MuString: A Mutation Testing Tool for Reducing Execution Time of Mutants Using String Matching Algorithm

Dissertation submitted in partial fulfillment of requirement for award of degree of

Master of Technology

in

Computer Science and Applications

Submitted By:

Rashi Mittal

(Roll No. 601203020)

Supervised By:

Ms. Sunita Garhwal

SMCA, Thapar University, Patiala

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS

THAPAR UNIVERSITY

PATIALA – 147004

July, 2014

Certificate

I hereby certify that the work which is presented in the dissertation entitled, "MuString: A Mutation Testing Tool for Reducing Execution Time of Mutants using String Matching Algorithm", in partial fulfillment of the requirements for the award of degree of the Master of Technology in **Computer Science and Applications**, submitted in School of Mathematics and Computer Applications, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Ms. Sunita Garhwal and refers others researcher's work which are duly listed in the reference section.

The matter presented in this dissertation has not been submitted for award of any other degree of this or any other university.

Rashi Mittal

(Rashi Mittal)

601203020

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

Sunita

(Ms. Sunita Garhwal)

Assistant Professor

School of Mathematics and Computer Applications

Countersigned by

(Dr. Rajesh Kumar)

Head

School of Mathematics and Computer Applications

Thapar University

Patiala

(Dr. S. K. Mohapatra)

Dean Academic Affairs

Thapar University

Patiala

Acknowledgements

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the encouragement and able guidance of my supervisor Ms. Sunita Garhwal. I thank my supervisor for her time, patience, discussions and valuable comments. Her enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to Dr. Rajesh Kumar, Associate Professor and Head, School of Mathematics and Computer Applications, for motivation and inspiration that triggered me for the dissertation work.

I will be failing in my duty if I don't express my gratitude to Dr. S. K. Mohapatra, Senior Professor and Dean of Academic Affairs of University, for making provisions of infrastructure such as library facilities, immensely useful for the learners to equip themselves with the latest field.

I am also thankful to the entire faculty and staff members of School of Mathematics and Computer Applications Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my close friends for their constant support.

Abstract

Today, testing is the most challenging and dominating activity used by industry, therefore, improvement in its effectiveness, both with respect to the time and resources, is taken as a major factor by many researchers. Software testing allows programmers to determine the quality of the software. Mutation testing is a branch of software testing that does more than this. It helps determine whether the test cases that have been created, effectively detect all the possible faults in the software. This allows the development of better test sets, thus, ensuring that maximum software quality is achieved.

Mutation testing works by inserting faults in the program. Various mutation operators are used to create these faulty programs. These programs are called mutants. The mutants depict software faults that may be caused by programmers while writing the software. Test cases are then executed on these mutants to determine if they have been *killed* or not. Test sets that kill all the mutants are considered to be good as they successfully detect all the possible program faults.

This dissertation targets this issue of time, effort that is needed for generating and executing large amount of mutants. A new tool *MuString* is developed that generate and execute mutants using *String Matching Algorithm* that reduces total execution time of mutants compared to other previous algorithms.

This technique takes the program as an input and then applies changes to that program using String Matching Algorithm. Complete methodology and its effectiveness have been demonstrated with the help of suitable examples.

List of Figures

List of Tables

List of Snapshots

Table of Contents

Chapter 2. Literature Survey

Chapter1

Introduction

This chapter includes the basic introduction of software testing and mutation testing. It also consists a brief introduction of the types, conditions, examples, and operators of mutation testing.

1.1 Software Testing

There are two classes of troubles in Computer software: faults or failures [39]. A fault is a circumstance that causes a functional unit to not be up to snuff in performing its requisite function. Failures arise in the system when the conveyed service diverged from the anticipated service. Software testing is a practice of determining faults in the software. Software testing is carried out to make sure that the correctness, completeness and quality of all software programs must meet. Testing is performed to verify that the algorithm used in the program is correct, and execute correctly in all possible conditions and it fulfills all the specified necessities. Software testing is the most crucial phase in Software Development Life Cycle (SDLC). It begins from the requirements specification phase, continue to the deployment phase, and so on [3, 32].

