International Journal of Management, IT & Engineering

(ISSN: 2249-0558)

# CONTENTS

# Chief Patron

**Dr. JOSE G. VARGAS-HERNANDEZ**
Member of the National System of Researchers, Mexico

Research professor at University Center of Economic and Managerial Sciences,
University of Guadalajara
Director of Mass Media at Ayuntamiento de Cd. Guzman
Ex. director of Centro de Capacitacion y Adiestramiento

# Patron

**Dr. Mohammad Reza Noruzi**
PhD: Public Administration, Public Sector Policy Making Management,
Tarbiat Modarres University, Tehran, Iran
Faculty of Economics and Management, Tarbiat Modarres University, Tehran, Iran
Young Researchers' Club Member, Islamic Azad University, Bonab, Iran

# Chief Advisors

**Dr. NAGENDRA. S.**
Senior Asst. Professor,
Department of MBA, Mangalore Institute of Technology and Engineering, Moodabidri

**Dr. SUNIL KUMAR MISHRA**
Associate Professor,
Dronacharya College of Engineering, Gurgaon, INDIA

**Mr. GARRY TAN WEI HAN**
Lecturer and Chairperson (Centre for Business and Management),
Department of Marketing, University Tunku Abdul Rahman, MALAYSIA

**MS. R. KAVITHA**
Assistant Professor,
Aloysius Institute of Management and Information, Mangalore, INDIA

**Dr. A. JUSTIN DIRAVIAM**
Assistant Professor,
Dept. of Computer Science and Engineering, Sardar Raja College of Engineering,
Alangulam Tirunelveli, TAMIL NADU, INDIA

# Editorial Board

# Editorial Advisory Board

## Technical Advisors

## Associate Editors

Title

# HIGHER ORDER MUTATION TESTING

# (RESULT- EQUIVALENT MUTANTS)

Author(s)

**Shalini Kapoor**

*CSE Deptt, GNI, Mullana Haryana*

**Rajat Kapoor**

*Assistant Manager Finance, Accenture, Noida*

## ABSTRACT:

Whenever we make a single change to the original program we get First Order Mutant (FOM). When we apply another single change to FOM we get Second Order Mutant (SOM).On applying another single change to SOM we get Third Order Mutant (TOM).Mutants other than FOM are called Higher Order Mutant (HOM).In this paper we prove that as we move from FOM to SOM to TOM there will not be any test data that will kill the Original Program resulting to the formation of equivalent mutants. Equivalent Mutants are never killed hence they will never detect any fault and thus they are considered useless.

**Keywords**: First Order Mutant (FOM), Second Order Mutant (SOM), Third Order Mutant (TOM), Higher Order Mutant (HOM).

## 1) INTRODUCTION:

Mutation testing (or Mutation analysis) is a method of software testing, which involves modifying program's source code in small ways. These, so-called mutations, are based on well-defined mutation operators that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a mutant. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be killed.

For example, consider the following C++ code fragment**:**

```
if (a && b)
   c = 1;
else
```

```
    c = 0;
```

The condition mutation operator would replace '&&' with '||' and produce the following mutant**:**

```
if (a || b)
    c = 1;
else
    c = 0;
```

Now, for the test to kill this mutant, the following condition should be met:

- Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this.
- The value of 'c' should be propagated to the program's output and checked by the test.

Generally, in the mutation testing, a fault is introduced by a small modification of a correct program code. The modified program is called mutant, and this process is called mutation. A transformation rule that generates a mutant from the original program is known as a mutation operator. If the mutant and the original program generate different outputs for a test case then the mutant is called **killed mutant.** The mutant is called **alive,** if no test case can distinguish between the mutant and the original program. If the mutant survives, then the test data is considered insufficient to explore the fault. In that case, the test data is extended until such a mutant is killed. Sometime, it is not possible to find a test case that distinguishes between the output of the mutant and that of the original program in which case the mutant is called **equivalent mutant [10].**

Mutation testing performs “change and check” testing strategy. Original program is slightly modified and then executed. The output of original program and that modified program with respect to the same input set are then compared. For example - we have a program P and slightly modified (mutated) program P’ and Let I be the input set. With execution of same input set I, program P gives output O and program P’ give output O’.

I→P→→O

I → P ‘→ → O’

If (O’ ≠ O) then it means, our test case is adequate and the functionality of the program is good.

Otherwise, if (O’ = O), then our test case is inadequate and the functionality of the program is poor.

Consider the example for mutant generation given in [12].

**Example:** Consider the program P =

```
if (c==a+b)
doThis();
else doThat();
```

Some of the possible mutants of P would be

```
P1: if (c==a-b)
doThis();
else doThat();
```

```
P2: if (c==a*b)
doThis();
else doThat();
```

```
P3: if (c==a/b)
doThis();
else doThat();
```

```
P4: if (c>a+b)
```

```
doThis();

else doThat();

P5: if (c<a+b)

doThis();

else doThat();
```

If the value of a=2 and b=2 then P2 is an equivalent mutant of P because it is not possible to find a test case that can ever kill this mutant.

