Volume 4, Issue 7, July 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Mutation Testing Using Mutation Operators for C# Programs

Pawar Sujata G.
IT Department & BVDU university
India

Idate Sonali R.
IT Department & BVDU university
India

Abstract: Software testing is the vital phase to develop program but developers can be simply neglect this phase for the reason that to finish work within restricted period. Generally developers complete product closer to the delivery period, hence they don't get sufficient time to create different test cases and test their program. The major difficulty in testing method is the creation of test cases that fulfill the criteria. Creating manual test cases is a tedious as well as cumbersome work for program developers in the last rush hours. The crucial activity of program design is testing. Mutation testing is a potent testing method to create various tests and evaluate the quality of program. In this paper, first the different types of mutation operators are explained. After coding, using mutation operator test cases can be generated from execution trace.

Keywords: Mutation testing, Mutation operators.

I. INTRODUCTION

Software testing is a vital phase of software development life cycle [SDLC].Software testing is a standard and powerful method of giving surety about software quality. Mutation testing is one of the methods of software testing to test software by applying mutation operators to original program. It is a very easy but inventive method used to test source program. The mutation testing concept was first developed by Richard Lipton in 1971 and since that time there is a huge growth in mutation testing.

Select a mutation operator and after choosing mutation operator, apply it to original program is called as mutation. Mutant is a simple program but having small change in source program by adding mutation operator to it.
Following is example of mutation testing:-

```
 int large()
{
if(s>p)
return(s);
else
return(p);
}
```

If values for s and p in one test set as s=1, p=2 then above original function returns value 2 which is actual output.
Five mutants created by replacing ">" operator by (<, <=,>=, =, < >) these operators in if statement of original function. For e.g. When you replace '>' operator by '<', then the mutant will be

```
int large()
{
If(s< p)
return(s);
else
return(p);
 }
```

If values for s and p in one test data set as s=1, p=2 then mutant returns value 1 which is different from actual output. Hence mutant is said to be dead or killed.
After executing each mutant, Result will be:-

Table: 1.1 Mutant examples and its Output

Sr. No.	Mutant	Result	Comparison
1	If s<p then	1	Dead
2	If s<=p then	1	Dead
3	If s>=p then	2	Live
4	If s=p then	2	Live
5	If s<>p then	1	Dead

After executing main program and mutant, result of both main program and mutant are different then the mutant is supposed to be dead or killed otherwise live. If the testing method has ability to find out change in main program and mutant, then test case is considered as adequate.

MUTATION OPERATORS:-

Mutation operators are of three types as given below

a) Statement level mutation operators
b) Method level mutation operators
c) Class level mutation operators.

These are explained in detail below.

a) Statement level mutation operators:-They include the generation of mutants to be successfully tested. There is a little difference between original program and mutant by one alteration and mutation is nothing but one modification in statement syntactically.

Operand Replacement Operators (ORO) – In this case, it changes a single operand by another constant or operand.
Expression Modification Operators (EMO) – In this case, it changes an operator or adding a new operator.

b) Method level mutation operators:-These are referred in integration as well as unit testing and these can be categorized in two types: 1) Inter-method 2) Intra-method

1) Inter-method:- In this type, when there is correlations between methods of one class then defects occur. In case of procedural language programs integration testing and testing procedures are comparable same.

2) Intra-method: - In this type when method is not implemented correctly then defects or faults arise. Conventional mutation operators for procedural programming will be sufficient as considered by researchers.

c) Class level mutation operators:-These are categorized into two types:
1) Inter class 2) Intra class

1) Inter class:-This testing occurs when multiple classes have to test in group and find defects in it. In this type of testing focuses the conventional integration testing and here most faults related to polymorphism, parent child relationship are found.

2) Intra class: - This testing occurs when there is a one class is to be tested only. In this case testing of whole class is done. Specialization of both i.e. the traditional unit and module testing is Intra-class testing. When they used, it checks the global methods of whole class. Within the class, tests are generally series of calls to methods.

