Reducing Mutation Testing Endeavor and Deporting the Equivalent Mutants

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Tannu Singla
(Roll No. 801131026)

Under the supervision of:
Ajay Kumar
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Reducing Mutation Testing Endeavor and Deporting the Equivalent Mutants*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ajay Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Tannu Singla)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ajay Kumar)
Assistant Professor,
CSED

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all, I am thankful to God for his blessings and showing me the right direction. With his mercy, it has been made possible for me to reach so far.

It gives me great pleasure to express my gratitude towards the guidance and help I have received from Mr. Ajay Kumar. I am thankful for his continual support, encouragement, and invaluable suggestion. It has been a great honor to work under him.

I am also thankful to Dr. Maninder Singh, Head, Computer Science and Engineering Department for his kind help and cooperation. I express my gratitude to Mr. Sumit Miglani and all other staff members of computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I would like to say thanks to all my friends especially Parneet Kaur, Vaibhav Aggarwal, Mehak Jain, Taranpreet Kaur, Amandeep Bakashi and Rupinder Singh for their support. I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

I am forever indebted to my parents for their constant support and encouragement throughout the past years. This thesis is devoted to them.

Tannu Singla

Tannu Singla

(801131026)

Abstract

Software testing is an important technique for the assurance of software quality. Mutation testing is the white-box fault-based testing technique for unit testing. It is a software testing technique that ameliorates the quality and reliability of the critical software. Mutation testing deals with mutating parts of the program intentionally and then detecting them. The purpose is not to find the faults but to generate an effective test suite, which can detect all the faults in the program.

Mutation testing suffers from the problem of high computational cost. The computational cost is high due to many mutants are generated for even small programs. We have experimented with different programs of variable sizes, and were able to come up with the solution of the problem.

Mutation Testing is one of the most dexterous testing techniques in retracing the faults. In order to evaluate the exact mutation score in mutation testing, the vital question is whether a mutant is equivalent to its program. Unfortunately, the answer to this question is not always possible. In this thesis, we are introducing mutation operators and conditions that deport the mutants to behave as equivalently. Based on the specific criteria, detection of equivalent mutants of the program becomes ingenious. They are utilitarian in calculating the mutation score of the program accurately.

This thesis presents a mutation testing technique based on the concept of the same mutation operator under similar conditions occur at different locations in the program. In the proposed technique, we assemble the tantamount behavior mutants under a group and a single mutant is contemplated from the group for performing mutation testing. The benefits of the proposed approach are reduction in time, effort and cost.

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

1.1Software Testing

Software testing is a crucial task in software development life cycle. Recent trends in software engineering confirm the importance of this activity all along the development process. Testing activities get started at the requirements specification stage, propagate down to the deployment stage and even continue after it. Testing is a concurrent lifecycle process of engineering, in order to measure and improve the quality of the software. Testing is the process of detecting differences between the existing and required conditions of the software. It helps in the evaluation of the software features. Testing software is more complex than exercising a program to see if it works.

Software testing is an evolutionary process to determine the existence of errors in the software. It is a process of executing programs with the objective of finding errors. There are many ways of software testing like review, inspection, audit, walk- through, etc. Testing is one of the most important process to verify and validate the correctness of the software. It also enhances the reliability of software. The cost of testing is about 40% of the overall software costs. Software testing activities includes generating test inputs, creating expected outputs, run software on input data, and verification of the output results.

A program failure is caused by a fault. Fault is a defect in the code. At times, fault is undetected for a long time, because of an error. Error is defined as program's intermediate unstable state [1]. Testing uses the test cases for the detection of faults. Test cases include the set of inputs with their execution conditions and the set of expected results with post- conditions. The collection of test cases typically related to a goal is called test suit. The test case which detects at least a fault that produces different output of the program than that of expected output is called adequate test case. If the test case is not able detect even a single fault, it is called inadequate test case. A good test case is the one having a high probability of finding the errors yet not discovered. Testing is a process

of executing a program with the intention of finding errors. Aim of testing is to trace fulfill all customer requirements.

1.1.1 Software Testing Process

The software testing process is composed of different activities. All these activities are managed, in a sequence to test any software. It is a process in which faults are located and corrected. It is process of fixing faults discovered during testing. The software testing lifecycle acts as a guide to the management so that progress is measurable in the form of achieving milestones.

The activities incorporated in software testing lifecycle are as follows [2]:

1.1.1.1 Test Planning

Test planning takes into account the important issues of testing strategy, responsibilities, risks, resource utilization, and priorities. It describes approach, scope and resources of the items to be tested and identified risks associated with them. The test plan is carried out at each level of the software product developed life cycle. Test plans are created at different levels of software depending upon the scope of the project. The system test plan includes a group of test cases that establish the system to be stable and all main functionality are working. The system test plan is started as soon as the first draft of requirements is complete. The output of the test planning is the test plan document for each level of testing.

1.1.1.2 Test Designing

The Test design process includes critical activities like determining the test objectives, selection of test case design techniques, developing test procedures, preparing test data, setting up the test environment, tools and test case pass/fail criteria. One rule of designing test cases is to cover all features but do not create too many test cases [2]. The test cases created are part of a document called test design specification [4]. The objective of the test design specification is to group similar test cases together. The test design specification documents have environmental needs, input specifications, output specifications and other requirements for the test case.

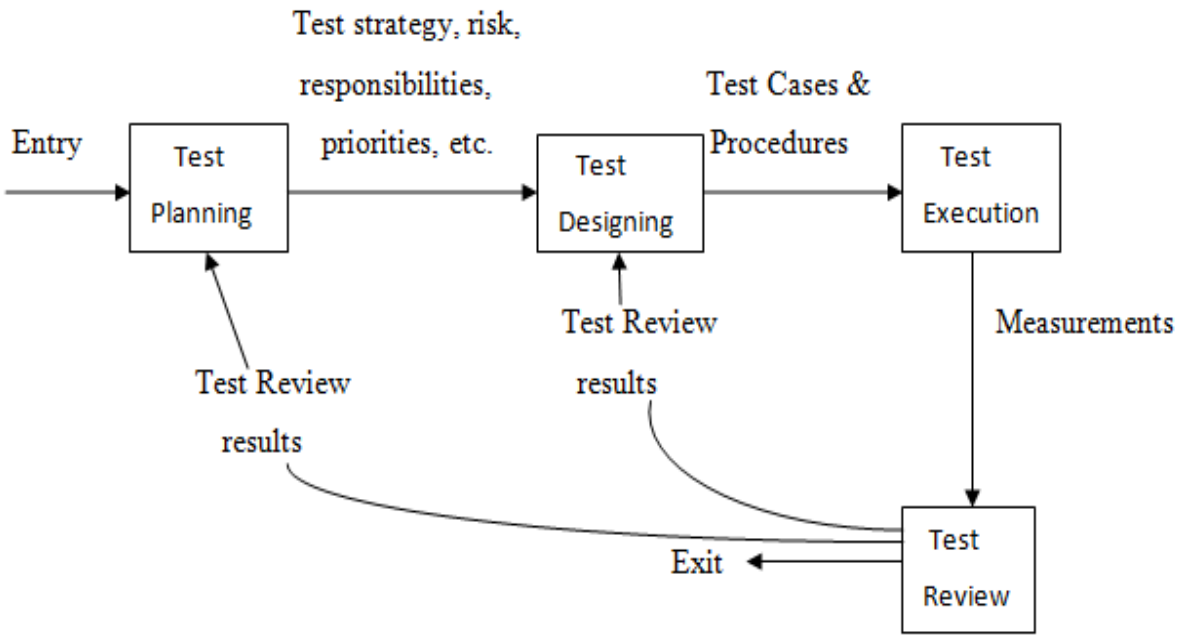

Figure 1.1: Software testing lifecycle [3]

1.1.1.3 Test Execution

Test execution is the process of running selected test cases and observing the results. It occurs later in software development lifecycle when code development activity is almost completed. The output of test execution includes test incident reports, test logs, testing status and test summary report [3]. Team members use test log as a record what occurred during test execution [4]. Test incident report has a description of events that occurred during testing requiring further investigation. Test summary report includes the summary of designed testing activities.

1.1.1.4 Test Review

The purpose of test review process is to analyze the data collected during software testing, to give feedback of the test planning, test designing and test execution activities [3]. Different assessments are performed on the review of software e.g. reliability analysis, coverage analysis etc. If the software meets the assessment criteria, it is ready to be released otherwise software need to be improved.

1.1.2 Software Testing Techniques

They are of two types:

1.1.2.1 Static Testing

It refers to the examination of software requirement specification (SRS), software design specifications (SDC), project documentation and other non-executable items through audits, desk checks, inspections, reviews, etc. Static testing verifies the software items requirements, design and code before execution of the test cases.

1.1.2.2 Dynamic Testing

Dynamic testing is to test the dynamic behavior of the software. It includes the development of test cases, test designing, test execution and reports. Black box testing and white box testing are two popular ways of dynamic testing. Both of these require the set of well-developed and well-structured test cases. It proves the correctness of software by performing in an exhaustive manner [5]. Dynamic testing is used for validating the software.

1.1.3 Classification of Test Techniques

Classification of test techniques is done on the basis of source of information deriving test data and criteria to measure the adequacy of test cases.

1.1.3.1 White-Box Testing

White-Box Testing is also known as glass-box testing. It uses the internal matters for the creation of tests. It is the testing underlying implementation of a piece of software without considering the external descriptions for that piece of software. It exercises internal data structure for their validation. It detects failures and errors in the code. White-box testing techniques are data flow testing, control flow testing, mutation testing, path testing, branch testing, etc.

1.1.3.2 Black-Box Testing

Black-Box Testing is also known as functional and behavioral testing. It enables the software engineer to derive sets of input conditions that will exercise all the functional requirements for a program [6]. It is the type of testing which tests a piece of software

without considering its underlying implementation. Black-box testing is applied on the later stages of testing. Black-box testing techniques include boundary value analysis, equivalence partitioning, etc.

1.1.3.3 Gray-box Testing

Gray-box testing is also known as translucent testing. It is an effective combination of white-box testing and black-box testing [6]. In gray-box testing internal structure is partially known to the tester. On the basis of this partial knowledge, the test cases are designed, and the piece of software under tests is considered as a black box and tester test it from outside. Gray-box testing techniques are pattern testing, matrix testing, etc.

1.2 Mutation Testing

Mutation testing is a white box fault-based testing technique. It works in combination with the traditional testing techniques. Mutation testing is not like other testing techniques which focus on correct functionality of the program. It focuses on the test cases used to test the programs. Its purpose is to provide a good set of test cases instead of trying to find the faults present in a program. Good test cases are able to discover all the possible faults in the program. Mutation testing is one of the strong testing criteria for the evaluation of programs as well as test suites [7]. Fault detection in software's is must e.g. In 1996, the ESA Ariane 5 rocket fails due to a simple unchecked type error. In 2002 software faults in Britain's national tax system resulted in more than 10000 erroneous tax.

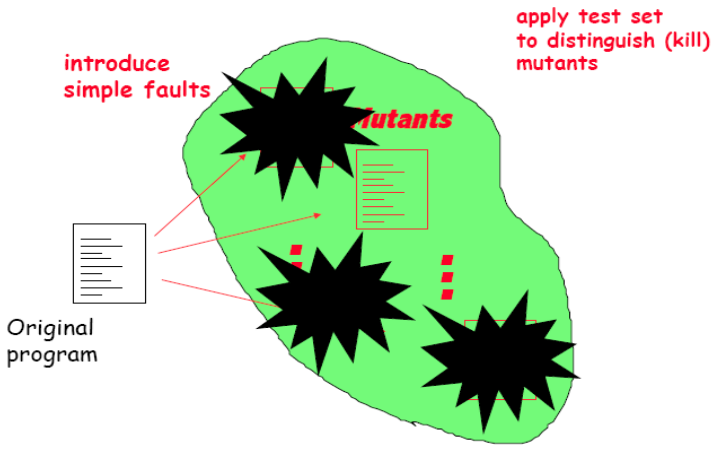

Figure 1.2: Mutation Testing [8]

Mutation testing involves the modification of program source code with slight changes. Slight changes will lead to a divergence in the program called mutant. To make these modifications mutation testing uses the well-defined mutation operators. Mutation system which introduces only one change at a time in the program is called *first-order* mutant. Mutation system which introduces umpteen changes in the program at a time is called *higher-order* mutant. The aim is to help the tester to develop effective test cases and locate weaknesses in the test data to be used for testing.

Example of Mutant:

Original Program		**First-order Mutant**
Public static in gcd (int p, int q) {		Public static in gcd (int p, int q) {
While(q!=0) {		While(q!=0) {
int temp;		int temp;
temp = q;		temp = q;
q = p%q;	*	**q = p/q;**
p = temp;		p = temp;
}		}
Return p;		Return p;
}		}

1.2.1 Mutation testing has following steps:

- Faults are introduced into the program with the help of mutation operators to create new versions of the program which are known as mutants.

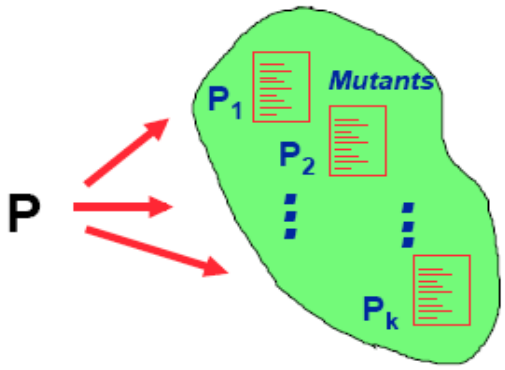

Figure 1.3: Program Mutant's [8]

- Mutant is slightly different from the original program.
- Test cases are applied on the original program and the mutant.
- The aim is to fail the mutant, thus demonstrating effectiveness of the test case [9].

