation operation will be performed. The selection will be performed by using checked list box.

Once run is performed. The internal process includes the substitutions of specified operators one by one in the source code. To view the operator or the mutant generated code we have separate tab for both the Class operators and class operators. In Fig. 5.3 the code view after mutation substitution is shown.

- The Class mutants view is given in Fig. 4.2
- It also finds the number of mutants in the code. Here in the presented example we only analyzed the Class operators.
- As we can see in Class Mutant View Fig. the red color source code represents the code segment where the substitution of operator is place.
- As we can see in the given source codes we have total 362 mutants. As the graph depicts the code have maximum UOI type of Class operator and the lowest is of LCR type.

The overall analysis of the Class operators is given in table 5.1

Operator	Number of Mutants
ABS	84
AOR	24
LCR	4
ROR	100
UOI	148

x The Fig. depicts OMR operator is having the highest mutant count.

The overall analysis of the class operators is given in table 5.2

Table – 5.2 Mutant Summary

Operator	Number of Mutants
IHD	0
IHI	0
IOD	1
IOP	0
IOR	1
ISK	0
IPC	0
PNC	0
PMD	0
PPD	0
PRV	3
OMR	0
OMD	0
OAO	0
OAN	0
JTD	0
JSC	0
JID	1
JOC	1
EOA	0
EOC	0
EAM	0
EMM	0

- As we can see in above table the class operators are shown with respective mutant count. The highest number of mutants is of type PRV. The analysis graph of class mutants is given in Fig.
- As have defined a smaller class with few functions. It has less number of class based mutants.
- Once the mutant generation code is complete the next work is to analyze the mutant classes.
- In this graphical user we accept the Mutant class and the respective test case accept from the user as the direct input.
- Here we have to select the mutant class and execute it. As the mutant class will be executed it will show the number of mutants it contains along with life mutants.
- It means the mutants are further categorizing as the live mutants or dead mutants.

CLASS LEVEL MUTANTS

To work with Class level mutants we have taken a class hierarchy such that:

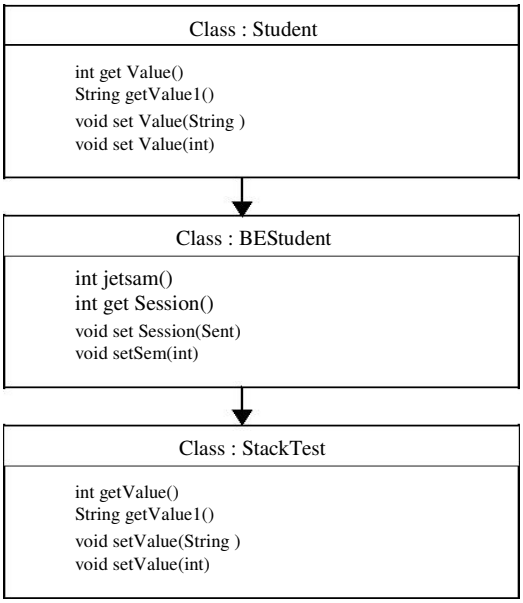


Figure 6. Test Class Map

- As the mutants generated it will store the each mutant code in a specific folder called “Result”.
- We can directly use the code from the result folder or we can use the other GUI to process on these mutants. The specified graph screen is presented by using Test Case Runner.

As we can see we have 3 classes in a multi level inheritance format. Student class is having 4 functions out of which two are performing overloading. Now Student class is inherited to BE Student Class and that is further inherited to Stack Test Class; we try to identify all the mutants in this file and got the following list of mutants in it.

Table 6 : Class Mutant Count

Operator	Number of Mutants
IHD	0
IHI	2
IOD	0
IOP	0
IOR	0
ISK	0
IPC	0
PNC	0
PMD	0
PPD	0
PRV	2
OMR	0
OMD	0
OAO	0
OAN	0
JTD	4
JSC	0
JID	0
JOC	0
EOA	0
EOC	0
EAM	0
EMM	0

As we can see the table only 3 kind of mutants find. It includes 2 instances of IHI mutant, 2more instances of PRV mutant and 4 instances of JTD.

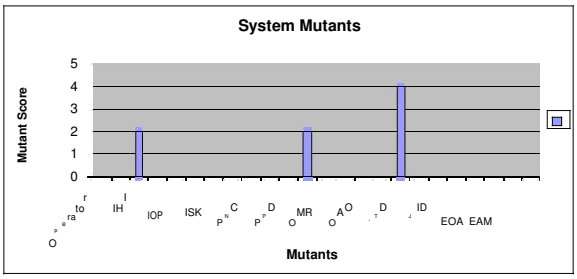


Figure 7. Mutant Score

An Analysis of the Mutants killed/alive

Test cases were created for each type of operates. When these test cases were executed against the mutants, the following results were obtained.
As we can the case Class operators we have total 10 mutants and all are killed by now. The mutant score here is 100%. The representation is given in the form a graph shown in Fig.8

Table 5.3

	Total Mutants	Killed Mutants	Live Mutants	Mutant Score
Class Operators	10	10	0	100%