Information Hiding:-

AMC: - Access Modifier Change
Mistakes occurred in object oriented programming in case of access control. Access levels like public, private always have poor access definitions and they do not create fault primarily but can reason defective behaviour while incorporating with additional classes. In case of C# language, there are four right visibility controls: - default, personal, public and protected. Access Modifier Change operator changes the right of entry level for methods as well as variables. For example suppose a field variable declared as protected access mode would have three mutants which are generated by AMC operator. In order prove truly that the variable is really protected, the test set should show difference between them when access modifier is public/private/default mode respectively.

For example:-
The original Code
```
 protected int b;
```
AMC Mutants:-
```
public int b;
private  int b;
int b;
```

Inheritance:-

The properties of parent class acquired by child class are called as inheritance. There are various types of inheritance such as single inheritance, multilevel inheritance, and multiple inheritances and so on. Improper use of inheritance leads to a different defects.

1. IHD: It means deletion of hiding variable
2. IHI: It means insertion of hiding variable
3. IOD: It means deletion of overriding method
4. IOP: It means calling position change of overriding method
5. IOR: It means renaming of overriding method

Live Mutant: -

The mutant which produces the similar result as main program and it cannot be destroyed then it is said to be equivalent mutant as well as live mutant. Sometimes our test case is not adequate means our test is unable to find difference among the main program and equal mutant because of same output; we can sort out problem by adding new test.

Killed/Dead Mutant: -

Choose any one mutation operator and apply to the main program for each part of the source code, Mutation testing is performed. Apply one mutation operator to the program, this small change in main program known as mutant program. When the test has ability to find out the change, then the mutant is considered as dead.

Mutation Testing brings a whole high level of fault-detection technique to the software developer.

II. LITERATURE SURVEY

Mateo projected operators of mutation which are linked with GUI components. The operators manage to analyze whether a component is exchanged with older version of similar component. It has a huge scope. This is a Graphical User Interface test automation tool an expansion which contains Graphical User Interface testing activities: creation, implementation and authentication. This tool is modified to create Graphical User Interface mutants with using the ability to implement these mutants dynamically and execute outcomes [1].

For unit level software testing, mutation testing is a method which is fault based. Weak mutation was projected as a mode which degrades the expenditure of mutation testing. Implementation of weak mutation shows output and we evaluated the usefulness versus the effectiveness of weak mutation. In addition to this we check several options.The most proper way to execute mutation which is a weak [2].

R. J. Lipton, R. A. DeMillo, and F. G. Sayward made team and they concluded that program tests that do not cover faults which are also helpful in case of uncovering complicated faults. The couple effect utilized during the testing process [3].

Mutation operator does not modify type and structural data declarations for conventional programming languages. The mutation operators developed for Java programs. The MuJava testing tool for Java programs are given[4] .
To demonstrate correctness for test data they consider two interpretations. They observe whether data adequate to express exactness exists for each interpretation and if it can be usually found and produced. They set the correlation in these questions and the problem of finding similarity between two programs [5].

Based on specified Java mutation operators, Derezi’nska presented C# specialized mutation operators set with extension and evaluated in a C# and developed mutation software known as CREAM. Derezi`nska and Szustek executed programs by using CREAM tool with group of C# mutation operators and displayed outcome [6].

Mutation technique based on traditional mutation operators is supported by several tools used for different programming languages. For creation and testing mutants for FORTRAN programs, the Mothra system used [7].

The FORTRAN program is translated to its intermediate form. Changes reflecting modification of a source code are introduced in that form. Execution of mutants is realized by interpretation of the program. Another tool is the Proteum system [8] used for a mutation analysis of C programs. Other mutation tools are Jester and Nester [9].