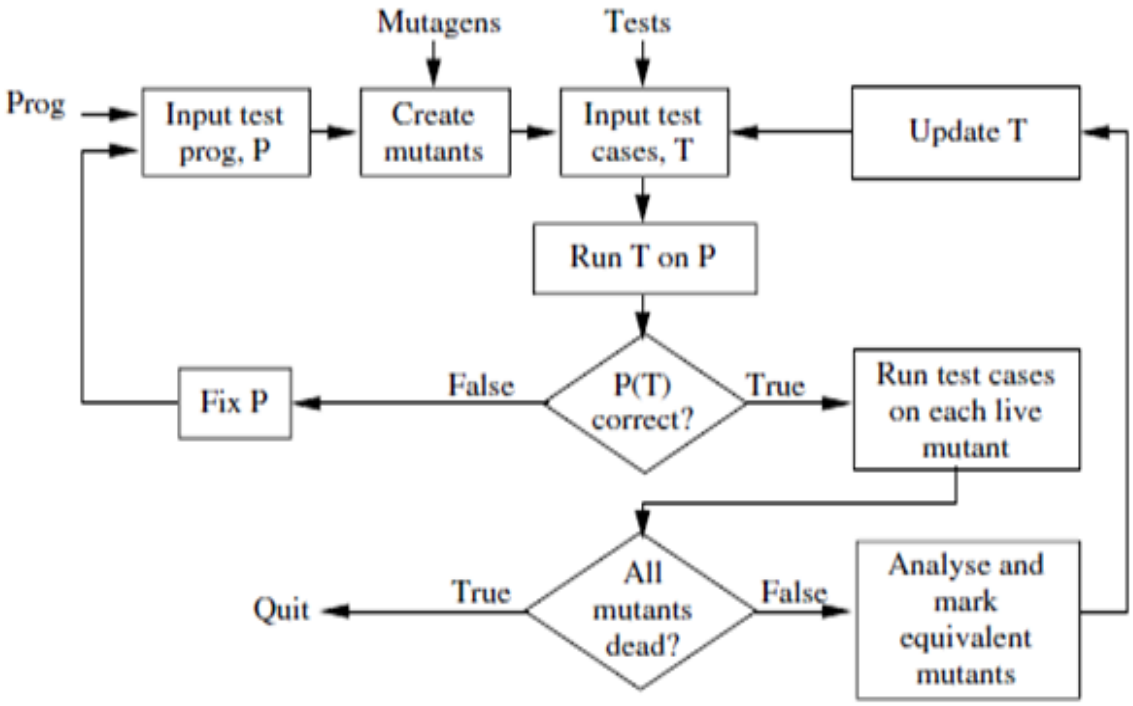

Figure 1.4: Mutation testing Process [10]

1.2.2 Assumptions

There are two types of assumption in mutation testing are:

1. **Competent programmer assumption:** Programmer has written nearly correct program to be desired.
2. **Coupling effect assumption:** states that test cases which kill simple mutants can also kill complex mutants.

1.2.3 Types of Mutant

Killed Mutant: on execution mutant produces different result from the original program.

Alive/Live Mutant: test cases are not able to detect the injected fault in the program.

Equivalent Mutant: a mutant that always produces identical results with the original program.

1.2.4 Equivalent Mutant

A mutant that always produces identical results with the original program is called equivalent mutant. One of the problems related to mutation testing is the detection of equivalent mutant.

Example of equivalent mutant:

	Original Program		**Equivalent Mutant**
1)	Public static in gcd (int p, int q) {		Public static in gcd (int p, int q) {
2)	While(q!=0) {		While(q!=0) {
3)	int temp;		int temp;
4)	temp = q;		temp = q;
5)	q = p%q;		q = p/q;
6)	p = temp;		p = temp;
7)	}		}
8)	**Return p;**	*	**Return p++;**
9)	}		}

However, it is not possible to find a test case, which can produce the mutant output different from the original program output. Thus the mutant is equivalent to the original one. These mutants are called equivalent mutants.

1.2.5 Mutation Score

Mutation score is calculated to evaluate the effectiveness of test cases. Let the program is P and test set T

MS (P,T) = [K/(M-E)] * 100 %

Where K is the total number of killed mutants, M is total number of mutants are generated and E is total number equivalent mutants.

Mutation Score is 100% if all mutants are killed, and if there any equivalent mutant in the program, is get detected.

1.2.6 Conditions for Test Data to kill a mutant

Three conditions are needed to satisfy test data to kill a mutant.

If P denotes a program, M denotes a mutant of P on statement S and T denotes the test data for P.

The conditions are as follows [11, 12]:

1. **Reachability condition:** S must be reached, because a mutant is presented as a syntactic change only in statement S and the other statements in mutant are syntactically equivalent to original program. If T cannot reaches the statement S, impossible to kill M [12].
2. **Necessity condition:** T must be able to bring M to a different state than that of P on statement S. For T to kill M, it is necessary S is reachable and on execution the state of M must be different from P for statement S [12]. M will never get killed, if the state of M and P do not differ after execution of S. Because the rest of statements of P and M are syntactically equivalent.
3. **Sufficiency condition:** The final state of M must be different from P. The different state caused by the necessity condition should propagate through the program's computation to result in a different output [11].

1.2.7 Weak Mutation Testing and Strong Mutation Testing

1.2.7.1 Weak Mutation Testing: It has to satisfy only first two conditions of mutation testing.

Example:

Program P		**Mutant M with weak mutation**
1) Int a,b;		Int a,b;
2) a = 2;		a = 2;
3) b = a + 1;		b = a +1;
4) If(b>a)	*****	**If(b>=a)**
5) {		{

6)	…….		…….
7)	}		}

1.2.7.2 Strong Mutation Testing: It has to satisfy all the three conditions of mutation testing. Strong mutation ensures that the test data caught the injected fault. Strong mutation is more powerful than weak mutation.

Example:

Program P		**Mutant M with strong mutation**
1) int a,b;		int a,b;
2) a = 2;		a = 2;
3) b = a + 1;		b = a +1;
4) If(b>a)	*****	**If(b<a)**
5) {		{
6) …….		…….
7) }		}

1.3 Mutation Operators

Mutation operators are inserted into the program for the creation of mutants. Mutation operator leads to the syntax change in the program. Mutation operators are like replace an operand with another syntactically legal operand, modify the expression by inserting new operator, and delete entire statements, etc. Categorization of mutation operators for Java are method level mutation operators and class level mutation operators.

1.3.1 Method level Mutation Operators

The method level mutation operators are modifying the program by inserting, replacing or deleting the primitive operator [13]. Method level mutation operators for Java are of six types.

1. Arithmetic operators
2. Relational operators
3. Conditional operators
4. Shift operators

5. Logical operators
6. Assignment operators

Some of the operators are sub-divided into binary, unary and short-cut versions. There are total 16 types of method-level operators. Presence of a mutant in a program is indicated by *.

Table 1.1: Method level Mutation Operators [14]

Category	***Operator***	***Description***
Arithmetic	AOR_B	Arithmetic Operator Replacement (binary)
	AOR_U	Arithmetic Operator Replacement (unary)
	AOR_S	Arithmetic Operator Replacement (short-cut)
	AOI_U	Arithmetic Operator Insertion (unary)
	AOI_S	Arithmetic Operator Insertion (short-cut)
	AOD_U	Arithmetic Operator Deletion (unary)
	AOD_S	Arithmetic Operator Deletion (short-cut)
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASR_S	Assignment Operator Replacement (short-cut)

1.3.1.1 Arithmetic Operators

Arithmetic operators perform mathematical computations on integers and floating point numbers. Arithmetic operators supported in Java are:

- Binary: v + v (addition), v - v (subtraction), v * v (multiplication), v / v (division), v % v (modulus).
- Unary: + (indicates positive value), - (indicates negative value).
- Short-cut: v++ (post-increment), ++v (pre-increment), v-- (post-decrement), --v (pre-decrement).

The arithmetic operators are explained below:

- **AOR_B/AOR_U/AOR_S:** Arithmetic operator replaces binary, unary and short-cut with another binary, unary and short-cut respectively.
- **AOI_U/AOI_S:** Arithmetic operator inserts unary and short-cut respectively. There is probability AOI_S operator show equivalence [11].
- **AOD_U/AOD_S:** Arithmetic operator deletes unary and short-cut respectively.

1.3.1.2 Relational operators

Relational operators compare the value of two operands. Relational operators supported in Java are:

- v > v (greater than), v >= v (greater than equal to), v < v (less than), v <= v (less than equal to), v == v (equal to) and v != v (not equal to).

Only replacement is possible in relational operators.

- **ROR:** This operator replaces a relational operator with another relation operator.

1.3.1.3 Conditional Operators

Conditional operators perform computations on the binary values of its operands. Condition operators supported in Java are:

- Binary: v && v (conditional AND), v || v (conditional OR), v & v (bitwise AND), v | v (bitwise OR) and ^ (bitwise XOR).
- Unary: !v (bitwise logical complement).

Conditional operators are explained below:

- **COR:** This operator replaces a binary conditional operator with another binary conditional operator.
- **COI:** This operator inserts unary conditional operators.
- **COD:** This operator deletes unary conditional operators.

1.3.1.4 Shift Operators

Shift operators manipulate the bits of the first operand in the expression by shifting to the value of the second operand either to the right or the left. Shift operators supported in Java are:

- v >> v (signed right shift), v << v (signed left shift) and v >>> v (unsigned right shift).

Shift operators are explained below:

- **SOR:** This operator replaces a binary shift operator with another binary shift operator.

1.3.1.5 Logical Operators

Logical operators perform logical comparison to produce a Boolean result for comparison statements. Logical operators supported in Java are:

- binary: v & v (AND), v | v (OR) and v ^ v (XOR).
- unary: ~v (bitwise complement)

Logical operators are explained below:

- **LOR:** This operator replaces a binary logical operator with another binary logical operator.
- **LOI:** This operator inserts unary logical operators.
- **LOD:** This operator deletes unary logical operators.

1.3.1.6 Assignment Operators

Assignment operators are to set the values of an operand. The short-cut assignment operator performs the computation on the right hand operand and then assigns its value to the left hand operand. Assignment operators supported in Java are:

- **For short-cut:** v += v (addition assignment), v -= v (subtraction assignment), v *= v (multiplication assignment), v /= v (division assignment), v %= v (modulus assignment), v &= v (bitwise AND assignment), v |= v (bitwise OR assignment), v ^= v (bitwise XOR assignment), v <<= v (right shift assignment), v >>= v (left shift assignment), v >>>= v (unsigned right shift assignment).

Assignment operators include the following operators:

- **ASR_S:** This operator replaces a short-cut assignment operator with another short-cut assignment operator.

1.3.2 Class level Operators

Class level mutants are categorized into four groups. First three groups are based on common features of all object oriented languages. The last group contains language features that are specific to Java. The groups are:

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

Class-level operators change the program syntax by inserting, modifying and deleting the expression under test. These operators are explained below in detail with examples taken from [15]. The presence of a mutant in a program is indicated by *.

1.3.2.1 Encapsulation (Access Control)

It controls the level of access of data and methods. In Java access levels are private, public and protected.

- **AMC:** This operator replaces the one access modifier with another access modifier. Use of improper access modifier can lead to incorrect results. This helps the tester in ensuring the appropriate level of accessibility is used in the program. This operator has probability of showing equivalence.

Original Code		**Mutants**
public Stack s;	*	private Stack s;
	*	protected Stack s;
	*	Stack s;

Table 1.2: Class level Mutation Operators [14]

Language Features	***Operator***	***Description***
Encapsulation	AMC	Access modifier change
Inheritance	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
	IPC	Explicit call to parent's constructor deletion
Polymorphism	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
	OAN	Arguments of overloading method call change
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
Java-specific Features	JTI	this keyword insertion
	JTD	this keyword deletion
	JSI	static modifier insertion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment & content assignment replacement
	EAM	Accessor method change
	EMM	Modifier method change
	JSD	static modifier deletion

1.3.2.2 Inheritance

Inheritance allows the use of methods and data of one class (parent class) in another class (child class) and boost code reusability. We define the five types of mutation operators are inheritance, covering variable hiding, method overriding, use of super and definition of constructors.

Variables' hiding allows a child class to hide the variables in the parent class. It includes mutation operators are IHD and IHI.

- **IHD:** In this operator parent class variable is accessed by deleting the hiding variable in child class. This operator has probability of showing equivalence.

Original Code	**Mutant**

Class List {
int size;
……
}
Class Stack extends List {
int size;
}
	Class List {
	int size;
	……
	}
	Class Stack extends List {
	* // int size;
	}

- **IHI:** This operator inserts the hiding variable in the child class. The method in the child class was referencing the parent class variable now referencing the inserted hiding variable. It works reverse of IHD. This operator has probability of showing equivalence.

Original Code	**Mutant**

Class List {
int size;
……
}
Class Stack extends List {
……....
……
}
	Class List {
	int size;
	……
	}
	Class Stack extends List {
	* int size;
	……..
	}

A child class modifies the behavior of its parent class by creating a method with the same name and arguments of the parent class is known as method overriding. It includes mutation operators are IOD, IOP and IOR.

- **IOD:** This operator deletes the entire declaration of an overriding method in subclass in order to reference the method use of the parent class version. This operator has probability of showing equivalence.

Original Code		**Mutant**
class Stack extends List {		class Stack extends List {
…….		……..
push (int a) {	*	/* push (int a) {
……		……
}		} */
}		}

- **IOP:** Sometimes an overriding method in a child class needs to call method it overrides in the parent class. It is essential to test that the method in the parent class is called at the right point in the program otherwise it would cause the incorrect state behavior.

Original Code		**Mutants**
class List {		class List {
…….		……
void SetEnv() {		void SetEnv() {
size = 5;		size = 5;
}		}
}		}
Class Stack extends List {		Class Stack extends List {
…		……
void SetEnv(){		void SetEnv() {
super.SetEnv();	*	size = 10;
size = 10;	*	super.SetEnv();
…		…
} }		} }

- **IOR:** This IOR operator is used to check the overriding method adversely affects other methods present. It renames the method being overridden in the parent class so that overriding method in the child class does not have an effect on the method in the parent class. This operator has probability of showing equivalence.