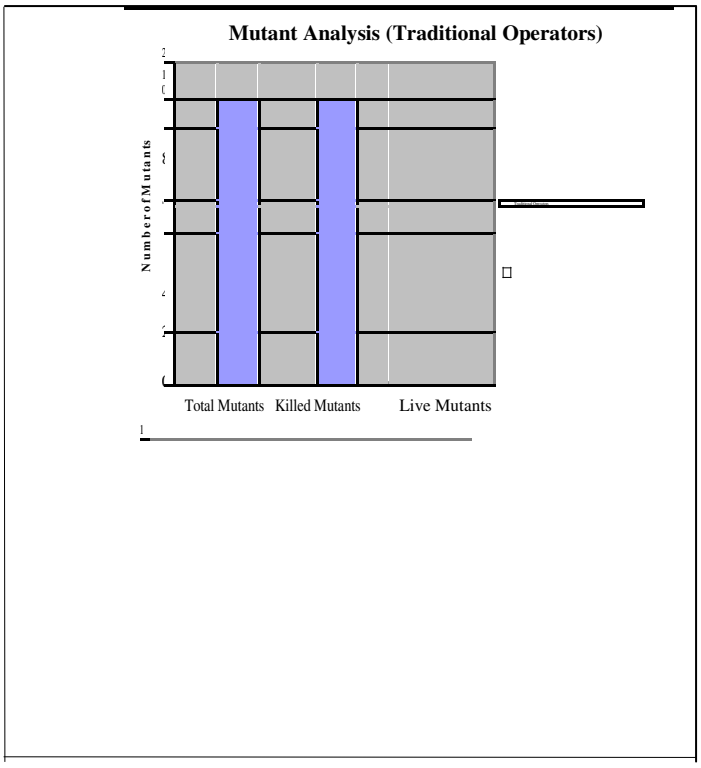


Figure 8. Mutation Analysis (Killed Vs. Total)

In case of Class operators the total mutant score respective to the given program code is represented in the form of Table 5.4.

Table 5.4

	Total Mutants	Killed Mutants	Live Mutants	Mutant Score
Class Operators	7	7	0	100%

As we can the case Class operators we have total 10 mutants and all are killed by now. The mutant score here is 100%.

III. CONCLUSION

The mutation testing is about to analyze the software code respective to the different operators. In this present work we have defined work with two types of operators called conventional operators and object oriented operators. We have designed a java based tool to automate the mutation testing with respect to different operators. The tool will first generate the mutants based code by performing the relative substitution of operators. Once the mutants generated, the next work is to perform the analysis in terms of live and dead mutants. The results show the successful implementation of vast range of conventional and object oriented operators.

In this present work we have automate most of available conventional and object oriented operators. The work can be extended in different direction. We can perform the mutation testing on other kind of operator relative to aspect oriented programming, component based programming etc. In same way the work is implemented for java based operators, the work can be extended to perform same work on other languages.

REFERENCES

[1] Ronald Finkbine, **"Usage Of Mutation Testing As A Measure Of Test Suite Robustness",** 0-7803-7844-X/03@2003 IEEE

[2] Dr. Richard Carver, **"Mutation-Based Testing Of Concurrent Programs",** INTERNATIONAL TEST CONFERENCE 1993 0-7803-14 29-8/93 @ 1993 IEEE

[3] **A.** Jefferson Offutt, **"A Practical Class For Mutation Testing: Help For The Common Programmer",** INTERNATIONAL TEST CONFERENCE 1994 0-7803-21 02-2/94@ 1 994 IEEE

[4] Hoijin Yoon, **"Mutation-based Inter-class Testing",** 0-8186-9183-2/98@1998 IEEE

[5] Simone do Rocio Senger de Souza,**"Mutation Testing Applied to Estelle Specifications",** Proceedings of the 33rd Hawaii International Conference on Class Sciences – 2000 0-7695-0493-0/00(c) 2000 IEEE

[6] Tafline Murnane**, "On the Effectiveness of Mutation Analysis as a Black Box Testing Technique",** 0-7695-1254-2/01@ 2001 IEEE

[7] K. K. Mishra, **"An Approach for Mutation Testing Using Elitist Genetic Algorithm"**, 978-1-4244-5540-9/10©2010 IEEE

[8] Ying Jiang**, "Contract-Based Mutation for Testing Components",** Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05) 063-6773/05© 2005 IEEE

[10] Prasanth Anbalagan, **"Efficient Mutant Generation for Mutation Testing of Point cuts in Aspect-Oriented Programs",** Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)(MUTATION'06) 0-7695-2897-x/06© 2006 IEEE

[11] Jeff Offutt**, "Mutation Testing Implements Grammar-Based Testing",** Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)(MUTATION'06) 0-7695-2897-x/06© 2006 IEEE

[12] Robert M. Hierons, **"Mutation Testing from Probabilistic Finite State Machines**", Testing: Academic and Industrial Conference - Practice And Research Techniques 0-7695-2984-4/07© 2007 IEEE

[13] Shufang Lee, **"Automatic Mutation Testing and Simulation on OWL-S Specified Web Services",** 41st Annual Simulation Symposium 1080-241X/08© 2008 IEEE

[14] Fabiano Cutigi Ferrari, **"Mutation Testing for Aspect-Oriented Programs",** 2008 International Conference on Software Testing, Verification, and Validation
0-7695-3127-X/08© 2008 IEEE

[15] Garrett Kaminski, **"Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing",** 2009 International Conference on Software Testing Verification and Validation 978-0-7695-3601-9/09© 2009 IEEE

[16] Chixiang Zhou**, "Mutation Testing for Java Database Applications",** 2009 International Conference on Software Testing Verification and Validation 978-0-7695-3601-9/09© 2009 IEEE

[17] Jin-hua Li, **"Mutation Analysis for Testing Finite State Machines",** 2009 Second International Symposium on Electronic Commerce and Security 978-0-7695-3643-9/09© 2009 IEEE

[18] Mike Papadakis, **"An Effective Path Selection Strategy for Mutation Testing",** 2009 16th Asia-Pacific Software Engineering Conference 1530-1362/09© 2009 IEEE

[19] William B. Langdon**, "Multi Objective Higher Order Mutation Testing with Genetic Programming",** 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques 978-0-7695-3820-4/09© 2009 IEEE