1.2 Objectives of Software Testing

Software testing follows these objectives [3]:

- Testing is a practice that executes the program with the motive of discovering errors in the program.
- Test cases are designed to perform testing, and better test case is one that has high chances of finding so far undistinguished errors.
- A successful test case is one that exposes so far undistinguished errors.

1.3 Testing Process

The framework is illustrated by IEEE829 standard within which whole testing process is managed [8]. There are many activities involved in the process of software testing. All of these activities are handled in sequence for testing of any software. In this process, faults are distinguished and accurate. It is a method of fixing faults that are exposed at the time of testing.

The life cycle of software testing works as a guide for managing so that the growth is computable in the form of attaining objectives.

In testing lifecycle, there are various activities as described in Figure 1.1.

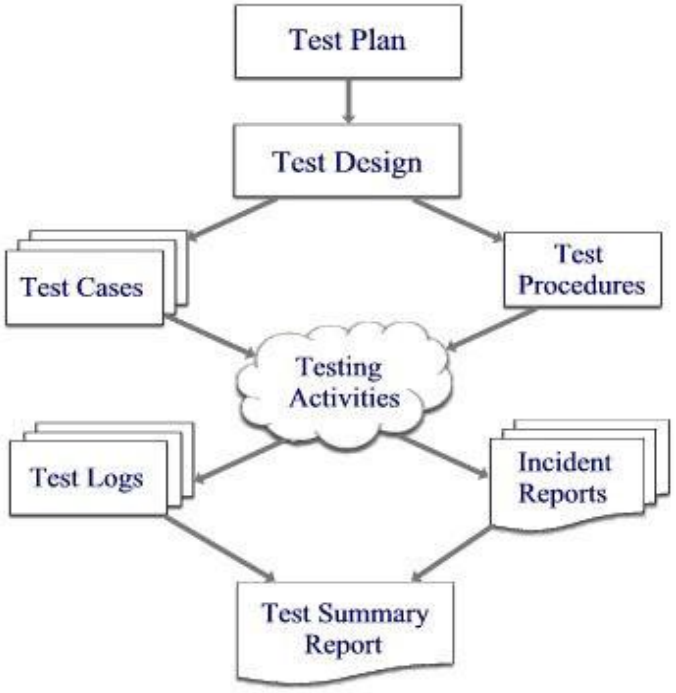


Figure 1.1: Testing Activities [8]

1.3.1 Test Plan

Test plan describes the scope, resources, agenda, and approach of testing activities in a given project. It also identifies the items, the characteristics of those items to be tested, and the individual testing tasks that are to be performed. Test planning is initiated immediately after the requirements are dictated. The main outcome of the planning phase is the document of test plan for every testing level.

1.3.2 Test Design

Test design refines the testing approach, identifies the test cases, and test item pass/fail criteria. Test cases that are developed as a component of a manuscript that is called test design specification [8]. This document comprises of input and output specification, needs of the environment and other things that are useful.

1.3.3 Test Cases

A test case is one that recognize constraints that are applied on test procedures. These are the conditions or a collection of conditions through which a well experienced tester will find out whether software or any features of these are properly working as desired or not. Test case structure comprises of identifier for test case, components that are to be tested, input and output that is expected specification, needs of environment, procedures required, *etc.*

1.3.4 Test Procedures

Test procedures illustrate the actual steps needed for operating the whole system and then execute all possible test cases to implement the design of tests. Test procedure structure comprises of identifier of test procedure, intention, requirements, steps, *etc.*

1.3.5 Test Logs

Test logs are brought into consideration for recording of what happened at the time of execution of a collection of test suite. Test logs structure comprises of identifier Test log, explanation, entries of events, *etc.*

1.3.6 Incident Reports

Incident reports help us to give an explanation of every event which occurs at the time of testing and that needs future investigation. Incident reports structure comprises of identifier, summary, incident explanation, incident impact, *etc.*

1.3.7 Test Summary Report

Test summary report structure comprises of identifier, all results summary, variance, comprehensive consideration, evaluation, approvals, and activities summaries, *etc.*

1.4 Software Testing Techniques

Software testing techniques are of two types.