Original Code		**Mutants**
class List {		class List {
…….		……
void f() {….}	*	void f'() {…..}
void m() {		void m() {
f();	*	f'();
…..		…..
}		}
}		}
Class Stack extends List {		Class Stack extends List {
…….		……
void f() {….}		void f() {…..}
void g() {		void g() {
f();		f();
…..		…..
}		}
}		}

In case of overriding, super keyword allows the access of member variables and functions of the parent class. It includes mutation operators are ISI and ISD.

- **ISI:** This operator inserts the *super* keyword on variable or method so that it overrides references of method or variable. This operator has probability of showing equivalence.

Original Code	**Mutant**
Class Stack extends List {	Class Stack extends List {
int f() {	int f() {
…….	……

```
return val + num;                    *    return val +super.num;
        }                                         }
    }                                          }
```

- **ISD:** This operator deletes the *super* keyword on variable. It is to ensure the overriding methods are used properly. It works reverse of ISI. This operator has probability of showing equivalence.

Original Code		**Mutant**
Class Stack extends List {		Class Stack extends List {
int f() {		int f() {
…….		……
return val + super.num;	*	return val + num;
}		}
}		}

The constructors of the parent class are not inherited like other methods. Parent class default constructor is invoked before invoking its own constructor automatically whenever child class object is created.

- **IPC:** This operator deletes the parent constructor call so that the default constructor of the parent class is called. This operator has probability of showing equivalence.

Original Code		**Mutant**
Class Stack extends List {		Class Stack extends List {
Stack(int a) {		Stack(int a) {
super(a);	*	// super(a);
……		…...
}		}
}		}

1.3.2.3 Polymorphism

Polymorphism allows the objects to behave differently with the same method. It has the number of methods with the same name but different type and different execution.

- **PNC:** It changes the instantiated type of an object reference. It changes constructor initiation of object by reference to an object of a different type than that with which it is declared.

Original Code		**Mutant**
A a;		A a;
a = new A();	*	a= new B();

- **PMD:** This operator changes the declared type of an object reference to the parent of the original declared type. This operator has probability of showing equivalence.

Original Code		**Mutant**
B b;	*	A b;
b = new B();		b= new B();

- **PPD:** This operator is similar to PMD, except it changes the declared type of the parameter object reference to the parent of the original declared type.

Original Code		**Mutant**
boolean equals (B o) {….}	*	boolean equals (A o) {….}

- **PCI**: This operator changes an object reference to the parent/child of the original declared type. The type of the object reference change for overriding methods and hiding variables leads to exhibit different behavior. This operator has probability of showing equivalence.

Original Code		**Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
PRef.toString();	*	// ((Child)pRef).toString();

Method overloading allows two or more methods of same class having the same name but they have different argument signatures. It includes mutation operator is OMR, OMD and OAN.

- **OMR:** This operator replaces the body of a method with other method of the same name of a class. It is to check the overloaded methods are invoked correctly.

Original Code		Mutant
Child List {		Child List {
…. …		……
void Add (int e) {…….}		void Add (int e) {…….}
void Add (int e, int n) {		void Add (int e, int n) {
…….	*	this.Add(e);
}		}
}		}

- **OMD:** This operator is used to check the coverage of all overloaded methods and also deletes each overloading method one by one.

Original Code		Mutant
class Stack extends List {		class Stack extends List {
…….		……
void Push(int a){….}	*	// void Push(int a){….}
void Push(float a){….}		void Push(float a){….}
}		}
}		}

- **OAN:** This operator changes the order or number of arguments in method invocations. The number of arguments is changed in such a way that the new argument list is accepted by other overloaded methods. This operator has probability of showing equivalence.

Original Code		Mutants
s. push(0.5, 2);	*	s. push(2, 0.5);
	*	s. push(2);
	*	s. push(0.5);
	*	s. push();

1.3.2.4 Java-specific Features

This group the few object-oriented language features which do not occur in all OO languages. It includes mutation operators such as JTI, JTD, JSI, JID, JDC, EOA, EAM, EMM and JSD.

- **JTI:** This operator inserts *this* keyword to check the member variables are used correctly if they are hidden with method parameters.

Original Code		**Mutant**
class Stack {		class Stack {
int size;		int size;
……		……
void setSize(int size){		void setSize(int size){
this.size = size;	*	this.size = this.size
……		……
}		}
}		}

- **JTD**: This operator deletes the *this* keyword from the program. It works opposite to the JTI.

Original Code		**Mutant**
class Stack {		class Stack {
int size;		int size;
……		……
void setSize(int size){		void setSize(int size){
this.size = size;	*	size = size
……		……
}		}
}		}

- **JSI:** This operator adds the *static* modifier to change the instance variables to class variables. This operator has probability of showing equivalence.

Original Code		**Mutant**
public int s = 100;	*	public static int s = 100;

- **JSD:** This operator removes the *static* modifier. It works opposite to JSI. This operator has probability of showing equivalence.

Original Code		**Mutant**
public static int s = 100;	*	public int s = 100;

- **JID:** This operator deletes any initialization of variable declaration and class construction. This operator has probability of showing equivalence.

Original Code		**Mutant**
class Stack {		class Stack {
int size = 100;	*	int size;
Stack() {….}		Stack() {….}
}		}

- **JDC:** This operator deletes any implementation of default constructor created by the programmer. This operator has probability of showing equivalence.

Original Code		**Mutant**
class Stack {		class Stack {
…….		…….
Stack() {….}	*	// Stack() {….}
…….		…….
}		}

- **EOA:** This operator uses the Java *clone()* method. It replaces an assignment of a pointer reference with a copy of object using Java method.

Original Code		**Mutant**
Stack s1, s2;		Stack s1, s2;
s1 = new Stack();		s1 = new Stack();
s2 = s1;	*	s2 = s1.clone();

- **EAM:** This operator changes the accessor method name for another compatible accessor method name.

Original Code		**Mutant**
point.getX();	*	point.getY();

- **EMM:** This operator changes the modifier method name for another compatible modifier method name.

Original Code		**Mutant**
point.setX(1);	*	point.setY(1);

Chapter 2

Literature Review

Mutation testing is used to assess the quality of the test set on the bases of its capability to reveal simple faults injected in the program being tested. We want to achieve high mutation score. To achieve high mutation score, test set should be able to kill all the mutants, but some of the mutants are equivalent. A mutant is said to be equivalent, if it is semantically same however syntactically different from the original program. These mutants are contributing to increasing the computation cost. A number of approaches and techniques have been proposed for reducing computation cost and the detection of equivalent mutants are explained below:

2.1 Approaches

For reducing the efforts and cost, a number of approaches have been proposed in the mutation testing.

2.1.1 'Do Smarter' Approaches

These approaches [15] works by distributing the computation cost among several machines or by preserving some information. Such as compiled code, so that same code does not get generated repetitively. It avoids the execution of entire program. For example a program having 1000 lines of code, 5 mutants are created by changing the 8^{th} line. Instead of compiling a mutant number of times, the compiled form of original code is saved. As program's compiled form is changing the mutant for a single line only.

Weak Mutation: It compares the internal states of the mutated program and original program after the execution of both program mutated portion [15]. Mutant is killed if the state of mutated program is different from the original program. Otherwise the mutant is still alive.

Distributed Architecture: This approach [15] uses the strategy of dividing the computation cost over multiple machines. It uses vector processors, SIMD machines, Hypercube (MIMD) machines and Network (MIMD) computers for the mutation

analysis. It considers every mutant independent of other mutant and executes it individually.

2.1.2 'Do Faster' Approaches

These approaches [16] involve producing and executing every mutant as quickly as possible.

Schema-based Mutation Analysis: This system [15] creates the *metamutant* in which all mutants are encoded into one source program. Metamutant is once compiled and executed in same programming environment. Computation cost is saved by avoiding repetitive compilation and execution.

Separate Compilation Approach: This system [15] creates, executes and runs each mutant individually but not in interpretative style. It executes 15 to 20 times faster than interpretative style E.g. Proteum system.

2.1.3 'Do Fewer' approaches

These approaches [15] execute minimal set of mutants. A subset of mutants is chosen from all the mutants created in such a way that it is sufficient to determine a good set of test cases.

Selective Mutation: The selective mutation achieves maximum coverage by using selected and least number of mutation operators [16]. E.g. some mutation operators produce the same mutant are discarded at the time of creation.

Mutation Sampling: It uses, randomly selected subset of mutants for testing. If that subset is not sufficient, takes another subset [15]. The subset size is not priori fixed, it uses Bayesian sequential probability ratio test for appropriate subset sizing.

2.2 Techniques

A number of techniques have been proposed for reducing cost consumption and finding equivalent mutants. Some of these techniques are implemented by using some algorithms.

2.2.1 Technique using criteria's for reduction of cost and increase the relative strength of the mutation testing

Mutation testing requires number of test set for testing of code, but these test sets may not be good for revealing faults in the program. Cost consideration may requires the generation of only one adequate test set with respect to a given code. The cost of the mutation testing is the major bottleneck in its use by practitioners. To overcome the problem of cost, two alternatives are used in this paper which distinguishes equivalent and non-equivalent [17]:

1) **The randomly selected x% mutation criterion**: In this mechanism randomly select the x% of each type of mutant and ignore the remaining mutants. The percentage of selected mutants varies between 10% and 100%, and then keeps on increasing with constant value to examine the cost effectiveness of mutation testing using a different percentage of mutants.
2) **The constrained mutation criterion:** another mechanism examines only a few specified types of mutants and ignores the others. For example, the constrained *abs/sor* mutation, examines only *abs* and *sor* mutants.

During execution of mutation testing using above mentioned criteria, those test cases are discarded which are not able to distinguish at least one non equivalent mutant. A good selection of mutation operators can reduce the examination cost dramatically without sacrificing the fault detecting capability of mutation testing.

2.2.2 Automatically Detecting Equivalent Mutants and Infeasible Paths

The paper [18] presented a strategy using mathematical constraints and originally developed test set to automatically detect equivalent mutants and infeasible paths. The authors of this paper assume the problem of detecting equivalent mutants is a more generalized problem of feasible path problem. The feasible path problem specifies that some test requirements are infeasible because of the program semantics. These requirements cannot be fulfilled by the program.

This strategy the makes use of the constraint-based testing technique. In this technique a number of mathematical properties, these properties should be met by any test case. Three mathematical conditions are specified to kill a mutant:

1. **Reachability:** The test case must execute the mutated statement otherwise mutant never be killed.
2. **Necessity:** To kill a mutant, the test case must cause the mutant to have different state of mutant from the original program on the mutated statement.
3. **Sufficiency:** The test case must cause the final state of the mutant to be different from the original program.

The constraints are applied to the program with the help of mathematical conditions. The constraints are used to determine the equivalent mutants.

2.2.3 An Evaluation of Selective Mutation

These papers [19, 20, 21] use the selective mutation. It is a way to saves execution cost by reducing the number of mutants that need to be executed. They show the effectiveness of selective mutation testing over non-selective mutation testing. As, the major computation cost of mutation testing is incurred when all the mutant programs runs against the test cases. The number of mutants generated for a program is roughly proportional to the product of the number of data references and the number of data objects. To perform selective mutation testing, firstly create the selective mutants for program to test. Then kill the mutants as much as possible and measure the effectiveness of results [19]. In selective mutants, efficient operators are used for the generation of mutants [21]. Only those mutation operators are expected, which generate mutants that require the construction of test cases. Those test cases are able to kill not only the mutants of the given operator but also mutants of other operators at a low cost are called effective test cases [21]. Goal of selective mutation is to use of operators that tend to produce mutants having semantically small faults [20]. These results signify that the use of efficient operators can provide efficiency for selective mutation [21]. It has almost the same coverage as that of non-selective mutation with significant reduction in cost.

2.2.4 Mutation Clustering

As more cluster [22, 23] the mutants, the more lowers the computation cost. By selecting mutants from each cluster, the size of the mutants gets reduced. Further, generate a test set that is mutation adequate for the mutants selected from the clusters [22]. It integrates

data clustering and mutation analysis, such that both mutant set and test set can be reduced dramatically [23].

The clustering algorithms used are *k*-means clustering algorithm and the agglomerative clustering algorithm. *K*-means algorithm has number of variations; one of these variations is to select good initial values as the *k* means clustering depends heavily on the initial selection of the *k* values. The agglomerative algorithm depends on a threshold value. The data objects are merged pair wise if the distance between them is below the threshold value. All the mutants are represented by the domain of variables to determine the centroid of a cluster [23]. One of the mutants from each cluster is selected randomly for both algorithms and generates the test set for selected mutant by using greedy approach [22]. It has CBT and DDR methods for the generation of test cases automatically [23]. Both of them focus on the domains of variables. CBT takes the algebraic expressions as constraint and DDR takes an initial set for each input [23]. That test set used in killing those mutants. Keeping that test set fixed, generate a new test set of the same size for other mutants in the cluster. Lastly both the test sets are observed and efficiency of each test set is analyzed based on the mutation score [22]. Those mutants are clustered together, subsequently reducing the size of mutants.

2.2.5 Mutation Analysis Using Mutant Schemata

This paper [24] performing mutation analysis with the use of program schemata to encode all mutants of a program into one metaprogram, which is compiled and run at a speed substantially higher than the previous interpretive systems. Preliminary studies indicate that 300% performance improvement has been reported. This method has additional advantage that it is easier to implement than interpretive systems, because it uses the same compiler and run-time support system during development. Rather than mutating an intermediate form of the program, it must be interpreted. Mutant Schema Generation (MSG) method enables to encode all mutations into one source level program. Then this program is compiled with the same compiler, used during development and is executed in the same operational environment at compiled-programs speed. While mutation systems based on mutant schemata, need not to provide the run-time semantics and environment. These are considerably less complex and easier to build.

