Volume 3, Issue 1, January 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Integration Testing of Object Oriented Component Using Program Slicing in the Detection of Equivalent Mutants

Upasana Shukla
Dept. Of CSE
GCET, Gr. Noida
India

Prof. Ajeet Kumar Bharati
Dept. Of CSE
GCET, Gr. Noida
India

Deena Nath Gupta
Dept. Of CSE
SIET, Gr. Noida
India

Abstract -- Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are design to represent. Therefore the quality of the mutation operator is key to mutation testing. 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. The proposed model is a hybrid of above two operators. With the help of proposed model, we can find equivalent mutant as well as kill mutant. The main advantage of using this model is that it works very efficient and takes less time too. By killing mutant efficiently, the program enhances very much.

Keywords – Mutation, Operator, Testing, Kill mutant, Generation.

1. INTRODUCTION

Mutation testing is a fault based testing technique firstly proposed in 1978. It measures the effectiveness of test cases. Mutation testing is based on the concept of fault seeding in a program that will be completely tested and all faults are detected and removed. The set of faults which are introduced in a program are called mutants. A test input distinguishes two programs if the programs produce different output for this input. Mutation testing is based around a simple notion: if a test set is good at distinguishing our program from other similar programs then it is likely to be good at finding faults. The justification for this is that if the program is faulty, then testing may be seen as trying to distinguish the faulty program from some (slightly different, correct) version of the program.
Generally speaking, mutation analysis uses well defined rules (called *mutation operators*) that are defined on syntactic structures (such as grammars or program source) to make systematic changes to the syntax or to the objects developed from the syntax [1, 2].
More precisely, given a program p, a *mutant* is some variant p' of p. typically we generate mutants by applying *mutation operators*: rules that allow us to transform programs. For example, the rule that says "replace the occurrence of the arithmetic operator + by the arithmetic operator *" is a mutation operator. Another example is replacing > by ≥ in a predicate. Normally we produce a mutant by applying one instance of one operator only: such mutants are called first-*order mutants*.
Typically for testing, only first order mutants are considered. If we apply a mutation operator to a mutant, we generate a mutant of a mutant. This is called a second order mutant. If we mutate a second order mutant, we obtain a third order mutant and so on. These "higher order" (i.e. higher than first order) mutants are not normally considered in Mutation Testing.
Generally speaking, mutation analysis uses well defined rules (called *mutation operators*) that are defined on
syntactic structures (such as grammars or program source) to make systematic changes to the syntax or to the objects developed from the syntax. When applied to programs, mutation analysis makes systematic changes to the program, then asks the tester to design inputs that cause the mutated program (*mutant*) to create output that is different from the original version of the program. We customarily think of these mutants as being faults, although it is possible that the original program was faulty and the mutant is correct, or the mutant has no affect on the functional behaviour of the program (*equivalent*). Most mutation systems introduce one change at a time (changing one terminal symbol in the grammar), although it is possible to make multiple changes (called *higher-order mutants*). This paper is focused on program mutation and assumes all mutants are single-order.

2. RELATED WORK

Research in mutation testing has a rich history and focuses on three kinds of activities such as defining mutation operators, experimentation and developing tools. The first one involves defining new mutation operators for different

languages. The second research activity is experimentation with mutations. Empirical studies have supported the effectiveness of mutation testing. Mutation testing has been found to be more powerful than statement and branch coverage and more effective in finding faults than data flow. Offutt et al. and Wong and Mathur evaluated the idea of selective mutation which identifies the critical mutation operators that provide almost the same testing coverage as non-selective mutation. Using this approach considerably decreases the number of mutants generated and thus reduces computational cost. The third kind of activities in mutation testing research is developing mutation tools. Mothra [30] and Proteum [31] were developed for Fortran and
C, respectively. Jester [32], Jumble [33] and MuJava [8, 34, 35], as well as the tool presented in this paper, are dedicated to the Java language. An important characteristic of mutation testing tools are the mutation operators supported by a tool. Here the work has been done with two types of mutation operators : (1) traditional mutation operators adapted from procedural languages and (2) OO (or class) mutation operators developed to handle specific features in OO programming.