MuJava testing tool using Java mutation operators explained in detail. Automated analysis and testing tool MuJava displayed results by using mutation operators which are based on class level. Mutation operators which are based on class level updates object oriented programming language features such as parent child relationship, polymorphism etc. Here they provided several new operators which are based on class level and analysis of the mutant’s number created [10].

To increase the quality of testing, mutation testing is a best technique and gives guarantee about the software reliability. Test case creation using mutation testing approach proposed, hence after coding tests may be created. To create effective test cases, mutation testing implementation discussed by them in detail [11]. I have focused concept based on mutation testing using mutation operators

III. PROPOSED SYSTEM OVERVIEW

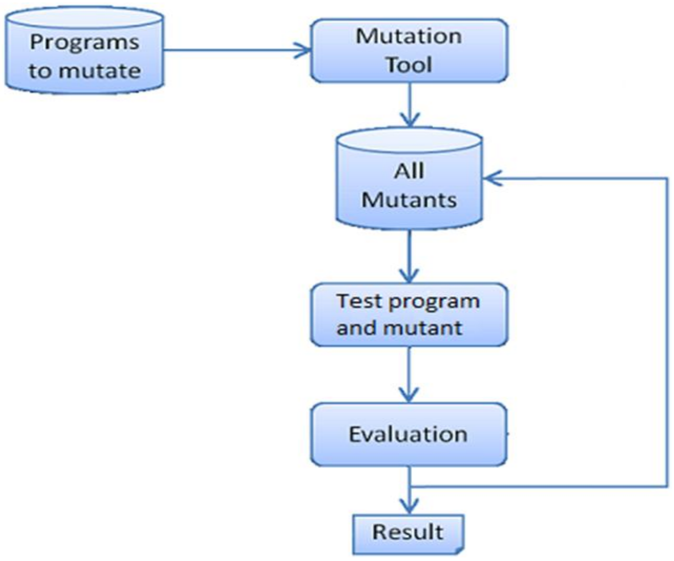


Figure1:-Proposed System

Mutation Testing Algorithm:

1. Accept mainl program as user input.

2. Run main program.

    i. If the result is wrong, the program should modify and test again.
    ii. If the result is correct as same as expected result then go to step 3.

3. Create mutant of main program by selecting code specific operators such as EMO, ORO etc

4. When mutant result is different from main program result, the mutant is supposed to be wrong then test is said to be successfully run.

5. Two types of mutants live:
    i) Functionally equal to the main program cannot be destroyed.
    ii) Able to kill but tests are not sufficient to destroy the mutant.