Mutation testing is typically computationally expensive because a program may have a large number of faults, and there may be a large number of mutants for even a small software unit. Therefore, we need to generate test case in such a way that the test data make the execution of the program to reach each mutated statement **[7].**

Weak mutation testing (or weak mutation coverage) requires that only the first condition is satisfied. Strong mutation testing requires that both conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

## 2) EQUIVALENT MUTANTS:

Many mutation operators can produce equivalent mutants. For example, consider the following code fragment**:**

```
int index=0;

while (...)

{

   . . . ;

   index++;

   if (index==10)
```

```
    break;
}
```

Boolean relation mutation operator will replace "==" with ">=" and produce the following mutant:

```
int index=0;
while (...)
{
 . . .;
 index++;
 if (index>=10)
    break;
}
```

However, it is not possible to find a test case which could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called equivalent mutants.

Equivalent mutants detection is one of biggest obstacles for practical usage of mutation testing. The effort, needed to check if mutants are equivalent or not, can be very high even for small programs.

## 3) MUTATION OPERATORS:

A variety of mutation operators were explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion.
- Replace each boolean subexpression with true and false.
- Replace each arithmetic operation with another one, e.g. + with *, - and /.
- Replace each boolean relation with another one, e.g. > with >=, == and <=.

- Replace each variable with another variable declared in the same scope (variable types should be the same).

These mutation operators are also called traditional mutation operators. Beside this, there are mutation operators for object-oriented languages, for concurrent constructions, complex objects like containers etc. They are called class-level mutation operators. For example the MuJava tool offers various class-level mutation operators such as: Access Modifier Change, Type Cast Operator Insertion, and Type Cast Operator Deletion.

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.

Using only first-order mutants has been justified in two ways. Firstly, it is argued that if our test finds the small differences defined by first-order mutants, then it is likely that it will find larger differences defined by higher-order mutants**:** this is called the coupling effect. Secondly, it is also argued that real programmers make small mistakes and thus that real programs are like first-order mutants of correct programs**:** this is called the competent programmer hypothesis.

The reason for only using first-order mutants is also pragmatic**:** if we do not restrict ourselves to first-order mutants, then the total number of mutants is likely to be extremely large. In fact, even when we only produce first-order mutants, mutation testing tools produce large numbers of mutants for even small pieces of code. This is one of the reasons why mutation testing currently does not scale up beyond unit testing.

A test case t kills the mutant p’ of p if p and p’ produce different output when given the input from t. A mutant p’ of p is an equivalent mutant if no test input kills p’. An equivalent mutant p’ of p is syntactically different from p (the code is different) but semantically equivalent (p and p’ define the same input/output function).In mutation testing, system produce some set of mutants and a test set is said to be adequate if it kills all of the non-equivalent mutants. The mutation coverage measurement is the percentage of the non-equivalent mutants produced that are killed by the test set. The aim is to produce a test set that achieves 100% coverage.

## 4) PROPOSED WORK:

To demonstrate that Higher Order Mutation Testing leads to equivalent mutants we take the example for swapping of two numbers in C language.

**Original_Program**

```
#include<stdio.h>

#include<conio.h>

void main()

{

int temp , x,y;

clrscr();

printf(“Enter the numbers to be swapped”);

scanf(“%d%d”,&x,&y);

temp=x;

x=y;

y=temp;

printf(“The numbers after swapping are%d%d”,x,y);

}
```

In the above program if value of x=5,y=10 than after swapping x will be equal to 10 and y will be equal to 5. Now we introduce a single change(changed variable x to y) in the Original_Program which is shown in Bold below and call it FOM(First Order Mutant)

**FOM**

1) **temp=y;**
2) x=y;
3) y=temp;

Change is made at line1. If value of x=5, y=10 FOM will give the output as x=10, y=10.

Since this output is different from Original_Program we say that FOM is killed.

Now we introduce a single change to the FOM which is shown in Bold below and call it SOM (Second Order Mutant)

**SOM**

1) temp=y;
**2) y=x;**
3) y=temp;

Change is made at line2. If value of x=5, y=10 SOM will give the output as x=5, y=10.

Since this output is different from Original_Program we say that SOM is killed.

Now we introduce a single change to the SOM which is shown in Bold below and call it TOM (Third Order Mutant)

**TOM**

1) temp=y;
2) y=x;
**3) x=temp;**

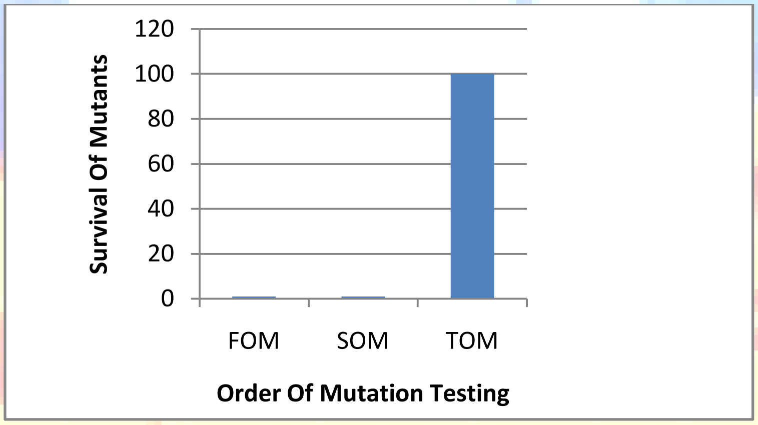
Change is made at line3. If value of x=5, y=10 TOM will give the output as x=10, y=5.