A mutant schema has two components, metamutant and metaprocedure set, both are represented by syntactically valid constructs. Metaprocedure is a function that corresponds to an abstract entity in the schema. A statement that has been changed to reflect such a generic form is said to be metamutated. A metamutation is a syntactically valid change that embodies other changes. The use of mutant schemata significantly speeds up the mutation analysis. MSG systems are smaller and easier to build, allows to quickly developing mutation tools for a variety of languages. MSG systems also provide more realistic testing. The program tested in the MSG systems retains all or most of its original operational behavior.

2.2.6 Sufficiency of Mutants

This paper [25] addresses, instead of considering the large number of mutation operators, considering a smaller subset of mutation operators is sufficient. That smaller subset can model the behavior of a full set. For this, they proposed statistical techniques for model selection, variable reduction and nonlinear regression. The techniques are:

1. **Model Selection Algorithms:** These algorithms construct and search all possible best-fitted linear models that attempts to predict the detection ratio of the mutants (AM) more accurately. Choose a set of possible predictors having the highest absolute correlation with AM and fitting a linear regression between them.
2. **Variable Reduction Techniques:** These techniques are used to reduce the number of predictor variables for the smaller set by avoiding less important predictors. Repeatedly, identifing a pair of highly similar variables and eliminate the one that generates more mutants, stop when all pairs fall below a certain correlation threshold.
3. **Nonlinear Regression Approach:** Transforming the data is the main technique for revealing a nonlinear relation. By nonlinear regression, the construction of a complementary set of nonlinear models is done in addition to the linear ones.

2.2.7 Decreasing the cost of mutation testing with second-order mutants

This article [26] has presented a technique to decrease the cost of mutation testing by means of the combination of first-order mutants. The idea is based on the reduction in the number of mutants by combining mutant pairs and generating into new mutants. Each

mutant is created with the introduction of two faults in the mutants, from where they come. Results lead to believe that mutants' combination does not decrease the quality of the test suite, whereas it assumes important savings in mutant execution and result analysis. Three combination strategies have been implemented by the testooj tool are *LastToFirst*, *DifferentOperators* and *RandomMix*. Thus reducing the number of mutants reduced to half of the original suite implies a significant reduction in the effort devoted to mutant execution and result analysis.

2.2.8 Using Higher Order Mutation for Reducing Equivalent Mutants in Mutation Testing

First order mutants used for mutation testing have proved to be highly cost consuming in terms of human effort over the years. Higher order mutation testing was introduced, and it was proposed to be more effective in dealing with the mutation testing challenges. Although higher order mutation generates more number of mutants than the first order mutation, the number of equivalent mutants is largely reduced. This paper [27] reports higher order mutation reduces the number of equivalent mutants' generation, depending upon the order of mutation by comparing the degree of equivalent in the second order and random order mutation.

First order mutants are generated by using Milu tool and randomly select some of them. Evaluate FOM and check if they are equivalent or not. Then increase the order of mutation for the creation of mutants. Similarly, some of HOM are selected randomly and find if they are equivalent or not, till the results have zero percent equivalence.

Then second order and random order mutants are generated and randomly select approximately half of them. Again evaluate HOM for equivalency. Test cases used for their evaluation are used same in FOM and HOM. HOM testing is truly capable for reducing the number of equivalent mutants as proposed in the previous work.

2.2.9 Higher Order Mutation Testing

This introduces the concept of subsuming Higher Order Mutants (HOMs) [28, 29], one that is harder to kill HOM than the first order mutants (FOMs) from which it is constructed. Subsuming HOMs denotes the subtle fault combinations. It is preferable to

replace the FOMs with the single HOM. It also removes the number of myths related to the mutation testing [29]. A strongly subsuming HOM is only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed. The set of test cases that kill HOM also kill each and every FOM. Fewer (but better) mutants mean requirement of fewer (but better) test cases [29].

This paper introduced the search based approach for the identification of efficient subsuming HOMs. This approach also overcomes the exponential explosion in the number of HOMs. The algorithm targets subsuming up all HOMs rather than searching for strongly subsuming HOMs. It uses greedy algorithm, a genetic algorithm and a hill climbing algorithm. The result indicates that the performance of genetic algorithm is best among all of them. However, others also improve the quality of results [28]. The higher order mutation testing may turn out to be far more scalable than first order mutation testing.

2.2.10 Using Constraints to Detect Equivalent Mutants

This technique [30] uses contradiction of the constraint system for the detection of equivalent mutants. In this paper, three broad strategies negation, constraint splitting and constants comparison are defined for the reorganization of infeasible constraints in the systems.

Negation: In this strategy two constraints are negation of each other if they belong to different and non-overlapping domain. If two constraints are syntactically or semantically equal it means they belong to the same domain.

Constraint splitting: In constraint splitting, two new constraints are generated from two given constraints. Then two new constraints are compared with one of the given constraint. If they are conflicting, the other given constraint will also be conflicting. Thus, the accuracy of the “constraint splitting" strategy is proved.

Constant comparison: A third strategy works for those constraints having a format (V *rop* T, where V is a variable, rop is a relational operator, and T is a constant). Also the variables in both constraints are required to be same. Both constraints are compared on the basis of their constant and a prediction is done. If possible on the basis of prediction, come to know that both the constraints are conflicting or not.

2.2.11 Using Program Slicing to Assist in the Detection of Equivalent Mutants

Mutation testing is difficult to apply to large programs because detection of equivalent mutants becomes more tedious and almost impossible. In this paper [31] the creation of program slices makes detection of equivalent mutants easier.

The strategy working in this paper uses weak mutation. A new boolean variable lets say z, is introduced in the original program p with its initial value set to be true. Then the slice of this variable is created, to determine that the mutant has been killed or not. Every time when program runs, node having variable z must be iterated. If the mutant has been killed the value of z becomes false, otherwise it remains true. If the value of z remains true at the end of the execution, it indicates equivalence.

The authors also discussed that the creation of amorphous slices may reduce the problem of detecting equivalent mutants even more by applying further transformations to the traditional slices. An interesting observation stated in the paper is an ideal amorphous slicing algorithm would yield this slice for all equivalent mutants, and the problem of equivalent mutant detection would disappear.

This paper discusses that the use of program slices can ease the process of detection of equivalent mutants and reduces the work carried out for identifying them manually. Also, it has been suggested that equivalent mutants will always create slices which are identical for all mutants.

2.2.12 Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution

This technique [32] uses the genetic algorithms. Genetic algorithms help in reducing the problem of generating a large number of mutants as well as the problem of equivalent mutant detection. This technique helps in attaining selective mutation without decreasing mutation operators applied to the programs. This technique work on the following three operations:

1. **Evolution of subsets of mutants against a fixed set of test cases:** Mutants are created and validated against a fixed set of test cases. The mutation score is generated, indicating the performance level of mutants against that particular set of test cases. A low score indicates the mutants are difficult to kill. Then the subsets of

these mutants are created. Non-equivalent mutants with the higher adequacy score are selected for subset creation. Genetic algorithms are used for the evaluation of this test set. The fitness function used to ensure the selected mutants are definitely killed by some test cases.

2. **Evolution of subsets of test cases against a fixed set of test mutants:** The same strategy is used for the creation of subset of test cases. Test cases are generated randomly. Mutation score is then assigned to each set of test cases by executing them against a fixed set of mutants. The mutation score represents the performance of test set against a given set of mutants. Again, each test set is evaluated using a fitness function; it results in another set of test cases which, when executed against a set of mutants gives a higher adequacy score.
3. **Combine the above two sets using co-evolution:** The two populations generated above are combined by using a fitness function. The mutation score of each mutant is re-evaluated with respect to the present population of test cases. Similarly, the score of every test case is re-evaluated with respect to the present population of the mutants.

This strategy ensures that equivalent mutants are not generated due to the ingenious design of the fitness function. The fitness function assigns an adequacy score of zero to all the mutants which are difficult to kill as they are more likely to equivalent mutants. Only mutants and test cases with higher adequacy scores are selected.

2.2.13 Using Compiler Optimization Techniques to Detect Equivalent Mutants

The technique [33] presented in this paper uses the compiler optimization techniques. Equivalent mutants can be detected by the use of code optimization algorithms. In this paper there are six techniques proposed, that are following:

1. **Dead code detection:** Mutants are created by the change in dead code. Such mutants will never be executed or are irrelevant. They will not affect the output of the program, thus mutants are equivalent.
2. **Constant propagation:** It creates a constant table, which keep the entries of variables with their constant definitions. Mutants that are still alive at the end of the execution

of a test case can be checked for equivalence using this constant table. A mutant that has an entry in this table will be declared as equivalent.

3. **Invariant propagation:** This technique has a table having the relationship between 2 variables. The table is called an invariant table. An equivalent mutant will have the identical definitions in the invariant table, thus not changing the value of the variable in the mutated program form that in the original program.
4. **Common sub-expressions**: This technique does not identify equivalent mutants directly. It is used in combination with other compiler optimization techniques. It will keep the track of all the temporary variables created during compilation and determines the equality of relationships between variables if any.
5. **Loop invariant detection:** An equivalent mutant changes the boundary of a loop by moving it inside or outside, while code compilation.
6. **Hoisting and sinking:** This works similarly to loop invariant detection. Equivalent mutants will generate the identical results regardless of hoisting or sinking the code.

2.2.14 Detecting Equivalent Mutants by Means of Constraint Systems

Mutation testing is useful for evaluating the quality of the test suite. In this technique constraint system is used for the generation of test scenarios which are able to distinguish between two different versions of a program. They start with a hypothesis which states that when the constraint system is not able to find any solution, it might be the case of encountering two equivalent mutants. The tool used for computing mutants is MuJava and for solving constraints is MINION. This strategy is applicable to medium size applications.

The algorithm [34] firstly translates the original program into a constraint system. Then is elimination phase, in which some of the mutants are eliminated from the set of mutants and a new set of mutants is generated. The elimination phase ends when the system has one or more solutions or none. In case constraint solver is not able to find even one solution, that mutant is considered semantically equivalent. Constraint system also delivers the test suite having a high mutation score than the original one.

2.2.15 Using Constraints for Equivalent Mutant Detection

This paper [35] proposed a method for solving the equivalent mutant problem using a constraint representation of the program and its mutant. The approach distinguishes test cases that force the program and its mutant to behave in a different way. The number of iterations of program for the generation of loop-free program is known in advance.

This approach checks the feasibility by giving constraint system mutants and test cases as input. If the test cases are feasible, the mutants are not equivalent. Otherwise other distinguishing test cases are searched. Second problem due to the number of iterations is considered during conversion constraint representation. Due to the nesting depth, the constraint solver at times returns no solution. Then, there is need of searching again. In that case, method nesting depth is set to pre-defined maximum value.

A tool named EqMutDetect [36] uses the above method for the detection of equivalent mutants in the embedded systems. Verification and validation of embedded system domain are very important.

2.2.16 Isolating First Order Equivalent Mutants via Second Order Mutation

This paper proposed a technique [37] named I-EQM. It works by employing dynamic execution of integrated first and second order mutation. This approach combines the impact on the program execution of the first order mutants with the impact on the output of second order ones, to isolate the first order equivalent mutants. The idea of this approach is to determine the impact of a first order mutant on other already killed mutants.

This automated technique is able to effectively classifying the mutants which are being killable or equivalent. It classifies correctly with precision and recall different mutants with the empirical approaches. The precision metric has the ability to categorize correctly the killable mutations. High precision indicates the capability to distinguish between killable and equivalent mutants. The recall metric measures the capability in retrieving killable mutants. A high recall shows the ability to recover the majority of live killable mutants. It improves the previously proposed approaches by selecting a considerably higher number of killable mutants with only limited loss on the precision.

2.2.17 Un-Covering Equivalent Mutants

This paper [38] uses the same methodology of [37], to examine whether changes in coverage can be used to detect non-equivalent mutants. The paper shows about 45% of all undetected mutants turned out to be equivalent. Manual classification takes 15 minutes per mutation approximately. Two main problems covered in this paper are the repeated execution of test suites requiring significant computing resources and the possibility of a mutation leaving the program's semantics unchanged. This paper examines the impact of mutations on coverage, whether lines are executed or not. The framework used for the experiment is JAVALANCHE. The paper proved that the equivalent mutants' coverage remains unchanged whereas non-equivalent mutants actually changed the coverage. The result is 75% of the mutants are correctly classified based on their impact on coverage. This means technique significantly reduces the effort for mutation testing and makes easy to deploy by using coverage measurement tools.

This paper [39] identified equivalent mutants and investigated the impact of a mutation on the execution. They assessed the problem of equivalent mutants similarly [38]. Assessing mutation shows equivalence in time and effort consuming. That makes the mutation testing prohibitive even for small programs. They discovered that mutations that alter the dynamic control flow are less likely to be equivalent. Thus, higher the impacts lower the chance of equivalence.

Chapter 3

Problem Analysis

3.1 Problems in Mutation Testing

Software testing is an important and critical part of the software development process and confirms the reliability and quality of the software. It also increases confidence in proper functioning and assists the evaluation of functional and nonfunctional properties of software. However, 50% of the cost and time is spent on testing the software thoroughly [2]. For this purpose a number of testing techniques have been developed. One of the problems related to these testing techniques is the generation of effective test data. By solving this problem software testing cost is reduced significantly. As mutation testing is fault based testing technique, it accesses the effectiveness of test data. The main goals of mutation testing are to provide a test adequacy criterion and diagnose faults in the software. It works [29] on the assumptions of competent programmer and coupling effect hypothesis. But the mutation testing suffers from the problem of the huge cost and effort consumption. The main reason of this cost consumption is the generation and execution of a large number of mutants. Even a small program might have hundreds of mutants. For example, a program having less than 100 KLOC will generate an enormous number of mutants [40]. The Java quadratic equation program with 25 LOC generated 361 mutants. No doubt creating test cases and executing all these test cases on every mutant can be computationally heavy. Each mutant must be executed at least once by the test cases, which leads to too much computation. Additionally, in case of equivalent mutant all the test cases are executed against that mutant in an attempt to kill it. Wong [40] says that the cost can be measured by two metrics, the size of the test set used to fulfill the criterion and the number of the mutants under observation.