IV. SYSTEM ARCHITECTURE

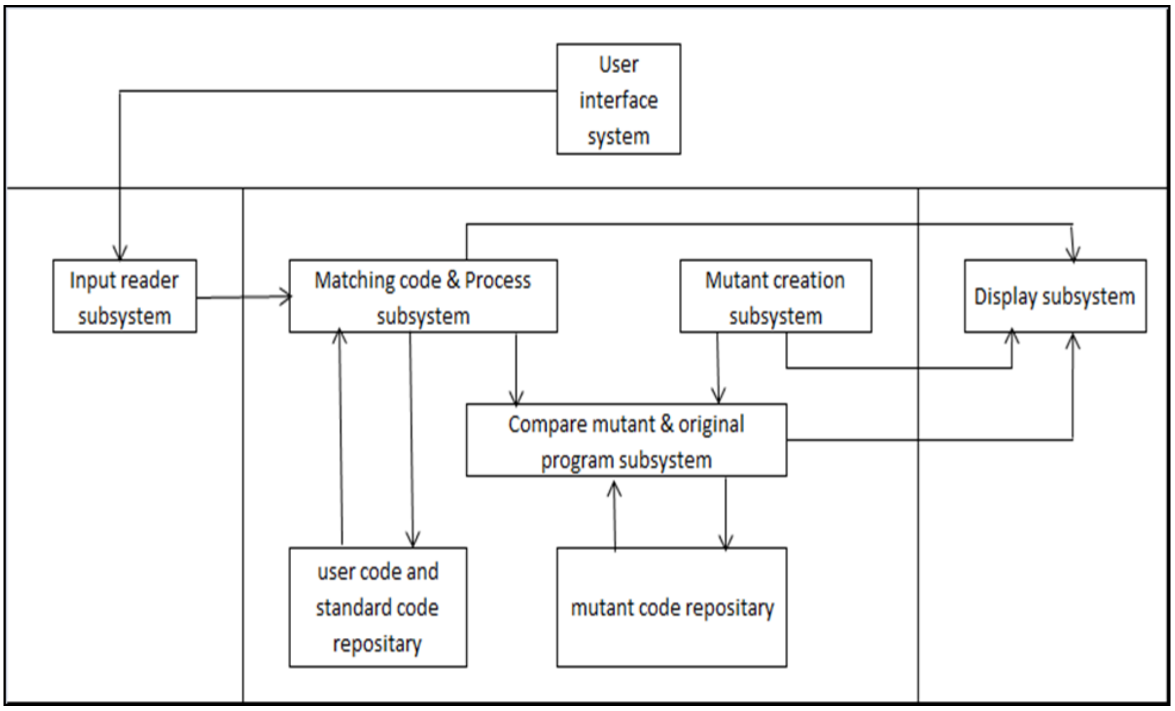

User interface system
Input reader subsystem
Matching code & Process subsystem
Mutant creation subsystem
Display subsystem
Compare mutant & original program subsystem
user code and standard code repositary
mutant code repositary

Figure2:system architecture diagram

In general, this system generally provides a user interface for communicating with the user. It accepts the input file from the user and compares with standard file displays the results. In next part It compares user input file, apply mutation operator to it called mutant. Then display result and compare main program result with mutant result to determine whether main program is correct or not. Our system works, in five steps:

1. **User interface system:** The user gives an input as input file for the system.
2. **Input reader subsystem:** The system accepts input file and browse standard file.
3. **Matching code and process subsystem:** This system matches input file code and standard code.
4. **Mutant Creation subsystem:** This system applies mutation operator to main program and creates mutant and compare result of main program and mutant.
5. **Display subsystem:** This system displays result.

V. PRACTICAL RESULTS

This proposed framework provides actual implementation of test case creation using c# language. Following Figure 1 shows the main framework of mutation testing of C# program.