2.1 ALGORITHM

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, 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.
Mutation testing involves three steps:
(1) Mutant generation. Mutation operators are applied to the original program to get a set of mutants. A mutation operator is a rule that introduces a syntactic change in the original program.
(2) Mutant execution. Tests are executed against the original program and all the mutants.
(3) Result analysis. Results of the executions are analyzed and the mutation score is calculated (measuring of the
quality of the test suite). The result analysis step counts the number of killed mutants, which are mutants that have different behaviour from the original program for at least one test.
Several mutation tools have been built. Mothra is an early mutation tool and mutates Fortran functions. Mothra
generates mutants, executes test cases defined by the tester or generated automatically, and performs results analysis to calculate the mutation score.
Proteum performs unit-level mutation analysis for C programs. A tester selects a C function, then uses Proteum to generate mutants, execute test cases and calculate the mutation score. Mutation has also been applied in at the class level. Java tools include Jumble [19], Javalanche [20] .The first tool to bring mutation to the system level is Certitude. Certitude applies mutation analysis to systems on chips for hardware verification. The mutation operators are applied to C++ code and communication protocols of TML SystemC-based designs at the integration level.

3. PROPOSED WORK

To develop the good quality software high quality testing is required. More than 60% of software has been failed due to less testing. Software cannot be 100% test. There is a great role of mutation testing for selecting the test sets. Mutation testing is used to measure, how “good” our tests are by inserting faults into the original program under test. Each fault inserted in the program generates a new program that is called a mutant that is slightly different from the original. In case of Mutation Testing we perform the code coverage with some common substitution is the existing source code. Same kind of work is presented in this proposed work. The work is about to automate the mutation process with the definition of functional level and class level operators. The work is also to analyze the fault rate by analyzing the successful substitutions of the different operators in the work. These separate substitutions are call mutants. The mutations that give the favorable results are called live mutants and the unfavorable are called Dead mutants. We have to analyze the live mutant's ratio. Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are design to represent. Therefore the quality of the mutation operator is key to mutation testing. 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. The proposed model is a hybrid of above two operators. With the help of proposed model, we can find equivalent mutant as well as kill mutant. The main advantage of using this model is that it works very efficient and takes less time too. By killing mutant efficiently, the program enhances very much.

3.1 MUTATION OPERATORS

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

1. Traditional Operators
2. Class Operators

Here the Class 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.

3.2 TRADITIONAL OPERATORS AND CLASS OPERATORS

To investigate the presence of some relationships among the set of test cases that kill the relational operator mutants, we use traditional operators. 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 set of test cases were also examined to study the relationships between them. While class operators 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:

- Encapsulation
- Inheritance
- Polymorphism
- Method Overloading
- Java-specific Features

Table – 1 All Operators for Mutation Testing

Arithmetic Operators	
AOR_B	**Arithmetic Operator Replacement**
AOR_U	**(binary) Arithmetic Operator**
AOI_U	**Arithmetic Operator Insertion**
AOI_S	**(unary) Arithmetic Operator**
AOD_U	**Arithmetic Operator Deletion**
AOD_S	**(unary) Arithmetic Operator**
Relational Operators	

ROR	**Relational Operators Replacement**
Conditional Operators	
COR	**Conditional Operator Replacement**
COI	**Conditional Operator Insertion**
COD	**Conditional Operator Deletion**
Shift Operators	
SOR	**Shift Operator Replacement**
Logical Operators	
LOR	**Logical Operator Replacement**
LOI	**Logical Operator Insertion**
LOD	**Logical Operator Deletion**
Assignment Operators	
ASRS	**Assignment Operator Replacement (short-cut)**
Encapsulation Operators	
AMC	**Access modifier change**
IHD	**Hiding variable deletion**
IHI	**Hiding variable insertion**
IOD	**Overriding method deletion**
IOP	**Overriding method calling position change**
IOR	**Overriding method rename**
ISI	***Super* keyword insertion**
ISD	***Super* keyword deletion**
Inheritance Operators	
IPC	**Explicit call to parent's constructor deletion**
PNC	**New method call with child class type**
PMD	**Member variable declaration with parent class type**
PPD	**Member variable declaration with child class type**
PCI	**Type cast operator insertion**
PCC	**Cast type change**
PCD	**Type cast operator deletion**
PRV	**Reference assignment with other compatible variable**
OMR	**Overloading method contents replace**
OMD	**Overloading method deletion**
Polymorphism Operators	