3.2 Problem Statement

This testing technique has large number of mutants, the use of this technique means running each and every mutant. Thus we can say that this approach requires high labor and cost for the execution of all the mutants. In case of equivalent mutant, detection is hard and very costly because of running and re-running all the test data in an attempt to

kill them. It means achieving a correct mutation score is costly and large effort consuming.

3.3 Our objectives

- Propose an approach that grouping the mutants to reduce the number of mutants. Grouping the large number of mutants and executing selected mutants from each group, considerably lowers the computation cost.
- Approach reduces the size of the mutants. Furthermore, it generates a test set that is mutation adequate for the mutants selected from the groups. Thus, that test set will be efficient in killing all the actual number of mutants.
- Propose an approach that detects equivalent mutant. This makes the testing ease with less effort consuming.
- Reduce the cost, effort and time for the execution of mutation testing and detect the equivalent mutants.
- Approach uses fewer mutants for execution that achieves same results and accurate mutation score as the detection of equivalent mutants.

Thus we will group the mutants and execute selected mutants, which reduce the cost, effort and time consumption. Additionally, the equivalent mutants are detected leading to calculation of accurate mutation score.

Chapter 4

Proposed Approach

In this chapter we discuss about the proposed technique to solve the problem stated in literature.

The proposed technique helps to solve the problem arises in mutation testing, which are equivalent mutant detection and large effort, time and cost consumption.

Table 4.1: Mutants illustrates Equivalence [11]

Mutant Type of Java	***Category***	***Operator***	***Probability of showing equivalence***
Method level Mutation Operator	Arithmetic	AOI_U	80%
		AOI_S	29.17%
Class level Mutation Operator	Encapsulation	AMC	50%
	Inheritance	IHD	71.43%
		IHI	80%
		IOD	5.88%
		IOR	100%
		ISI	100%
		ISD	100%
		IPC	100%
	Polymorphism	OAN	50%
		PMD	100%
		PCI	100%
	Java-specific Features	JSI	80%
		JSD	80%
		JID	100%
		JDC	100%

4.1 Proposed Technique

In the proposed technique, first equivalence of mutants is determined and mark them equivalent. In second step, groups of mutants having mutation operator at different locations under similar conditions in a program code are created.

4.1.1 Equivalent Mutant detection

Following are the reasons for the occurrence of equivalent mutants in a program:

- A small change in the syntax of the program but the semantics remains the same [40].
- No effect of syntax change on the program as per program pre-conditions [39].
- Change is done in dead code [21].
- Infeasible constraint created by mutation operator [21].

Table 4.1 contains the mutant having probability of showing equivalence. We found among these reasons some mutation operators under particular conditions contribute its mutant to behave equivalently. Presence of a mutant in a program is indicated by *. Following mutation operators and corresponding conditions behaves them equivalently:

4.1.1.1 Arithmetic Operator Insertion of unary Operator(AOIU)

- **Plus operator:** Inserting + operator in front of any variable or if already one + operator is present in front of variable inserting another + operator with space keeps semantics same of the mutant.

Original Code		**Mutants**
a= b + c;	*	a= +b + c;
	*	a= b + +c;

4.1.1.2 Arithmetic Operator Insertion of short-cut Operator(AOIS)

- **Post operator (++, --):** Variable on which operator is applied, is not used later in its block, keeps overall results of the mutant and its original program same.

Original Code	Mutants
{	{
a= b + c;	a= b + c;
System.out.println(a);	* System.out.println(a++);
	* System.out.println(a--);
}	}

4.1.1.3 Access Modifier (AM): When change from more stringent to less stringent access modifier operator, detection of mutation operator is not possible. Following such changes are:

- Private to protected
- Private to public

Original Code	Mutants
private Stack s;	* public Stack s;
	* protected Stack s;
	* Stack s;

- Protected to public

Original Code	Mutants
protected Stack s;	* public Stack s;
	* Stack s;

4.1.1.4 Inheritance

- **Hiding variable insertion and deletion (IHI and IHD):** When once an access is given to child class variable and attempt to access parent class is made, compiler will not be able to detect variable hiding.
- **Overriding method deletion and rename (IOD and IOR):** In case functionality of method present in the child class is similar to the method of the parent class, the output remains same.

Original Code	Mutants
class List {	class List {
…….	……

```
void f() {…abc…}                    *    void f() {…abc…}
void m() {                               void m() {
f();                                *    f();
…..                                      …..
}                                        }
}                                        }
Class Stack extends List {               Class Stack extends List {
…….                                      ……
void f() {…abc…}                         void f() {…abc…}
void g() {                               void g() {
f();                                     f();
…..                                      …..
}                                        }
}                                        }
```

- **Super keyword insertion and deletion (ISI and ISD):** Use of these mutant operators always shows equivalence.

```
Original Code                            Mutant
Class Stack extends List {               Class Stack extends List {
…….                                      ……
 int f() {                               int f() {
…….                                      ……
 return val + num;                   *    return val +super.num;
 //  return val + super .num;        *    return val +num;
          }                                      }
  }                                       }
```

- **Explicit call to parent's constructor deletion (IPC):** Use of this mutant operator always shows equivalence.

```
Original Code                            Mutant
Class Stack extends List {               Class Stack extends List {
  …….                                     ……
  Stack(int a) {                          Stack(int a) {
```

super(a);	*	// super(a);
}		}
}		}

4.1.1.5 Polymorphism

- **Arguments of overloading method call change (OAN):** Mutants are not detected if the overloading method uses default argument values.

Original Code		**Mutants**
s. push(0.5, 2);	*	s. push();
	*	s. push(0.5);

- **Member variable declaration with the parent class type (PMD):** When the mutant changes a constructor to its any base class constructor, the output remains same.

Original Code		**Mutant**
B b;	*	A b;
b = new B();		b= new B();

- **Type cast operator insertion (PCI):** Use of this mutant operator always shows equivalence.

Original Code		**Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
PRef.toString();	*	// ((Child)pRef).toString();

4.1.1.6 Java specific features

- **Static modifier insertion (JSI):** In case variable is used only once in the program.

Original Code		**Mutant**
main()		main()
{		{
public int s = 100;	*	public static int s = 100;
System.out.println(s);		System.out.println(s);

```
//program end                                    //program end
}                                                }
```

- **Static modifier deletion (JSD):** In case variable is used only once in the program, and it is not using the default value.

Original Code		**Mutant**
main()		main()
{		{
public static int s = 100;	*	public int s = 100;
System.out.println(s);		System.out.println(s);
//program end		//program end
}		}

- **Member variable initialization deletion (JID):** Use of this mutant operator always shows equivalence.

Original Code		**Mutant**
class Stack {		class Stack {
int size = 100;	*	int size;
Stack() {….}		Stack() {….}
}		}

- **Java- supported default constructor creation (JDC):** Use of this mutant operator always shows equivalence.

Original Code		**Mutant**
class Stack {		class Stack {
…….		…….
Stack() {….}	*	// Stack() {….}
}		}

4.2 Proposed Algorithm

In the proposed technique after the detection of equivalent mutant, groups of mutants are created. Groups are created of the mutants having mutation operator at different locations under similar conditions in a program code. Mutants under similar conditions having same mutant operator are grouped in one group. For example, a variable in print

statement and return statement behaves similarly but in an arithmetic statement and conditional statement behaves differently. One of the groups may contain all the mutants formed using a mutation operator on the variables which are present in print and return statements. Similarly another group may contain all the mutants formed using a mutation operator on the variables which are present in all conditional statements. We can execute only one mutant from every group. The result of single mutant whether killed or live is assumed to be same for all other mutants in its group. These results are used in calculating the overall results for the entire program code. The calculation is done by multiplying the number of mutants present in its group and then summing up all group results. Using the proposed technique, similar results are obtained as the existing approaches by executing lesser mutants. Additionally, proposed approach takes fewer efforts, cost and time consumption.

4.2.1 Algorithm: Similar Behavior Mutation Testing (SBMT) Algorithm

Input: Original program and Test suit.

Output: Mutation score.

1. Remove the infeasible path and dead code from the code.
2. Generate a set of mutants using all constraints and necessary conditions then
   - Analyze mutant domain and mark equivalent mutant those following above mentioned reasons for equivalence.
   - Create G_i groups where $i=1$ to $i=n$. Each group is having N_j mutants where $j=1$ to $j=m$.
   - $Killed\_Mu\tan t \leftarrow Live\_Mu\tan t \leftarrow 0, j \leftarrow 1$
3. For $p=1$ to $p=n$ do

   For each test case from the test suite

   Apply current test case on Mutant N_j of G_p

   If ($result = killed$)

   $Killed\_Mu\tan t \leftarrow Killed\_Mu\tan t + N_m$

   Else

   $Live\_mu\tan t \leftarrow Live\_mu\tan t + N_m$

End If

4. If results are not satisfactory, then

   If $(j < m)$

   $j \leftarrow j + 1$

   Go to step 3.

5. $Mutation\_score \leftarrow Killed\_Mu\tan t / Live\_Mu\tan t$

The above algorithm SBMT execute on a single mutant from each group will produce result similar to the actual results. This algorithm works in conjunction with the traditional algorithms and reduces the efforts for performing mutation testing.

Chapter 5

Experimental Results

This chapter contains the implementation details of the SBMT algorithm and the experiment results of this approach.

5.1 Implementation Details and Experimental Results

The proposed algorithm is applied on nine Java programs [30, 42, 43] that cover distinct types of applications. Table 5.1 contains the description of these Java programs. Code length of these Java programs varies from a small number of statements to a large number of statements. All possible method level and class level type mutants for Java are created for these nine programs. Method level operators are arithmetic operators, conditional operators, shift operators, logic operators, relational operators and assignment operators and class level operators are applied on object-oriented features such as encapsulation, polymorphism, access modifier, inheritance and java-specific features [11].

Table 5.1: Experimental Programs Description

Program	***Description***
Quad	Finds the root(s) of a quadratic equation.
Insert	Sorts an array of integers using insertion sort
Warshall	Calculates the transitive closure of a Boolean matrix.
Bsearch	Binary search on an integer array to search an integer.
Bub	Bubble sort on an integer array.
Trishmall	Classifies a triangle as equilateral, isosceles, scalene, or illegal.
Mid	Returns the middle value of three integers.
Euclid	Euclid's greatest common divisor algorithm.
Pat	Returns the starting and last position of a pattern if it is in the subject.

The above mentioned reasons contributing in the creation of equivalent mutant are analyzed in these programs. Results are determined manually for the detection of equivalent mutants. All the mutants are analyzed; those mutants following them are marked as equivalent. Thus, the chances of mistakes are quite rare.

Table 5.2: Results of proposed mutation operators and conditions for deporting equivalent mutants

Program	***Proposed mutation operators and conditions for deporting equivalent mutants***	
	Total Mutants	***Equivalent Mutants***
Quad	361	40
Insert	363	41
Warshall	307	33
Bsearch	300	28
Bub	296	33
Trishmall	392	79
Mid	181	23
Euclid	158	25

Table 5.2 contains the results of above mentioned mutation operators and their conditions for equivalent mutants. No execution of mutants is done on the collection of these results. Hence the detection of equivalent mutant is easy and less effort consuming. Later in this chapter, these results are proved by executing them and comparing the results with existing techniques.

In table 5.3, detail of the experiment results has been reported. The approximate true results of the programs are calculated on executing three mutants from each group. The number of groups of mutants for a program varies from program to program depending upon many factors. The large number of mutants in a group has no adverse effect on the result rather it facilitates in reducing the effort of calculation.

Table 5.3: Mutation testing results using SBMT

Program	Executed Mutant	Killed Mutant	Approx. Total Killed	Total %of Executed	Total %of Killed
Quad	192	152	277	53.185	100
	297	227	275	77.285	99.27
	361	277	277	100	100
Insert	173	138	277	47.658	98.16
	281	218	271	77.41	99.63
	344	262	272	94.76	100
	363	272	272	100	100
Warshall	94	74	252	30.61	93.33
	152	120	251	49.61	93.64
	210	159	234	68.40	99.15
	268	205	236	87.29	100
	307	236	236	100	100
Bsearch	139	95	217	46.33	96.875
	212	151	227	70.66	98.66
	262	193	227	86.66	98.66
	300	224	224	100	100
Bub	151	124	244	51.01	98.75
	219	179	242	73.98	99.58
	269	219	242	90.87	99.58
	296	243	241	100	100
Trishmall	195	127	241	49.74	98.36
	355	231	245	90.56	100
	382	245	245	97.44	100
	386	245	245	98.46	100
	390	245	245	99.48	100
	392	245	245	100	100
Mid	64	51	140	35.35	98.55
	128	100	138	70.71	100
	154	120	138	85.08	100
	163	126	138	90.05	100
	172	132	138	95.02	100
	181	138	138	100	100
Euclid	99	69	110	62.65	98.214
	132	92	111	83.54	99.107
	158	112	112	100	100
Pat	68	54	106	53.125	100
	128	106	106	100	100

In figure 5.1, every program delineates the execution of the number of mutants from their groups and the corresponding percentage of detected killed mutants from total killed mutants for the entire program code. Figure 5.1 illustrates that the expected result of the programs are procured only by considering four mutants from each group of every program. Hence if, any group has mutant count more than four, there's no need to execute those excess mutants. On the execution of a single mutant from every group, the results are more than 93 percent close to required results and consumption of effort and time is less than 60 percent.

1st Execution - one mutant is executed from every group
2nd Execution- two mutants are executed from those groups having two or more mutants
3rd Execution- three mutants are executed from those groups having three or more mutants
4th Execution- four mutants are executed from those groups having four or more mutants

Figure 5.1: Execution of number of mutant's Vs %killed mutant detected of total killed mutants

5.1.1 Graphical Representation

The results obtained on executing the test cases against the mutants of program Quad, are shown in figure 5.2. These results show that on the execution of single mutant from each group 100% results are obtained for the program. Hence, there is no need to execute rest of the mutants of the program. Mutants are executed less than 54% of the total number of mutants.