1.4.1 Static Testing

It corresponds to the inspection of SRS (software requirement specification), SDS (software design specifications), documentation of the entire project and additional objects during desk

checks, reviews, audit, inspections, *etc* [32]. This testing validates the accuracy of design, codes, and stated requirements before the test suites executed.

1.4.2 Dynamic Testing

For testing the dynamic behavior of software, dynamic testing is used [3]. It defines the implementation of test suites, designing of tests, execution and reports. Dynamic testing can be performed as Black box testing and white box testing. It cannot provide evidence of accuracy of the software until it is executed in an exhaustive way.

1.5 Categorization of Testing Techniques

Testing techniques can be classified according to the origin of information that is deriving test data and standard to compute the adequacy of test suite.

1.5.1 White-Box Testing

White-Box Testing is same as glass-box testing. It considers the deep or inner structure for the designing of tests. It tests the implementation of software components and not worries about the outer explanations for that component of software [13]. White-box testing approaches includes mutation testing, data flow testing, path testing, control flow testing, and many more.

1.5.2 Black-Box Testing

Black-Box Testing is same as behavioral or functional testing. In this testing input conditions for a program can be derived by software engineer that will implement entire functional requirements [13]. In this kind of testing, part of the software is tested without knowing its core implementation. Black box testing approaches includes boundary value analysis, equivalence partitioning, and many more.

1.5.3 Gray-box Testing

Gray-box testing is same as translucent testing. It can be seen as an amalgam of white-box and black-box testing [13]. In this testing, the tester knows about the inner structure partially. Depending upon the partial information tester proposes the test cases and component of software under test is considered as a black box and tester test it from outside. Gray-box testing approaches include matrix testing, pattern testing, and many more.

1.6 Mutation Testing

Mutation testing (MT) is a white box fault oriented technique. It facilitates in combination with the conventional testing approaches. It emphasis on the test suits that are taken to analyze the programs, not like other testing approaches that emphasis on accurate functioning of the program. The main motive of MT is to create a better collection of test cases rather than trying to discover the faults that are present in the program. Mutation Testing is the basic testing measure for the assessment of not only programs but also for the test suites [6]. Revealing of faults in the program is a necessity, *e.g.*, the ESA Ariane five rocket in the year 1996 not succeed because of a unchecked error, the result of national tax system in Britain contained faults over 10000 invalid taxes in the year 2002.