OAN	**Arguments of overloading method call change**
JTI	***this* keyword insertion**
JTD	***this* keyword deletion**
JSI	***static* modifier insertion**
JSD	***static* modifier deletion**
JID	**Member variable initialization deletion**
JDC	**Java-supported default constructor creation**
EOA	**Reference assignment & content assignment replacement**
EOC	**Reference comparison & content assignment replacement**
EAM	**Accessory method change**

The design architecture of this system is divided into two stages. In the stage 1 we generate the mutants of the original program and in the stage 2 we run the original program and mutants on the set of test cases, after that we find the list of killed mutants.

4. 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.

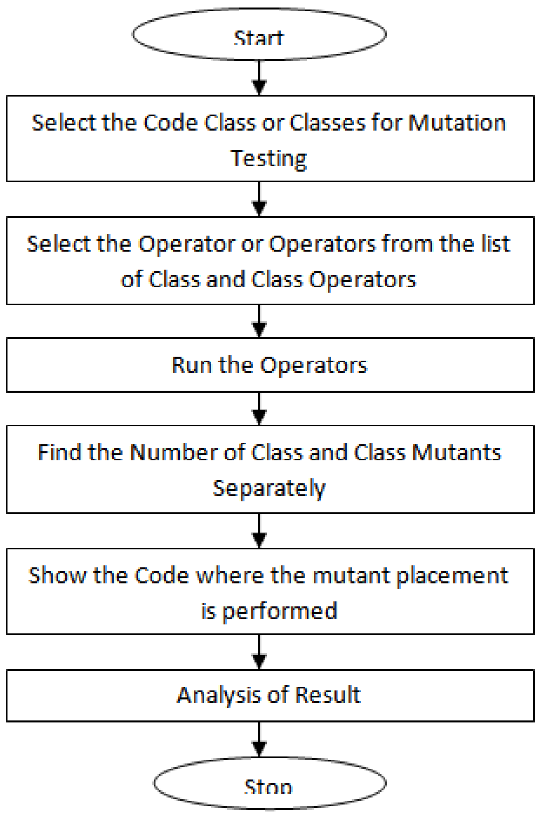
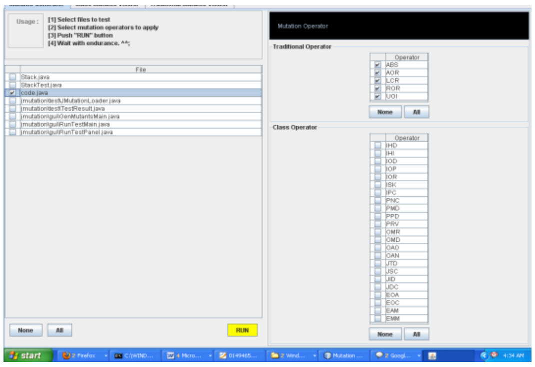

Start
Select the Code Class or Classes for Mutation Testing
Select the Operator or Operators from the list of Class and Class Operators
Run the Operators
Find the Number of Class and Class Mutants Separately
Show the Code where the mutant placement is performed
Analysis of Result
Stop

Figure1: Test Case Generation

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.



Usage : [1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

	File
☐	Stack.java
☐	StackTest.java
✓	code.java
☐	jmutation\test\JMutationLoader.java
☐	jmutation\test\TestResult.java
☐	jmutation\gui\GenMutantsMain.java
☐	jmutation\gui\RunTestMain.java
☐	jmutation\gui\RunTestPanel.java