Since this output is same as Original_Program we say that TOM is alive. Since there is no test case that can kill TOM we say that this produces Equivalent Mutants. Higher in the order we went, we got equivalent mutants.

We depict the results of mutants in the table below.

| MUTANT | DATA & RESULT | | STATUS |
|---|---|---|---|
| | X=5 | Y=10 | |
| Original Program | 10 | 5 | ---- |
| FOM | 10 | 10 | KILLED |
| SOM | 5 | 10 | KILLED |
| TOM | 10 | 5 | ALIVE |

The result is shown in graph below:



## 5) CONCLUSION:

The paper concludes that Lower Order Mutation Testing (LOM) is more powerful in finding faults. As we move to Higher Order Mutation Testing –Third Order Mutants and higher, Equivalent Mutants are obtained which have very high survival rate and hence turn out useless for finding faults.

## 6) REFERENCES:

- A. T. Acree. On Mutation. Phd thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.
- T. A. Budd. Mutation Analysis of Program Test Data. Phd thesis, Yale University, New Haven, Connecticut, 1980.
- W. E.Wong. On Mutation and Data Flow. Phd thesis, Purdue University, West Lafayette, Indiana, 1993.
- A. P. Mathur and W. E. Wong. An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria. Technique report, Purdue University, West Lafayette, Indiana, 1993.
- A. S. Namin and J. H. Andrews. On Sufficiency of Mutants. In Proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION'07), pages 73–74, Minneapolis, Minnesota, 20-26 May 2007.
- A. P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. In Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC'79), pages 604–605, Tokyo, Japan, 11-13 September 1991.
- M. Sahinoglu and E. H. Spafford. A Bayes Sequential Statistical Procedure for Approving Software Products. In Proceedings of the IFIP Conference on Approving Software Products (ASP'90), pages 43–56 Garmis Partenkirchen, Germany, September 1990. Elsevier Science.
- R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88), pages 142–151, Banff Alberta, Canada, July 1988. IEEE Computer society.
- A. J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In Proceedings of the 15th International Conference on Software Engineering (ICSE'93), pages 100–107, Baltimore, Maryland, May 1993. IEEE Computer Society Press.
- W. E. Wong and A. P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software, 31(3):185–196, December 1995.

- K. N. King and A. J. Offutt. A Fortran Language System for Mutation- Based Software Testing Software: Practice and Experience, 21(7):685–718, October 1991

- E. S. Mresa and L. Bottaci. Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. Software Testing, Verification and Reliability, 9(4):205–232, December 1999.

- A. S. Namin and J. H. Andrews. Finding Sufficient Mutation Operators via Variable Reduction. In Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06), page 5, Raleigh, North Carolina, November 2006. IEEE Computer Society.

- A. S. Namin and J. H. Andrews. On Sufficiency of Mutants. In Proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION'07), pages 73–74, Minneapolis, Minnesota, 20-26 May 2007.

- A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology, 5(2):99–118, April 1996.

- R. M. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. Software Testing, Verification and Reliability, 9(4):233–262, December 1999.

- M. Harman, R. Hierons, and S. Danicic. The Relationship Between Program Dependence and Mutation Analysis. In Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00), pages 5–13, San Jose, California, 6-7 October 2001. Published in book form, as Mutation Testing for the New Century.

## 7) AUTHORS PROFILE:

Shalini Kapoor received the bachelor degree in Computer Science and Engineering from Haryana Engineering College, Jagadhri, India in 2003. She received her Master degree in Information Technology from Karnataka State University, Mysore, India in 2011. She has 2.5 years industrial experience and 4.5 years teaching experience. Presently she is working in Computer Science and Engineering Department of Guru Nanak Institutions Mullana.

Rajat Kapoor is currently qualified as CFA Level II, MBA Finance, Post Graduate Diploma in Personnel Management & Labour Welfare, Single course certification in Marketing Management, Single course certification in Business Law, NSE Certified, and Diploma from Aptech Education Centre. He has more than 6 Years of professional experience in Finance & Accounts domain serving Royal Enterprises Group, YamunaNagar (Comprising of Royal Enterprises, Sumo Wood Industry, R.K Industries) dealing in Water Treatment, Wooden Cooling Towers, Plywood, Veneer, Cable Drums, Defence Boxes, Wooden Packing Boxes, etc business for 5+ Years, Pyramid IT Consulting Pvt Ltd, Noida dealing in IT&ITES, Staffing, Recruitment business for 1 Year and is currently working as Assistant Manager Finance at Accenture, Noida upon a process of Cairn Energy India Pty Ltd. He is conceptually strong with an innovative and analytical approach to the work with an eye for detail. He is in a habit of using a systematic approach in objectively identifying the real source of a given problem, design solutions, think sceptically & test the theories (related to the research works) in order to prove & improve the final product.