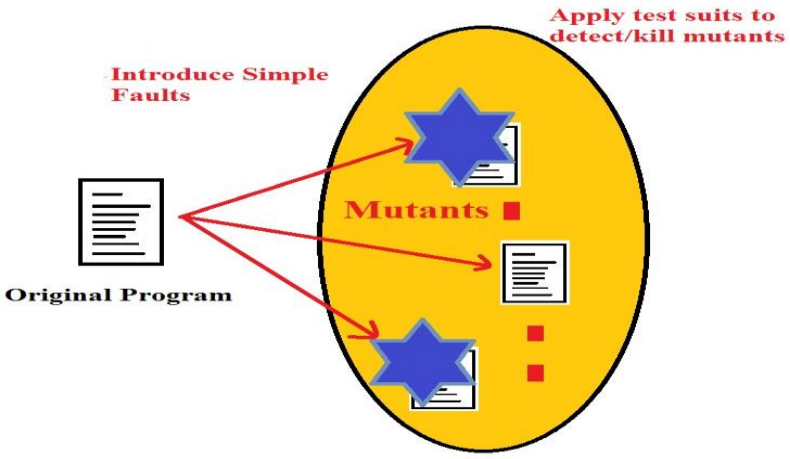


Figure 1.2: Mutation Testing

Mutation testing is firstly recommended by DeMillo *et al.* in 1978[1]. It focuses on measuring the satisfactoriness of a test suite to identify faults. The basic prototype on which MT works is that the faults are introduced via the programmer itself that signify the blunder that programmers repeatedly do. Such faults produce a set of damaged programs that are entitled as mutant programs. Mutant programs are different from the original program by only one fault. On both the programs *i.e.*, original and mutated, test suites are applied. Our main motive is to cause the mutant program to not succeed, consequently measuring the satisfactoriness of the test suite. The Mutant that commences only one fault in the program at a time is defined as *first-order* mutant while the other that commences several faults or changes in the program at a time is defined as *higher-order* mutant.

Example 1.1:

Original Program		**First-order Mutant**
1) int main() {		int main(){
2) int rem, sum=0, number=123;		int rem, sum=0, number=123;
3) while (number){		while (number){
4) **rem=number%10;**	***	**rem=number/10;**
5) number=number/10;		number=number/10;
6) sum=sum+rem; }		sum=sum+rem; }
7) printf("%d", sum);		printf("%d", sum);
8) return 0; }		return 0;}

Here in Example 1.1 program finds the sum of digits of the number and original program gives output 6. However, if we change only one operator, *i.e.*, replace % by / then it will give different or faulty output 13, and it will become the first order mutant.

Mutation testing can be defined in these steps:

- Insertion of faults in the program and constructing new versions using previously defined mutation operators that are called mutant.

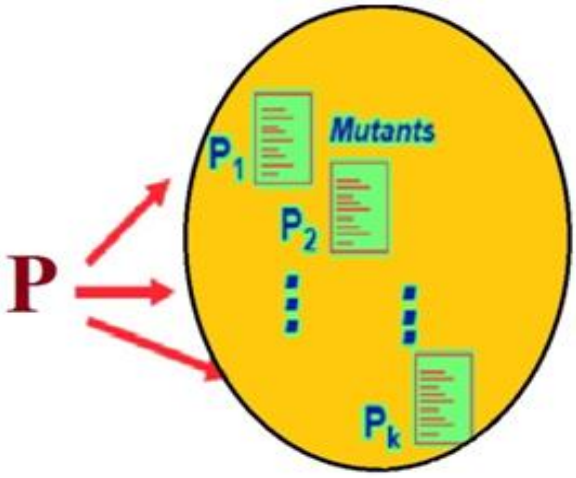


Figure 1.3: Mutant's of the Program

- Mutant programs are different from the original program by only one fault.
- Test suits are applied on the original and mutant programs.
- Our main motive is to cause the mutant program to not succeed, consequently measuring the satisfactoriness of the test suite.