Mutation Operator

Traditional Operator

	Operator
✓	ABS
✓	AOR
✓	LCR
✓	ROR
✓	UOI

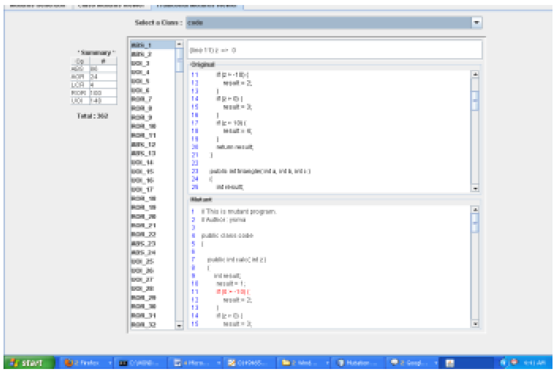
Class Operator

	Operator
☐	IHD
☐	IHI
☐	IOD
☐	IOP
☐	IOR
☐	ISK
☐	IPC
☐	PNC
☐	PMD
☐	PPD
☐	PRV
☐	OMR
☐	OMD
☐	OAO
☐	OAN
☐	JTD
☐	JSC
☐	JID
☐	JDC
☐	EOA
☐	EOC
☐	EAM
☐	EMM

Figure2. Class and Traditional operator Selection Window for Mutation Generation

Once the 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 Traditional operators and class operators. In Fig.1 and 2 the code view after mutation substitution is shown.



Select a Class : code

* Summary *

Op	#
ABS	86
AOR	24
LCR	4
ROR	100
UOI	148

Total : 362

ABS_1, ABS_2, UOI_3, UOI_4, UOI_5, UOI_6, ROR_7, ROR_8, ROR_9, ROR_10, ROR_11, ABS_12, ABS_13, UOI_14, UOI_15, UOI_16, UOI_17, ROR_18, ROR_19, ROR_20, ROR_21, ROR_22, ABS_23, ABS_24, UOI_25, UOI_26, UOI_27, UOI_28, ROR_29, ROR_30, ROR_31, ROR_32

(line 11) [illegible] => 0

Original

```
        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;
```

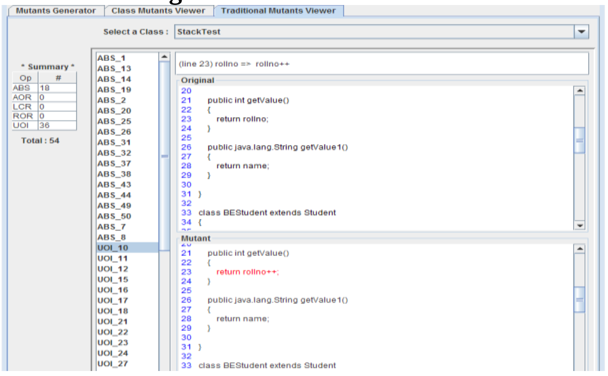
Mutant

```
// This is mutant program.
// Author : yuma

public class code
{

    public int calc( int z )
    {
        int result;
        result = 1;
        if (z > -10) {
            result = 2;
        }
        if (z < 0) {
            result = 3;
```

Fig.3: Class Mutant Viewer



Select a Class : StackTest

* Summary *

Op	#
ABS	18
AOR	0
LCR	0
ROR	0
UOI	36

Total : 54

ABS_1, ABS_13, ABS_14, ABS_19, ABS_2, ABS_20, ABS_25, ABS_26, ABS_31, ABS_32, ABS_37, ABS_38, ABS_43, ABS_44, ABS_49, ABS_50, ABS_7, ABS_8, UOI_10, UOI_11, UOI_12, UOI_15, UOI_16, UOI_17, UOI_18, UOI_21, UOI_22, UOI_23, UOI_24, UOI_27

(line 23) rollno => rollno++

Original

```

    public int getValue()
    {
        return rollno;
    }

    public java.lang.String getValue1()
    {
        return name;
    }

}

class BEStudent extends Student
{
```

Mutant

```
    public int getValue()
    {
        return rollno++;
    }

    public java.lang.String getValue1()
    {
        return name;
    }

}

class BEStudent extends Student
```

Fig.4: Traditional Mutant View

As we can see in above Fig. the red colours source code represents the code segment where the substitution of operator is place

The overall analysis of the Traditional operators is given in table1.

Table 1: Mutant Summary

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

The result of the table is presented in the form of graph.

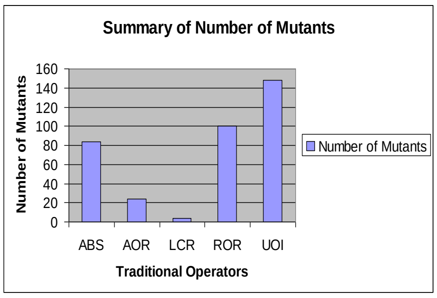


Fig.5: Summary of Total Traditional Mutants

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 Traditional operator and the lowest is of LCR type.