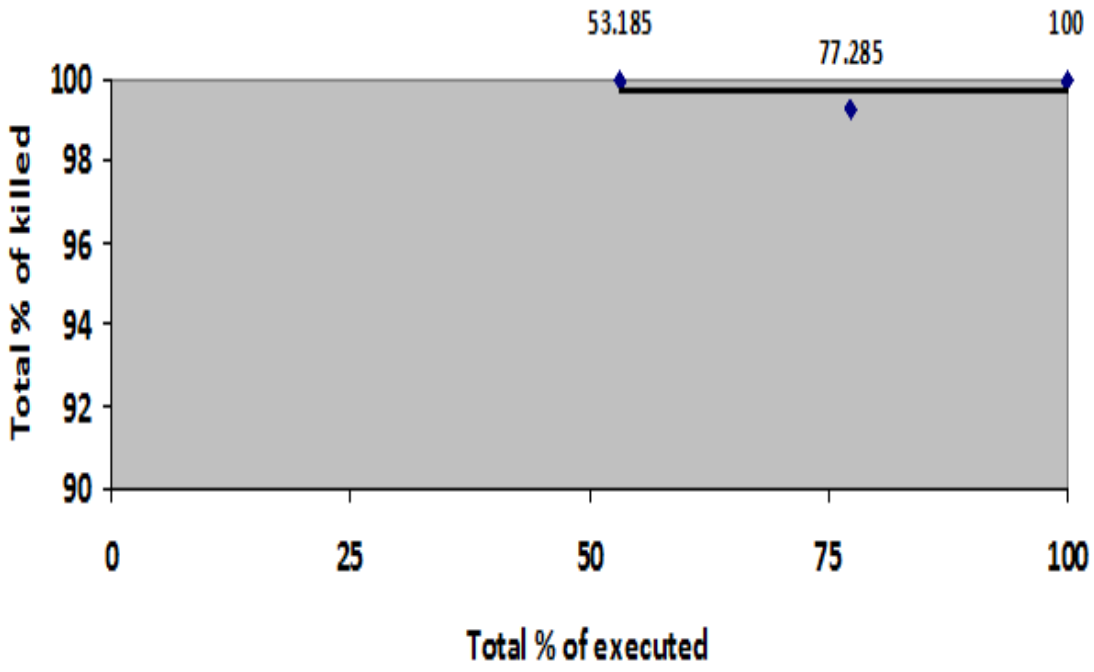

Figure 5.2: Quad program

The results obtained on executing the test cases against the mutants of program Insert, are shown in figure 5.3. These results show that on the execution of single mutant from each group more than 98% results are obtained for the program. On the execution of three mutants from each group 100% results are obtained. Hence, there is no need to execute rest of the mutants of the program. Number of mutants required for execution is less than the total number of mutants.

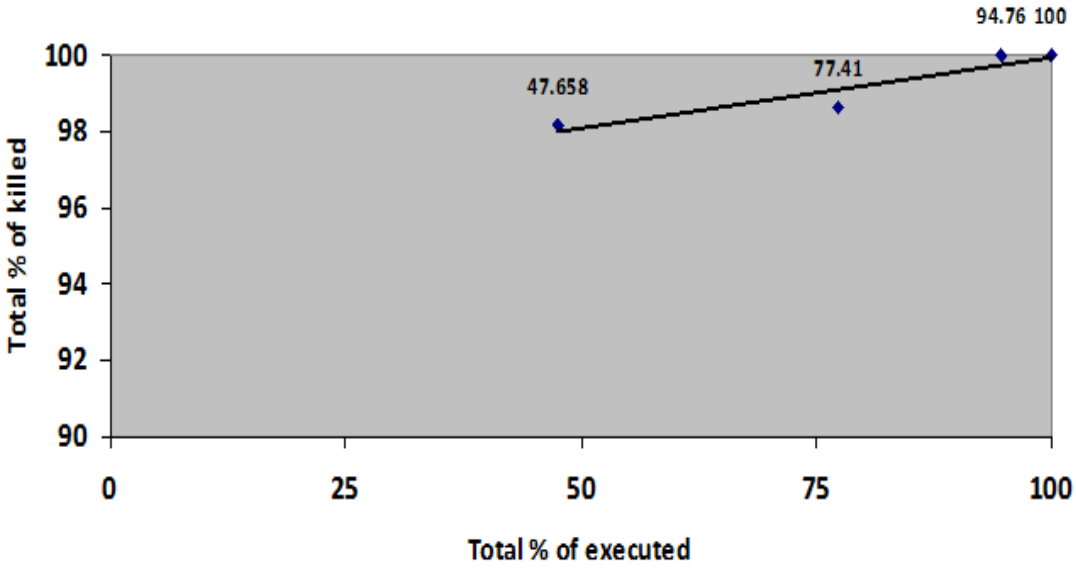

Figure 5.3: Insert program

The results obtained on executing the test cases against the mutants of program Warshall, are shown in figure 5.4. These results show that on the execution of single mutant from each group more than 93% results are obtained for the program. On the execution of four mutants from each group 100% results are obtained. Hence, there is no need to execute rest of the mutants of the program. Number of mutants required for execution is less than the total number of mutants.

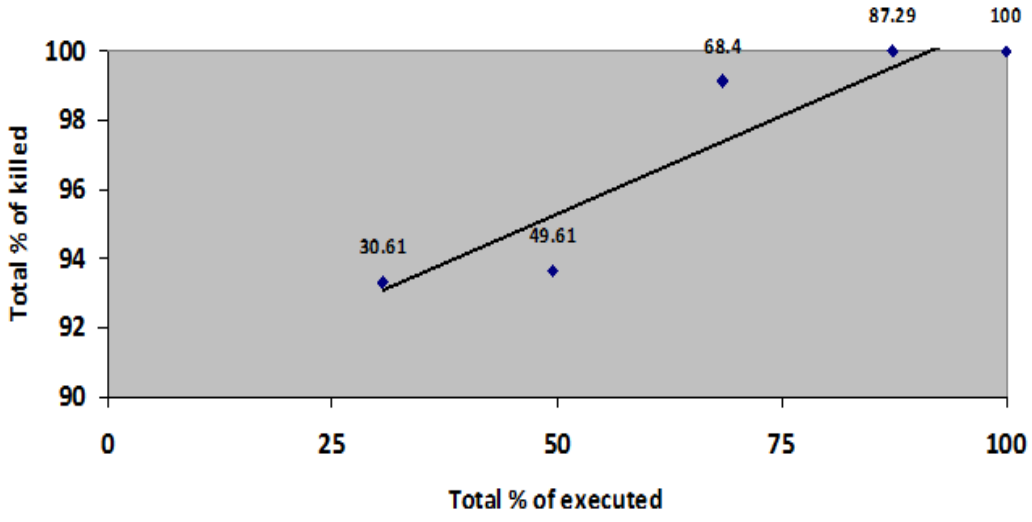

Figure 5.4: Warshall program

The results obtained on executing the test cases against the mutants of program Bsearch, are shown in figure 5.5. These results show that on the execution of single mutant from each group more than 96% results are obtained for the program. On the execution of four mutants from each group 100% results are obtained. But we get satisfactory results on the execution of two mutants from each group. Hence, no need to execute rest of the mutants of the program.

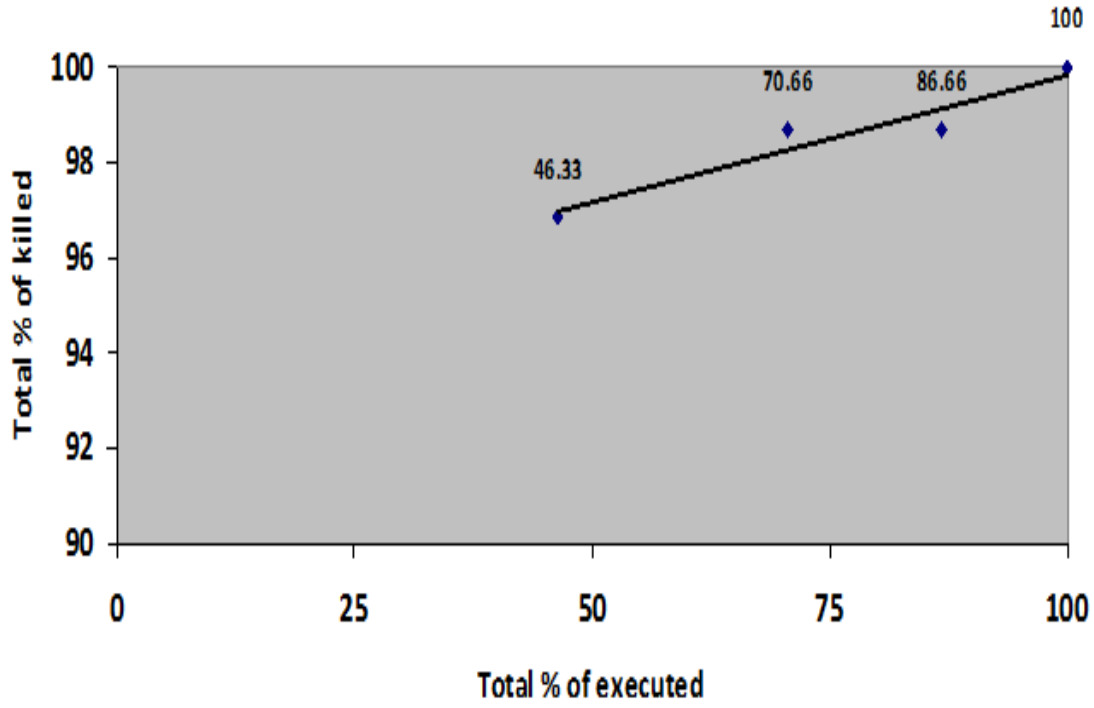

Figure 5.5: Bsearch program

The results obtained on executing the test cases against the mutants of program Bub, are shown in figure 5.6. These results show that on the execution of single mutant from each group more than 98% results are obtained for the program. On the execution of four mutants from each group 100% results are obtained. But we get satisfactory results on the execution of two mutants from each group that is 99.58%. Hence, no need to execute rest of the mutants of the program.

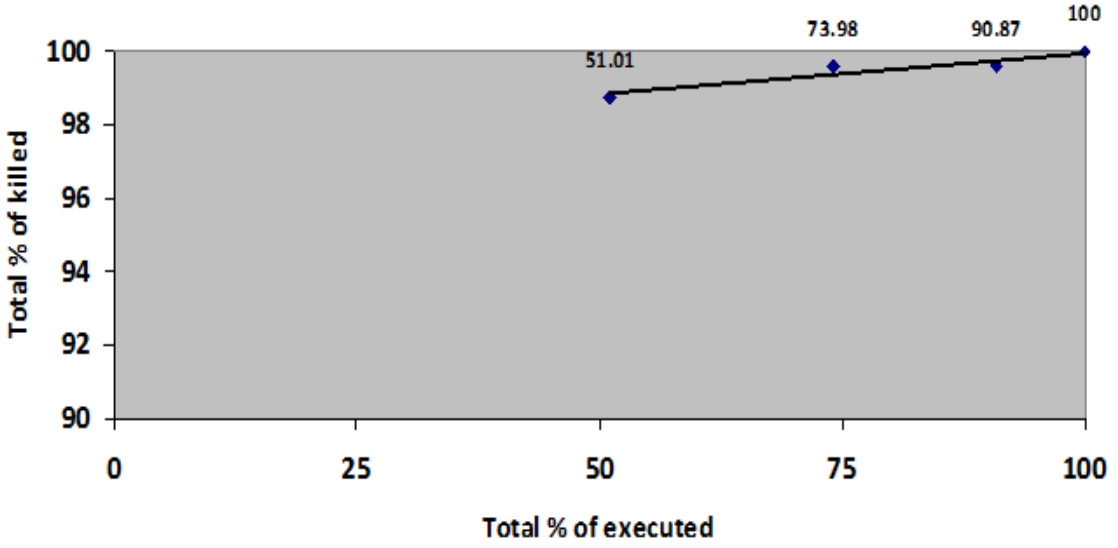

Figure 5.6: Bub program

The results obtained on executing the test cases against the mutants of program Trishmall, are shown in figure 5.7. These results show that on the execution of single mutant from each group more than 98% results are obtained for the program. On the execution of two mutants from each group 100% results are obtained. Hence, there is no need to execute rest of the mutants of the program. Number of mutants required for execution is less than the total number of mutants.

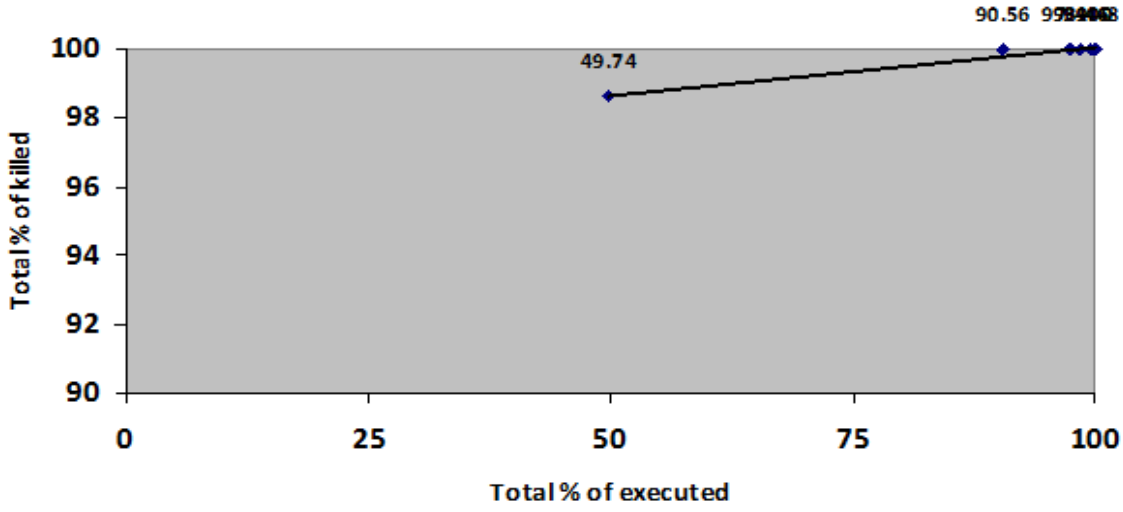

Figure 5.7: Trishmall program

The results obtained on executing the test cases against the mutants of program Mid, are shown in figure 5.8. These results show that on the execution of single mutant from each group more than 98% results are obtained for the program. On the execution of two mutants from each group 100% results are obtained. Hence, there is no need to execute rest of the mutants of the program. Number of mutants required for execution is less than the total number of mutants.