1.7 Mutation Testing Process

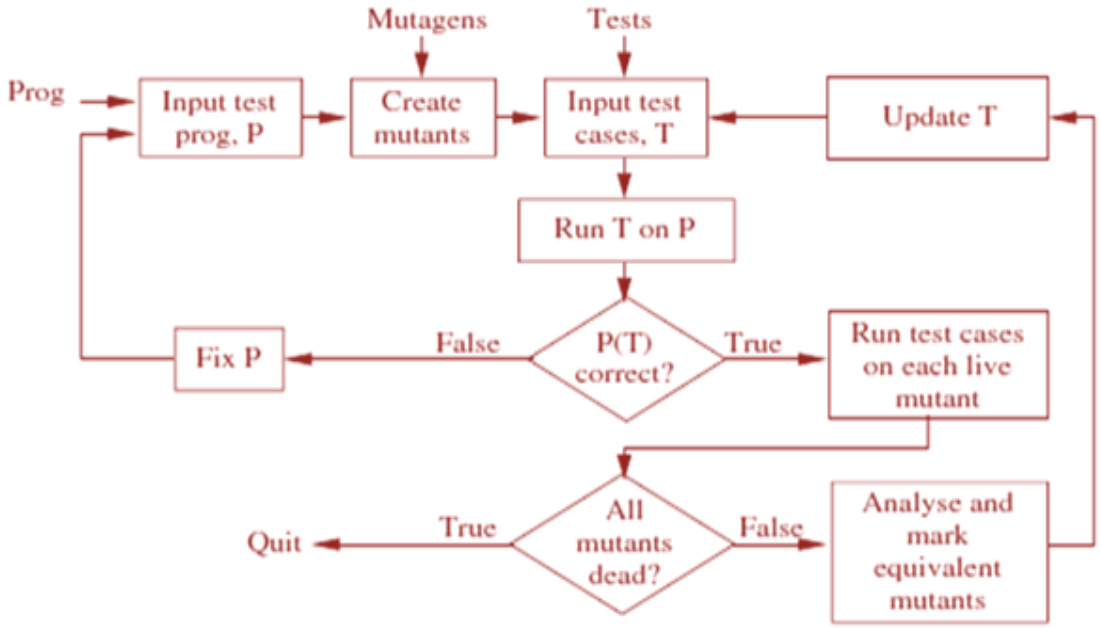


Figure 1.4: Process of Mutation Testing [18]

1.8 Types of Mutant

There can be 3 types of mutants:

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

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

iii) Killed Mutant: different outcome produced on execution of mutants against with the main program.

1.9 Assumptions in Mutation Testing

Assumptions in mutation testing are:

i) Competent Developer: programmer has written almost accurate program that is required.

ii) Coupling effect: dictates that test suites that destroy (kill) simple mutant can also destroy complex mutant.

1.10 Equivalent Mutant

A mutant that always produces identical results with the original program is called equivalent mutant. Recognition of equivalent mutant is one of the main issues associated with mutation testing.

Example 1.2:

Original Program		**Equivalent Mutant**
1) int main() {		int main(){
2) int rem, sum=0, number=123;		int rem, sum=0, number=123;
3) while (number){		while (number){
4) rem=number%10;		rem=number %10;
5) **number=number/10;**	***	**number=(number *1) /10;**
6) sum=sum+rem; }		sum=sum+rem;}
7) printf("%d", sum);		printf("%d", sum);
8) return 0; }		return 0;}

There is no universal test case which can generate a different outcome from the original and mutant program. Thus, the original and mutant programs are equivalent. These types of mutants are defined as equivalent mutants.

1.11 Mutation Score (MS)

Mutation score is evaluated to measure the efficiency of test cases. The MS is the number of detected faults over the total number of inserted faults.

$$Mutation\ Score = \frac{100 * Dead\ Mutants}{Total\ no.\ of\ mutants - no.\ of\ equivalent\ mutants}$$

MS will be 100% if entire set of mutants would kill or not live and if all the equivalent mutants are distinguished.

1.12 Test Data Conditions for Killing Mutants

There are three conditions that must be required to kill a mutant for satisfying the test data.

Let a program signifies by P, and a mutant of P on statement S is signifies by M and the test data for P is signifies by T. The conditions are given as [29, 39]:

i) ***Reachability condition:*** mutant is only syntactic change in statement S, so S must be reachable because the other statements in mutant and original program are exactly same [29]. If S is not reachable by test data T then, it is impossible to kill mutant M.

ii) ***Necessity condition:*** M must come in the state that differs from statement S of original program P after applying test data T [29]. It is compulsory that S must be reachable, and states of P and M must differ on statement S for killing M by test data T.

iii) ***Sufficiency condition:*** The final state of M must be different from P. Thus, the different state caused by the necessity condition must propagate through the program's computation to result in a different output [39].

1.13 Types of Mutation Testing

Mutation Testing can be classified as Weak Mutation Testing, and Strong Mutation Testing.

1.13.1 Weak Mutation Testing

This fulfills only starting two conditions of Mutation Testing.