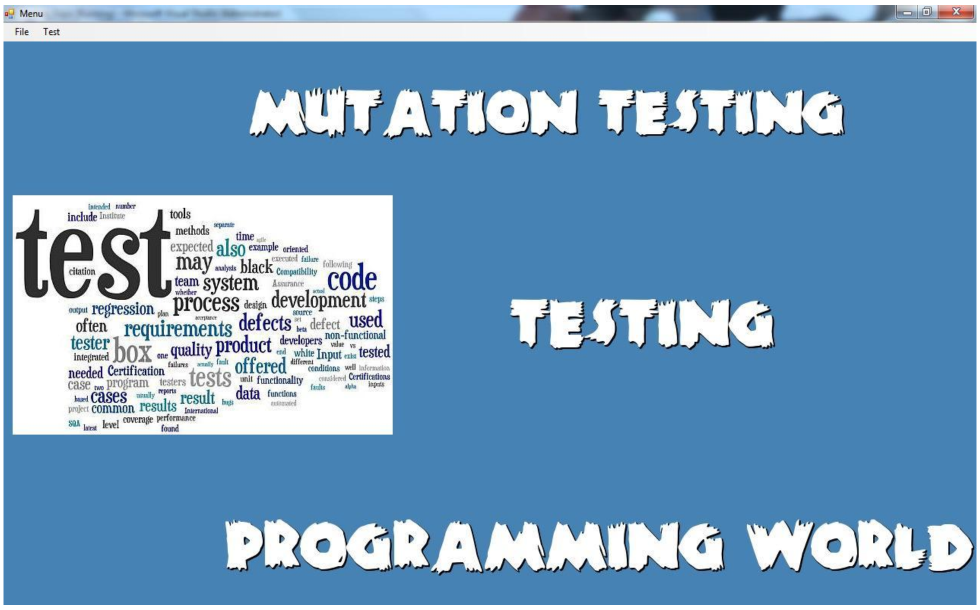
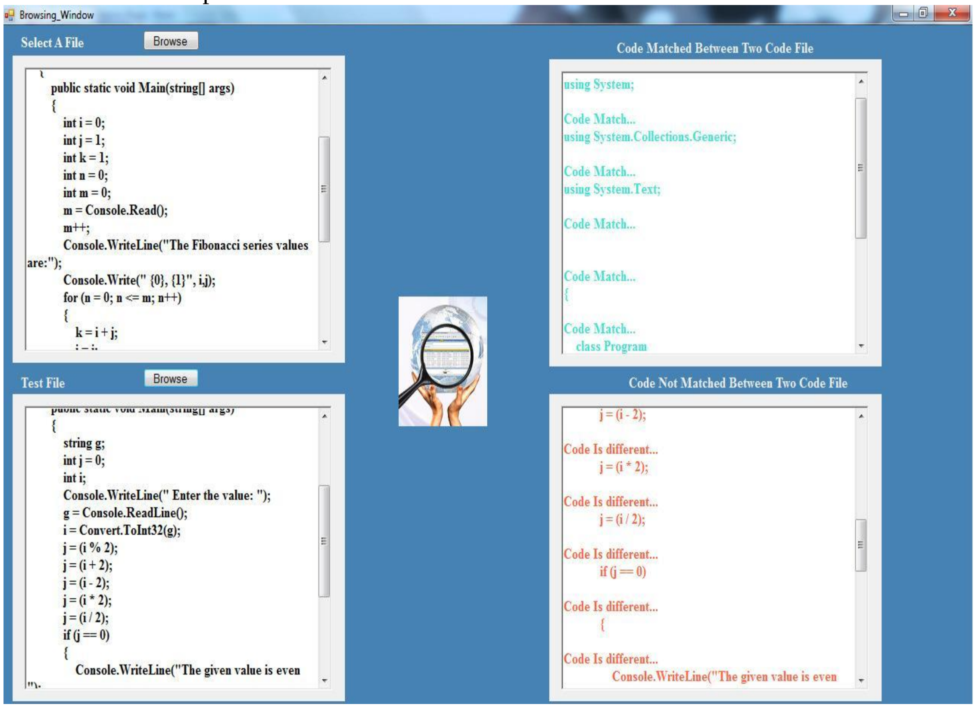


Figure 3: Framework of Mutation testing

Following Figure 2 is shows the match code of two programs. Here we check whether input code & test file code equals or not, if same then which part is match.


Browsing_Window
Select A File
Browse
Test File
Browse
Code Matched Between Two Code File
Code Not Matched Between Two Code File

```
    public static void Main(string[] args)
    {
        int i = 0;
        int j = 1;
        int k = 1;
        int n = 0;
        int m = 0;
        m = Console.Read();
        m++;
        Console.WriteLine("The Fibonacci series values
are:");
        Console.Write(" {0}, {1}", i,j);
        for (n = 0; n <= m; n++)
        {
            k = i + j;
```