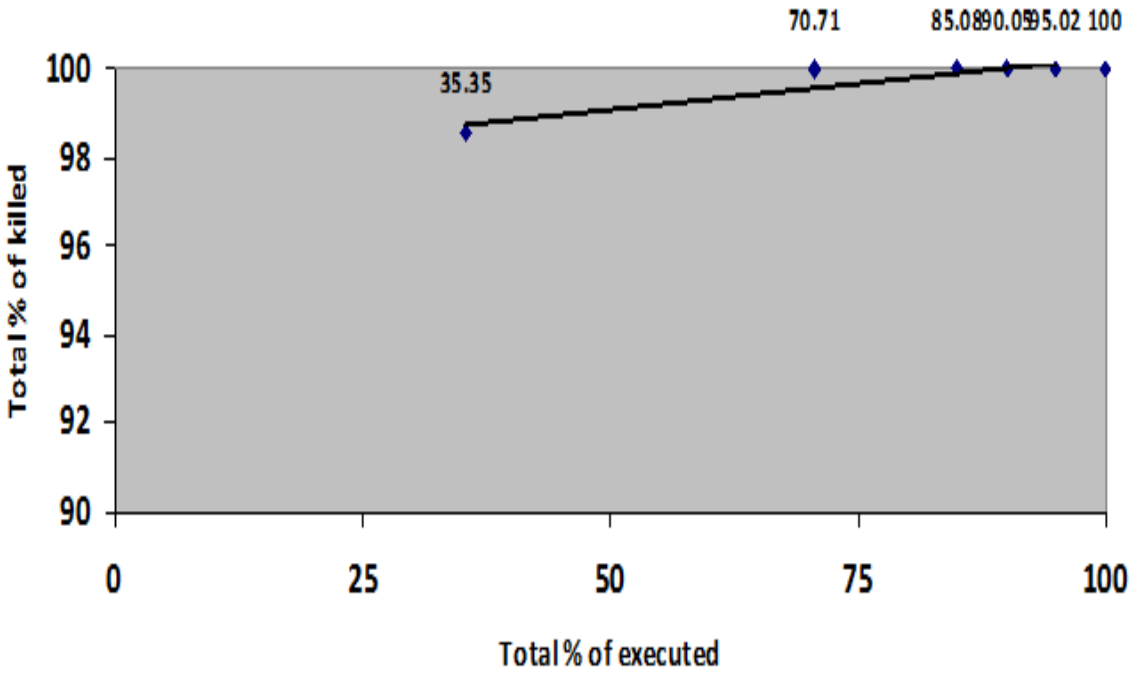

Figure 5.8: Mid program

The results obtained on executing the test cases against the mutants of program Euclid, are shown in figure 5.9. These results show that on the execution of single mutant from each group more than 98% results are obtained for the program. On the execution of three mutants from each group 100% results are obtained. But we get satisfactory results on the execution of two mutants from each group that is 99.107%. Hence, there is no need to execute rest of the mutants of the program.

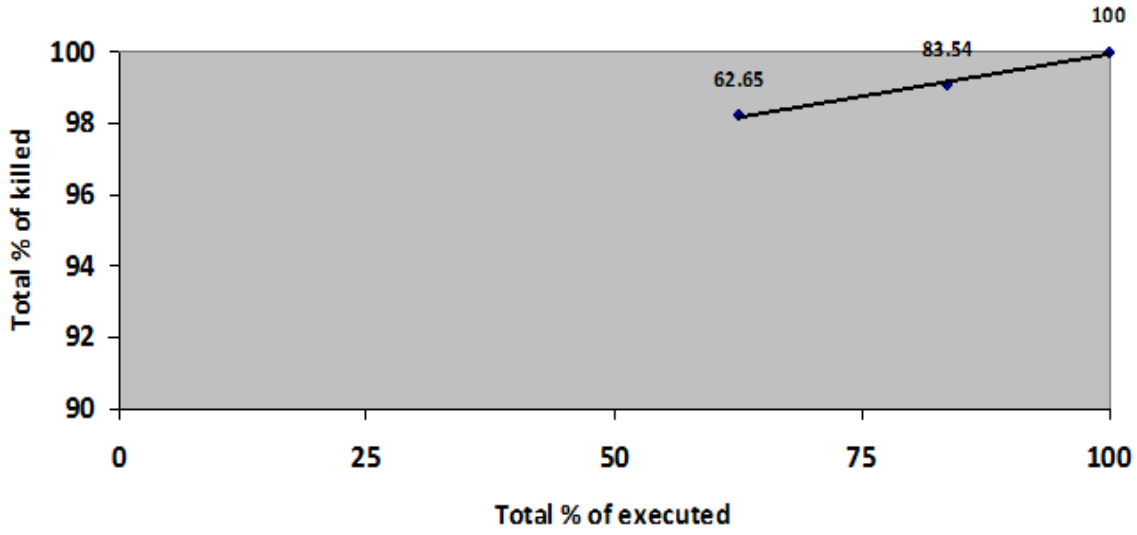

Figure 5.9: Euclid program

The results obtained on executing the test cases against the mutants of program Pat, are shown in figure 5.10. These results show that on the execution of single mutant from each group 100% results are obtained for the program. Hence, there is no need to execute rest of the mutants of the program. Number of mutants required for execution is less than the total number of mutants.

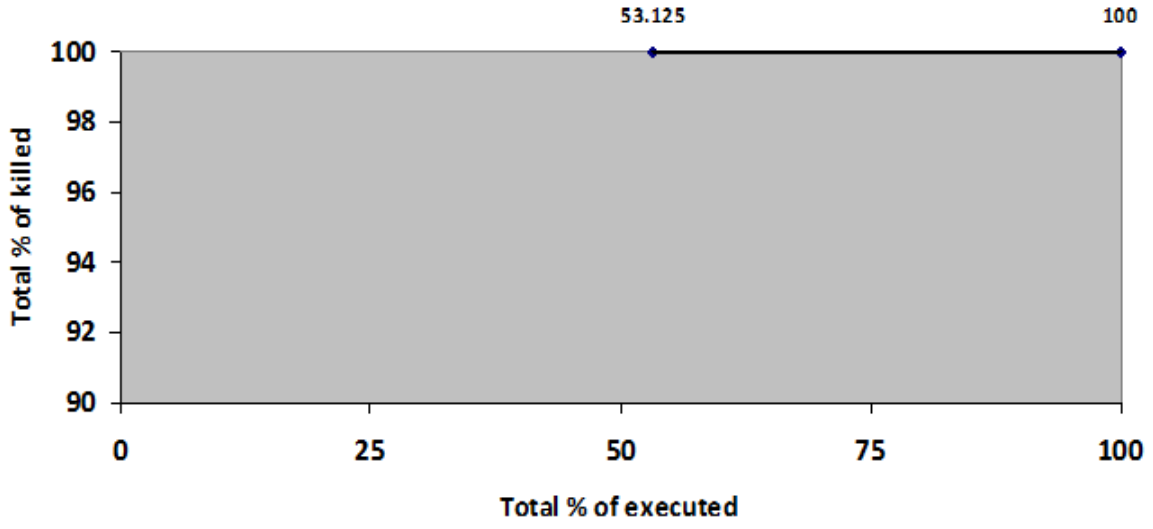

Figure 5.10: Pat program

5.2 Comparison

Table 5.4 shows the comparison between results of existing approach [42] and the proposed approach for checking the equivalence of mutants.

Table 5.4: Comparison between results of existing approach and proposed approach for deporting mutants to equivalent mutants

Program	***Offutt and Pan proposed technique[42]***			***Proposed mutation operators and conditions for deporting equivalent mutants***		
	Total Mutants	***Killed Mutants***	***Equivalent Mutants***	***Total Mutants***	***Killed Mutants***	***Equivalent Mutants***
Quad	359	47	31	361	277	40
Insert	460	320	46	363	272	41
Warshall	305	192	35	307	236	33
Bsearch	299	211	27	300	224	28
Bub	338	232	35	296	243	33
Trishmall	544	334	97	392	245	79
Mid	183	43	13	181	138	23
Euclid	196	147	24	158	112	25

On observing table 5.4, it is concluded that the results of existing approach [42] is approximately same as the results collected on using above mentioned mutation operators and their conditions. The deviation between the results is observed in table 5.4. The deviation is due to the presence of live mutants. Live mutants could be categorized as killed or equivalent mutant. Some of the mutants which were earlier considered as live are decided to be equivalent by using the above mentioned criteria. Some of the equivalent mutants are not found due to pre-condition, no semantic change, etc. As the results are collected using these reasons of equivalence without executing the mutants.

Table 5.5, illustrates the comparisons of the existing techniques [30, 33, 42] with the proposed (SBMT) technique.

Table 5.5: Comparison between SBMT and existing techniques

Technique	*Type Of Mutants*		*Applicable for large lines of code*	*Domain analysis before testing*	*Potential of detecting equivalent mutant*
	Class Level	*Method Level*			
Offutt and Pan proposed technique[30]	No	Yes	Yes	Yes	Good
Pan proposed technique[33]	Yes	Yes	Yes	Yes	Low
Offutt and Lee proposed technique[42]	Yes	Yes	No	No	Low
SBMT	Yes	Yes	Yes	Yes	Good

The proposed approach is applicable to both method and class level mutants. This approach is also applicable to the programs having large lines of code and analyses the domain before testing. Major benefits of the proposed approach are the reduction in effort, time and cost consumption and detect equivalent mutants with ease.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

Mutation testing is a fault based testing technique that works in conjunction with traditional testing techniques. It is done by generating faults and observes the effect by going through three conditions: reachability, infection and propagation. Mutation testing suffers from the problem of detection of equivalent mutants and large effort, time and cost consumption.

This thesis presents an approach solving these problems mutually. The proposed approach is inexpensive for mutation testing. It works on the reduction of mutation cost as well as helping in detection of equivalent mutant. SBMT determines mutation operators and corresponding conditions contributing towards the creation of equivalent mutants. Method level mutation operators AOIU and AOIS are contributing to equivalent values. Class level mutation operators AM, IHI, IHD, IOD, IOR, ISI, ISD, IPC, OAN, PMD, PCI, JSI, JSD, JID and JDC shows equivalence. Equivalent mutant occurs due to pre-conditions, change in dead code and no semantic change in code. If these conditions and the corresponding mutation operator are kept in mind while creating mutant, the problem of equivalent mutant detection is solved with ease and less effort consumption.

The SBMT algorithm works for mutants having same mutation operator at different locations in a program under similar conditions, by accumulating them in a group. The proposed algorithm covers all types of Java mutants and can be applied to programs of any length. The SBMT algorithm helps in detracting time, effort and cost consumption to 60 percent. Additionally it helps in calculation of mutation score more accurately.

6.2 Future work

In future works can be carried out on the following directions:

- Automation of the proposed approach for the removal of equivalent mutants at the time of their creation.
- Extending the proposed approach to other types of mutants.

- This technique has a limitation that it is not competent to work on the subgroup of a group. The work can be extended on subgroups.
- An automated system for the formation of groups can be evolved and integrated with SBMT.
- There is still a scope of reduction in cost, time and efforts of mutation testing.

References

[1] A. Bertolino, and E. Marchetti, "A brief essay on software testing", Wiley-IEEE Computer Society, The Development Process of Software Engineering, vol. 1, pp. 393-411, 2005.

[2] R. S. Pressman, "Software Engineering – A Practitioner's Approach", McGraw Hill Education Asia, 2005.

[3] W. Afzal, "Metrics in Software Test Planning and Test Design Processes", M. S. Thesis, MSE-2007:02, Blekinge Institute of Technology, Sweden, pp. 1-111, Jan 2007.

[4] IEEE Standard 829-1998, IEEE Standard for Software Test Documentation, IEEE, 1998.

[5] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, New York, 1989.

[6] Mohd. E. Khan, "Different Forms of Software Testing Techniques for Finding Errors", IJCSI, vol. 7, no. 1, pp 11-16, May 2010.

[7] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs", In Proceedings of the International Conference on Software Testing, Verification, and Validation, Los Alamitos, pp. 52-61, April 2008.

[8] "Error Seeding and Mutation testing", Available at: http://laser.cs.umass.edu/courses/cs521-621.Fall10/documents/06-MutationTesting-R-2.pdf.

[9] A. J. Offutt, J. Voas, and J. Payne, "Mutation Operators for Ada", Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, October 1996.

[10] P. May, J. Timmis, and K. Mander, "Immune and evolutionary approaches to software mutation testing", In Proceedings of the International Conference on Artificial Immune Systems (ICARIS '07), Santos, Brazil, pp. 336-347, 2007.

[11] M. Umar, "An evaluation of Mutation Operators for Equivalent Mutants", M.S. thesis, Computer Science, King's College, London, 2006.

[12] M. Polo, S. Tendero, and M. Piattini, “Integrating techniques and tools for testing automation: Research Articles”, Software Testing, Verification and Reliability, vol. 17, no. 1, pp. 3-39, 2007.

[13] Yu-Seung Ma, Yong-Rae Kwon, and A. J. Offutt, “MuJava: An Automated Class Mutation System”, Software Testing, Verification and Reliability, vol. 15, no. 2, pp. 97-133, June 2005.

[14] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt, “Inter-class Mutation Operators for Java”, In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE’02), IEEE Computer Society, Annapolis, Maryland, November 2002.