Example 1.3:

	Program P		**Mutant M with weak mutation**
1)	Int m, n;		Int m, n;
2)	m = 10;		m = 10;
3)	n = m * 4;		n = m * 4;
4)	**if(n>m)**{... }	***	**if(n>=m)**{...}

1.13.2 Strong Mutation Testing

Strong mutation fulfills all the three conditions of Mutation Testing. Strong mutation testing is more powerful than weak mutation testing.

Example 1.4:

	Program P		**Mutant M with strong mutation**
1)	Int m, n;		Int m, n;
2)	m = 10;		m = 10;
3)	n = m * 4;		n = m * 4;
4)	**if(n>m)**{...}	***	**if(n<m)**{...}

1.14 Method Level Mutation Operators

These operators alter the main program by replacing, deleting, inserting the operator [39]. These operators are of six types as shown in Table 1.1.

Some of the operators are subdivided into binary, unary and short-cut versions. Thus, there are total 16 method-level operators [25].

Table 1.1: Method Level Mutation Operators [39]

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)

1.14.1 Arithmetic Operators

Numerical calculations on entire integers and floating-point numbers are performed by using arithmetic operators. In java the following arithmetic operators are supported:

- *Binary*: o + o (Add), o – o (Subtract), o * o (Multiply), o / o (Divide) and o % o (Mod).
- *Unary*: + (Positive value), - (Negative value)
- *Short-cut*: o++ (Post increment), ++o (Pre increment), o-- (Post decrement), --o (Pre decrement) .

  Here ‘o’ signifies operands on which mutation operators are applied at method level.

1.14.2 Relational Operators

Relational operators compare the value of two operands. In java the following relational operators are supported:

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

1.14.3 Conditional Operators

Computations are applied on the binary values of the operands. In java the following conditional operators are supported:

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

1.14.4 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. In java the following shift operators are supported:

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

1.14.5 Logical Operators

Boolean results are compared logically which are produced for expressions that we have to compare. In java the following logical operators are supported:

- *Binary*: o & o (AND), o | o (OR) and o ^ o (XOR).
- *Unary*: ~o (bitwise complement)

1.14.6 Assignment Operators

To set the values of the operand, assignment operators are used. On the right hand side of the operand calculations are performed, and the value is assigned to the left hand side operand. In java the following shift operators are supported:

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

1.15 Class Level Mutation 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. These four groups are shown in Table 1.2.

Table 1.2: Class Level Mutation Operators [39]

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

Class-level operators alter the syntax of any program by injecting, modifying and removing the expression that needs to be tested.

1.15.1 Encapsulation

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

- **AMC:** One access modifier is replaced with another modifier. Incorrect access modifier can lead to erroneous results. AMC permits the tester to make sure that accessibility level is correctly implemented in the program [20].

Example 1.5

Original Code		**Mutants**
private Data d;	***	public Data d;
	***	protected Data d;

Here, *** signifies the presence of a mutation operator in a particular expression of program using class-level operators.

1.15.2 Inheritance

Inheritance allows the use of methods and data of one class (superclass) in another class (child class) and increase the code reusability. Five operators are used for applying inheritance.

- **IHD:** Hiding variable is deleted in the subclass through IHD operator. Thus, the Superclass variables can be accessed.

Example 1.6

Original Code		**Mutant**
Class Vehicle {		Class Vehicle {
int color ;		int color;
…… }		……}
Class Car extends Vehicle {		Class Car extends Vehicle {
int color;	** *	// int color;
…… }		……}

- **IHI:** Hiding variable is inserted in the subclass through IHI operator.

Example 1.7

Original Code		**Mutant**
Class Vehicle {		Class Vehicle {
int color ;		int color;
…… }		……}
Class Car extends Vehicle {		Class Car extends Vehicle {
……	** *	int color;
…… }		……}

The behavior of the superclass can be modified by creating a method with the same name and arguments of the superclass is known as "*method overriding*". IOD, IOP and IOR operators are developed for overriding in Mutation Testing.

- **IOD:** In the subclass whole declaration is deleted of the overriding method by IOD operator. Thus, when method is referenced, the superclass method is accessed.