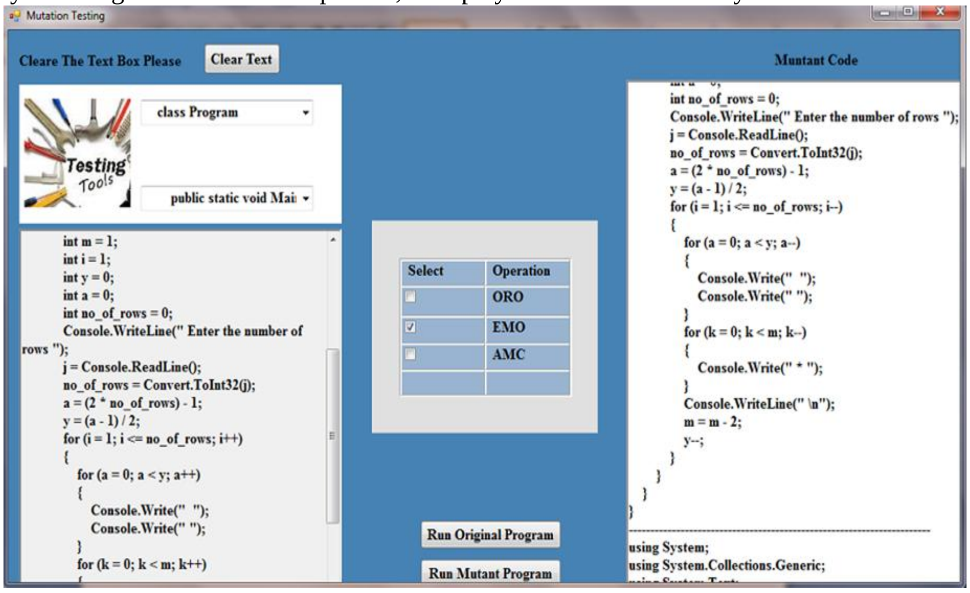
```
    {
        string g;
        int j = 0;
        int i;
        Console.WriteLine(" Enter the value: ");
        g = Console.ReadLine();
        i = Convert.ToInt32(g);
        j = (i % 2);
        j = (i + 2);
        j = (i - 2);
        j = (i * 2);
        j = (i / 2);
        if (j == 0)
        {
            Console.WriteLine("The given value is even
");
```

```
using System;

Code Match...
using System.Collections.Generic;

Code Match...
using System.Text;

Code Match...


Code Match...
{

Code Match...
   class Program
```

```
      j = (i - 2);

Code Is different...
      j = (i * 2);

Code Is different...
      j = (i / 2);

Code Is different...
      if (j == 0)

Code Is different...
      {

Code Is different...
         Console.WriteLine("The given value is even
```

Figure 4: input file and test file matched code and unmatched code

Create mutant by selecting one of mutation operator, it displays mutants successfully



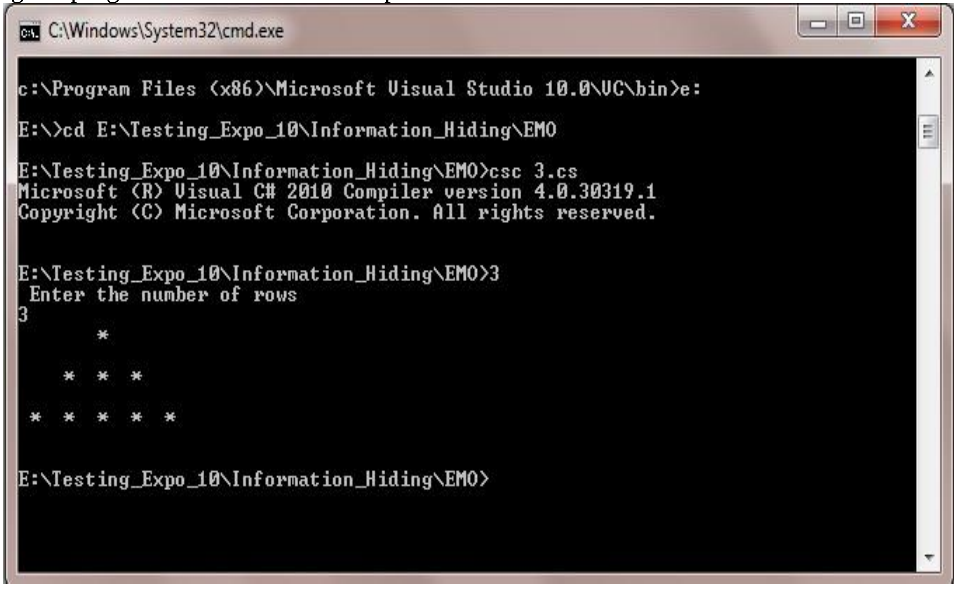
```
        int m = 1;
        int i = 1;
        int y = 0;
        int a = 0;
        int no_of_rows = 0;
        Console.WriteLine(" Enter the number of
rows ");
        j = Console.ReadLine();
        no_of_rows = Convert.ToInt32(j);
        a = (2 * no_of_rows) - 1;
        y = (a - 1) / 2;
        for (i = 1; i <= no_of_rows; i++)
        {
          for (a = 0; a < y; a++)
          {
            Console.Write("  ");
            Console.Write(" ");
          }
          for (k = 0; k < m; k++)
```