[15] A. J. Offutt, and R. H. Untch, “Mutation 2000: Uniting the Orthogonal”, In Proceedings of the workshop on Mutation Analysis (MUTATION’00), published in book form, as Mutation Testing for the new century, San Jose, California, vol. 24, pp. 34-44, October 2001.

[16] P. May, K. Mander, and J. Timmis, “Software vaccination: An artificial immune system approach to mutation testing”, In Proceedings of the International Conference on Artificial Immune Systems (ICARIS ’03), Edinburgh, UK, volume 2787 of Lecture Notes in Computer Science, pp. 81-92, 2003.

[17] W. E. Wong, and A. P. Mathur, “Reducing the Cost of Mutation Testing: An Empirical Study”, Journal of Systems and Software, vol. 31, no. 3, pp. 185–196, December 1995.

[18] A. J. Offutt, and J. Pan, “Automatically Detecting Equivalent Mutants and Infeasible Paths”, Software Testing, Verification and Reliability, vol. 7, no. 3, pp. 165–192, September 1997.

[19] A. J. Offutt, G. Rothermel, and C. Zapf, “An Experimental Evaluation of Selective Mutation”, In Proceedings of the International Conference on Software Engineering (ICSE’93), IEEE Computer Society, Baltimore, Maryland, pp. 100–107, May 1993.

[20] A. J. Offutt, G. Rothermel, and C. Zapf, “An Experimental Determination of Sufficient Mutation Operators”, ACM Transactions on Software Engineering, IEEE Computer Society, Baltimore, MD, pp. 100 –107. May 1993.

[21] E. S. Mresa, and L. Bottaci, “Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study”, Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 205–232, December 1999.

[22] S. Hussain, “Mutation Clustering”, M. S. Thesis, King’s College London, UK, 2008.

[23] C. Ji, Z. Chen, B. Xu, and Z. Zhao, “A Novel Method of Mutation Clustering Based on Domain Analysis”, In Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE’09), Massachusetts: Knowledge Systems Institute Graduate School, Boston, July 2009.

[24] R. H. Untch, A. J. Offutt, and M. J. Harrold, “Mutation Analysis Using Mutant Schemata”, In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’93), Cambridge, Massachusetts, pp. 139–148, 1993.

[25] A. S. Namin, and J. H. Andrews, “On Sufficiency of Mutants”, In Proceedings of the International Conference on Software Engineering (ICSE COMPANION’07), Minneapolis, Minnesota, pp. 73–74, May 2007.

[26] M. Polo, M. Piattini, and I. Garcia-Rodriguez, “Decreasing the Cost of Mutation Testing with Second-Order Mutants”, Software Testing, Verification and Reliability, vol. 19, no. 2, pp. 111–131, June 2009.

[27] A. O. Akinde, “Using Higher Order Mutation for Reducing Equivalent Mutants in Mutation Testing”, Asian Journal of Computer Science and Information Technology, vol. 2, no. 3, 2012.

[28] Y. Jia, and M. Harman, “Constructing Subtle Faults using Higher Order Mutation Testing”, In Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM’08), Beijing, China, pp. 249–258, September 2008.

[29] Y. Jia, and M. Harman, “Higher order mutation testing”, Information and Software Technology, vol. 51, no. 10, pp. 1379-1393, 2009.

[30] A. J. Offutt, and J. Pan, “Detecting Equivalent Mutants and the Feasible Path Problem”, In Proceedings of the Annual Conference on Computer Assurance, IEEE Computer Society, Gaithersburg, Maryland, pp. 224–236, June 1996.

[31] R. M. Hierons, M. Harman, and S. Danicic, “Using Program Slicing to Assist in the Detection of Equivalent Mutants”, Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 233–262, December 1999.

[32] K. Adamopoulos, M. Harman, and R. M. Hierons, “How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution”, In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO’04), Seattle, Washington, USA, vol. 3103, pp. 1338–1349, June 2004.

[33] J. Pan, “Using Constraints to Detect Equivalent Mutants”, M.S. Thesis, George Mason University, Fairfax VA, 1994.

[34] S. Nica, M. Nica, and F. Wotawa, “Detecting equivalent mutants by means of constraint systems”, In Proceedings of the International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011), Barcelona, Spain, pp. 21-24, 2011.

[35] S. Nica, and F. Wotawa, “Using Constraints for Equivalent Mutant Detection”, In Proceedings of the Workshop on Formal Methods in the Development of Software, Open Publishing Association, France, vol. 1207, no. 2234, pp. 1-8, 2012.

[36] S. Nica, and F. Wotawa, “EqMutDetect—A tool for equivalent mutant detection in embedded systems”, In Proceedings of the Workshop of Intelligent Solutions in Embedded Systems (WISES), Klagenfurt, Austria, pp. 57-62, 2012.

[37] M. Kintis, M. Papadakis, and N. Malevris, “Isolating First Order Equivalent Mutants via Second Order Mutation”, In Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), Montreal, Quebec, Canada, pp. 701-710, 2012.

[38] D. Schuler, and A. Zeller, “(Un-) Covering Equivalent Mutants”, In Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), Paris, France, pp. 45-54, April 2010.

[39] BJM Grun, D. Schuler, and A. Zeller, “The impact of equivalent mutants”, In Proceedings of the International Workshop on Mutation Analysis (MUTATION’09), published with Proceedings of the International Conference on

Software Testing, Verification, and Validation Workshops, Denver, Colorado, IEEE Computer Society, pp. 192-199, April 2009.

[40] W. E. Wong, "On Mutation and Data Flow", Phd Thesis, Purdue University, West Lafayette, Indiana, 1993.

[41] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing", In Proceedings of the International Conference on Software Testing, Verification, and Validation, Paris, France, pp 100-109, April 2010.

[42] A. J. Offutt, and S. Lee, "An Empirical Evaluation of Weak Mutation", IEEE Transactions of Software Engineering, vol. 20, no. 5, pp. 337-345, 1994.

[43] Y. Jai, and M. Harman, "An Analysis and Survey of the Development of Mutation Testing", IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, 2010.

Publications

Accepted

[1] T. Singla, and A. Kumar, "Mutation Operators corresponding Conditions Contributing in Deporting them Equivalently", IJCST, vol. 4, no. 2, 2013.

Communicated

[2] T. Singla, and A. Kumar, " Reducing Mutation Testing Endeavor using the Similar conditions for the same Mutation Operators occurs at Different Locations", International Journal, 2013.

APPENDIX A – Experimental Java programs (Source Code)

The programs written in this section were developed to create mutants for performing experiment.

Quad.java

```
/* @author: tannu */
class Quad{
   public static void main(String args[]){
      double b=2, a=2, c=1, dis, root1, root2;
      dis =(b*b)-(4*(a)*(c));
      if(dis<0 ){
                  System.out.println("Roots are imaginary");       }
      else if(dis==0)      {
         root1 = (-b)/( 2*a) ;
         System.out.println("roots are real and equal:"+ root1);           }
     else{
                  root1 = ((-b)+ Math.sqrt(dis))/(2*a);
                  root2 = ((-b-Math.sqrt(dis))/(2*a));
                  System.out.println("The result is:"+ root1);
                  System.out.println("The result is:"+root2); }
           }
   }
```

Insert.java

```
/* @author: tannu */
public class InsertSort{
          public static void main(String a[]){
           int i;
```

```
        int array[] = {12,9,4,99,120,1,3,10};
        System.out.println("Values Before the sort:\n");

        for(i = 0; i < array.length; i++)
        System.out.print( array[i]+"  ");
        System.out.println();
        insertion_srt(array, array.length);
        System.out.print("Values after the sort:\n");

        for(i = 0; i < array.length; i++)
        System.out.print(array[i] +"  ");
 }
 public static void insertion_srt(int array[], int n){
        int i=1, j=0, temp=0;
        for ( i = 1; i < n; i++){
         temp = array[i];
         for(j=(i-1);j>=0;j--){
                if((array[j]>temp)){
                array[j +1]=array[j]; }
                else{
                break;}
                }
                array[(j+1)]=temp; }
     }
}
```

Warshall.java

```
/* @author: tannu */
public class Warshall {
    public static void main(String a[]){
```

```
        int array[][] ={ {1, 1, 0, 1},
                         {0, 1, 1, 0},
                         {0, 0, 1, 1},
                         {0, 0, 0, 1} };
         int N=array.length;
         for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                        System.out.print(array[i][j]);   }
                System.out.println();    }
                System.out.println("after");
        WarshallsAlgorithm(array,N);
         }
public static void WarshallsAlgorithm(int[][] adj, int N){
        int i=0, j=0, k=0;
           for (k = 0; k < N; k++){
           for (i = 0; i < N; i++){
                  for (j = 0; j < N; j++){
                        adj[i][j] = (adj[i][j] | (adj[i][k] & adj[k][j]));   }
                }
           }
      for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
                 System.out.print(" "+ adj[i][j]);  }
        System.out.println();   }
      }
}
```

Bsearch.java

```
/* @author: tannu */
public class Bsearch {
```

```
public static void main(String[] args) {
 int[] intArray = new int[10];
 int searchValue = 0, index ,i=0;
 System.out.println("Enter 10 numbers");
 Scanner input = new Scanner(System.in);
          for ( i = 0; i < intArray.length; i++) {
                   intArray[i] = input.nextInt();
          }
          System.out.print("Enter a number to search for: ");
          searchValue = input.nextInt();
          index = binarySearch(intArray,searchValue);
          if (index != -1) {
          System.out.println("Found at index: " + index);  }
         else {
         System.out.println("Not Found");     }
         }
public static int binarySearch(int[] search, int find) {
          int start, end=0, midPt=0;
          start = 0;
         end =  search.length - 1;
                   while (start <= end) {
                   midPt = (start + end) / 2;
                   if (search[midPt] == find) {
                   return midPt;   }
                   else if (search[midPt] < find) {
                   start = midPt + 1;  }
                   else {
                   end = midPt - 1;  }
          }
          return -1;   }
}
```

Bub.java

```
/* @author: tannu */
class BubSort{
        public static void main(String a[]){
         int i;
         int array[] = {12,9,4,99,120,1,3,10};
        System.out.println("Values Before the sort:\n");

        for(i = 0; i < array.length; i++)
        System.out.print( array[i] +"  ");
        System.out.println();
        bubble_srt(array,array.length );
        System.out.print("Values after the sort:\n");

        for(i = 0; i < array.length; i++)
        System.out.print(array[i]+"  ");
        System.out.println();
        }
        public static void bubble_srt( int a[], int n ){
                int i, j, t=0;
                for(i = 0; i <n; i++){
                        for(j = 1; j < (n - i); j++){
                                if (a[(j -1)] > a[j]){
                                t = a[(j -1)];
                                a[(j -1)] = a[j];
                                a[j] = t;  }
                        }
                }
        }
}
```

Trishmall.java

```
/* @author: tannu */
class Trishmall
{
	private int side1, side2, side3;
	private static final String P_EQUILATERAL = "equilateral";
	private static final String P_ISOSCELES   = "isosscles";
	private static final String P_RIGHTANGLED = "right-angled";
	private static final String P_SCALENE     = "scalene";
	private static final String P_IMPOSSIBLE  = "impossible";

	public Trishmall(int s1, int s2, int s3)
	{		side1 = s1;
			side2 = s2;
			side3 = s3;
	}
	public String classify()
	{	if (isImpossible())
		{			return P_IMPOSSIBLE;
		}
		 if(side1 >0 || side2 >0 && side3> 0 ) {
			if (side1 == side2 && side2 == side3 && side3 == side1)
			{	return P_EQUILATERAL;
			}
			if (side2 == side3 || side1 == side2 || side1 == side3)
			{	return P_ISOSCELES;
			}
			if (isRightAngled())
			{		return P_RIGHTANGLED;
			}
```

```
            }
            return P_SCALENE;
        }
        public boolean isRightAngled()
        {
        int[] sides = new int[] { side1, side2, side3 };
        return sides[2] == Math.sqrt((long) sides[0] * sides[0] + (long) sides[1] * sides[1]);
        }
        public boolean isImpossible()
        {           if (side1 <= 0 || side2 <= 0 || side3 <= 0)
            {           return true;
            }
            return false;
            }
        public static void main(String[] args)
            {           Trishmall trishmall = null;
            try  {
                        trishmall= new Trishmall (3,4,5);  }
            catch (Exception e)
                    {
        System.out.println("Usage: java Quadrangle <side1:int> <side2:int> <side3:int>");
                    return;  }
                    System.out.println("Type: " + triangle.classify());  }
        }
```

Mid.java

```
/* @author: tannu */
public class midno
```

```
{       public static void main(String args[]){
        int a = 10;
        int b= 20;
        int c = 30;
        if(a>=b&&a>=c){
                if(b>c)
                System.out.println("mid no. is "+ b);
                else
                System.out.println("mid no. is "+c);}
        else if(b>=a&&b>=c){
             if(a>c)
            System.out.println("mid no. is "+a);
             else
            System.out.println("mid no. is "+c);      }
    else{
         if(a>b)
          System.out.println("mid no. is "+a);
          else
          System.out.println("mid no. is "+b);           }
    }
}
```

Euclid.java

```
/* @author: tannu */
public class Euclid
{       public static int gcd2(int p, int q)
        {               System.out.println("gcd(" + q + ")" );
                        int temp =0;
                         while (q != 0) {
                                temp = q;
```

```
                q = p % q;
                p = temp;        }
            return p;      }
    public static void main(String[] args) {
            int p = 4;
            int q = 3;
            d2 = gcd2(p, q);
        System.out.println("gcd(" + p + ", " + q + ") = " + d2);            }
    }
```

Pat.java

```
/* @author: tannu */
public class Pat
{       public static void main(String args[])
        {
        String text = "6hello8901hhhelllo7890hello";
        String word = "hello";
        for(int i=0; i!=-1;) {
        i= (text.indexOf(word,i+1));
                if(i!=-1)
                System.out.println("first index" +  i);
                }
        int k=text.length();
        for (int i = k; i != -1; ) {
        int j = text.lastIndexOf(word, i - 1);
         i=j;
        if(i!=-1)
        System.out.println("last"+i); }
     }
}
```