The overall analysis of the class operators is given in table2.

Table2: 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.

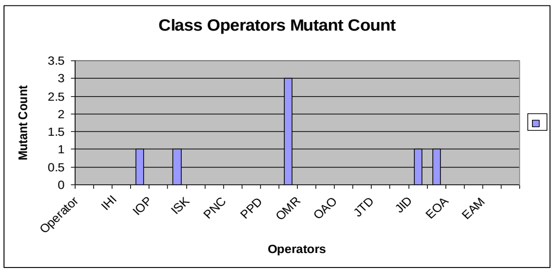


Fig.6: Class Mutants in Stack class

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

As the mutants generated it will store the each mutant code in a specific folder called “Result”.

Once the mutant generation code is complete the next work is to analyze the mutant classes.

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:

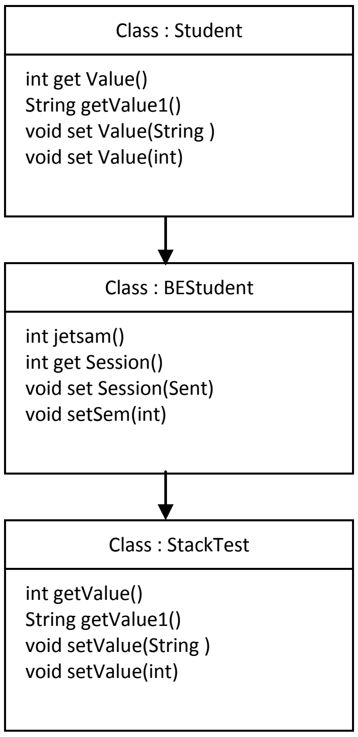


Figure7. Test Class Map

- 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 - 3 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.

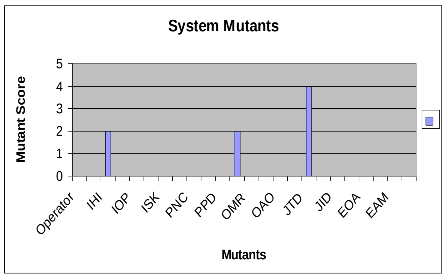


Figure8. Mutant Score

An Analysis of the Mutants killed/alive

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

Test cases were created for each type of operates. When these test cases were executed against the mutants, the following results were obtained.

As in case of traditional 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.9

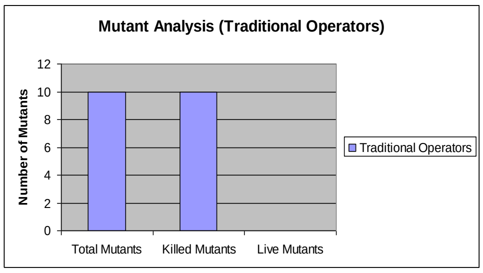


Figure9. Mutation Analysis (Killed Vs. Total for traditional operators)

In case of Class operators the total mutant score respective to the given program code is given as

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

As we can the case Class operators we have total 7 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.10

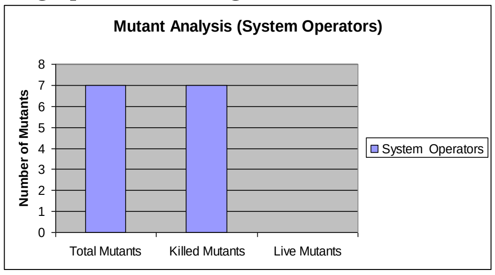


Figure10. Mutation Analysis (Killed Vs. Total for class operators)

5. 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] Sandra Camargo Pinto Ferraz Fabbri," **Mutation Testing Applied to Validate Specifications Based on Statecharts".**

[6] 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.

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

[8] K. K. Mishra,**" An Approach for Mutation Testing Using Elitist Genetic Algorithm"**, 78-1-4244-5540-9/10©2010 IEEE.

[9] Ying Jiang**," Contract-Based Mutation for Testing Components",** Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05© 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.