Select	Operation
☐	ORO
✓	EMO
☐	AMC

```
        int no_of_rows = 0;
        Console.WriteLine(" Enter the number of rows ");
        j = Console.ReadLine();
        no_of_rows = Convert.ToInt32(j);
        a = (2 * no_of_rows) - 1;
        y = (a - 1) / 2;
        for (i = 1; i <= no_of_rows; i--)
        {
          for (a = 0; a < y; a--)
          {
            Console.Write("  ");
            Console.Write(" ");
          }
          for (k = 0; k < m; k--)
          {
            Console.Write(" * ");
          }
          Console.WriteLine(" \n");
          m = m - 2;
          y--;
        }
      }
    }
}
--------------------------------------------------------------
using System;
using System.Collections.Generic;
```

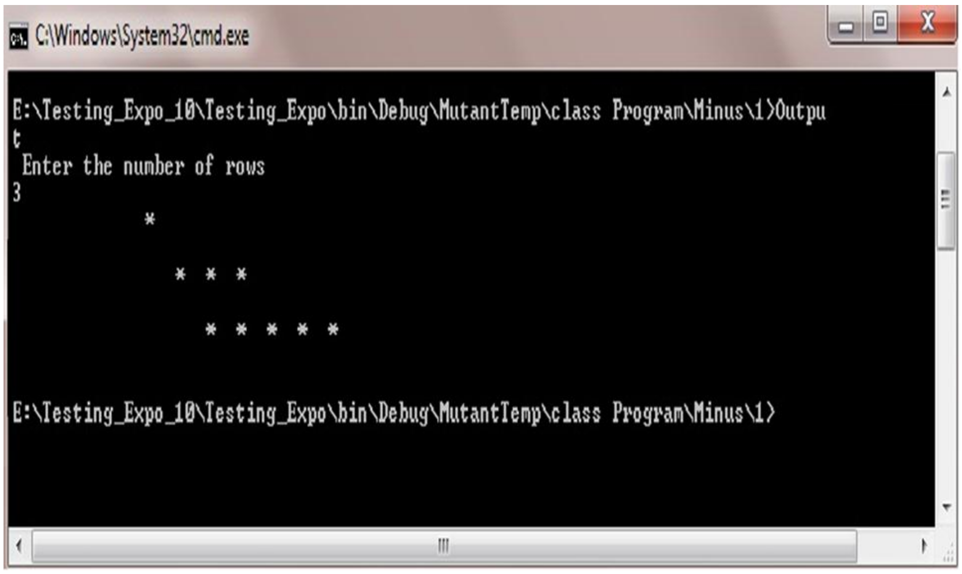
Figure5: Select and apply mutation operator and display mutants

Execute Original program and mutant and compare result.



```
c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin>e:

E:\>cd E:\Testing_Expo_10\Information_Hiding\EMO

E:\Testing_Expo_10\Information_Hiding\EMO>csc 3.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.


E:\Testing_Expo_10\Information_Hiding\EMO>3
 Enter the number of rows
3
        *

    *  *  *

 *  *  *  *  *


E:\Testing_Expo_10\Information_Hiding\EMO>
```

Figure6: Output of Original Program.



```
E:\Testing_Expo_10\Testing_Expo\bin\Debug\MutantTemp\class Program\Minus\1>Outpu
t
 Enter the number of rows
3
            *

               *  *  *

                  *  *  *  *  *


E:\Testing_Expo_10\Testing_Expo\bin\Debug\MutantTemp\class Program\Minus\1>
```

Figure7: Output of Mutant.

VI. CONCLUSION

To find faults in C# programs, mutation operators are provided. These operators of mutation are based on features of object oriented. To support interclass level testing these mutation operators are developed using C# and help developers ,testers to find faults related with inheritance, information hiding. Thus, this system provides method to improve quality of testing and improve efficiency using various mutation operators. Automated mutation testing is done without requiring manual intervention.

VII. FUTURE WORK

For future work, there are various regions and a need to get mutation information from other projects. There is possibility that live mutants may be causes problem that cannot be killed, these mutants are also called as equivalent mutants. Analyzing test outcomes and verifying equal mutant behavior manually is complicated process; hence there is a need of future investigation for equivalent mutants.

The computational cost of running all the mutations against a test set is high. In some cases to test multiple mutants, it requires more computation time. To overwhelm such problems prefer cost reduction techniques such as named 'do smarter' ,'do faster' and 'do fewer' to reduce mutant execution cost.

Given techniques such as Mutant Clustering, Mutant Sampling as well as Selective Mutation will use to decrease mutants size. As total no of mutants decrease by using above technique, mutant execution speed will increase simultaneously which increases the chances of industry adoption in future.

REFERENCES

[1] Mutation at System and Functional Levels, Mateo, P.R. Usaola, M.P. Offutt, Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), Paris, France, 2010.

[2] Test-Driven Development by Example, Beck K., 2003: Addison- Wesley.

[3] Hints on test data selection: Help for the practicing programmer R. J. Lipton, R. A. DeMillo and F. G. Sayward. IEEE Computer, April 1978.

[4] MuJava : An Automated Class Mutation System, Yu-Seung Ma, Je Ofutt, and Yong Rae Kwon ,Division of Computer Science Department of Electrical Engineering and Computer Science Korea, Advanced Institute of Science and Technology, Korea

[5] Two notions of correctness and their relation to testing. T. A. Budd and D. Angluin Acta Informatica, November 1982.

[6] CREAM- A System for Object-Oriented Mutation of C# Programs, A. Derezi´nska and A. Szustek, Warsaw University of Technology, Warszawa, Poland, Technique Report, 2007.

[7] A Fortran Language System for Mutation-based Software Testing, King K., Offutt A., Software Practice and Experience, July 1991.

[8] Tool for the Assessment of Test Adequacy for C Programs, Delmaro M., Maldonado J.Proteum ,In Proc. of Conf. on Performability in Computing Sys., PCS96, Jul. 1996, pp. 79-95.

[9] Nester, http://www.nester.sourceforge.net

[10] Mutation Testing for JAVA,Sunita Garhwal, Ajay Kumar ,Poonam Sehrawat , U.I.E.T. Kurukeshtra University Kurukeshtra Dept of. Computer Science & Eng. Thapar Institute of Eng. & Tech Lecturer U.I.E.T. K.U.K.

[11] Test case generation using mutation operators and fault classification, Mrs. R. Jeevarathinam, Dr. Antony Selvadoss Thanamani, Department of Computer Science SNR Sons College, Coimbatore, Tamilnadu, India. Associate Professor and Head Department of Computer Science NGM College, Pollachi, Tamilnadu, India.