Example 1.8

Original Code		**Mutant**
Class Car extends Vehicle {		Class Car extends Vehicle {
…….		……..
wheel (int v) {	***	/* wheel (int v) {
…… } }		……}*/ }

- **IOP:** A method that is overriding in a subclass sometimes requires to call a method that it overrides in the superclass. If the method in the superclass is not called at the correct point in the program, it will take the program to an inaccurate state.

Example 1.9

Original Code	**Mutant**
class Vehicle {	class Vehicle {
…….	……
void Engine() {	void Engine() {

```
 amount = 5;                                   amount = 5;
  ….. } }                                        ….. }}
class car extends Vehicle {                    class car extends Vehicle {
…..                                            ……
void Engine(){                                 void Engine() {
super.Engine();                    ** *        amount = 10;
amount = 10;                       ***         super.Engine();
…  }                                             … }
```

- **IOR:** IOR operator checks that the overriding method is effecting another methods or not. Method that is being overridden in the superclass is renamed. Thus, the method that is overriding in the subclass would not effect on the method in the superclass.

Example 1.10

Original Code **Mutant**

```
class Vehicle{                                 class Vehicle {
…….                                            ……
void p() {….}                      ***         void p'() {…..}
void q() {                                     void q() {
p();                               ***         p'();
 …..}}                                           …..}}
Class Car extends Vehicle {                    Class Car extends Vehicle {
…….                                              ……
  void p() {….}                                   void p() {…..}
   void q() {                                     void q() {
    p();                                           p();
    …..}}                                             …..}}
```

To access the member functions and variables of the superclass, a super keyword is used in case of overriding. ISI and ISD operators are developed for this.

- **ISI:** On the methods and variables insertion of "*super"* keyword is done by ISI operator. Thus, it overrides references of methods and variables.

Example 1.11

Original Code		**Mutant**
Class Car extends Vehicle {		Class Car extends Vehicle {
…….		……
int p() {		int p() {
…….		……
return s * count;	***	return s * super.count;
}		}
}		}

- **ISD:** The "*super"* keyword on methods and variables is deleted by ISD operator. Thus, references to the methods and variables will now go to the overriding methods and variables. ISD works exactly opposite of ISI.

Example 1.12

Original Code		**Mutant**
Class Car extends Vehicle {		Class Car extends Vehicle {
…….		……
int p() {		int p() {
…….		……
return s * super.count; }}	***	return s * count; }}

The superclass constructors are not inherited as other methods. Super class default constructor is invoked before invoking its own constructor automatically whenever subclass object is formed.

- **IPC:** IPC deletes the superclass constructor call so that the default constructor can call the superclass.

Example 1.13

Original Code		**Mutant**
Class Car extends Vehicle {		Class Car extends Vehicle {
…….		……
Car (int k) {		Car (int k) {
Super (k);	***	// super (k);
…… }}		…...}}

1.15.3 Polymorphism

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

- **PNC:** The type or the constructor is changed that is for instantiating an object. The reference of an object that was declared will change to a different type.

Example 1.14

Original Code		**Mutant**
M m;		M m;
m = new M();	***	m= new N();

- **PMD:** Declared type of an object reference is changed to the parent of the original declared type.

Example 1.15

Original Code		**Mutant**
N n;	***	M n;
n = new N();		n= new N();

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

Example 1.16

Original Code		**Mutant**
boolean op (N o) {….}	***	boolean op (M o) {….}

- **PCI**: The actual type of an object reference is changed to the parent/child of the original declared type by PCI operator. The type of the object reference change for overriding methods and hiding variables exhibit different behavior.

Example 1.17

Original Code		**Mutant**
child cRef;		child cRef;
parent pRef = cRef;		parent pRef = cRef;
pRef.toString();	***	((child)pRef).toString();

When two or more methods of a class having similar names but their argument differs is called Method overloading. OMR, OMD, OAN comes under this.

- **OMR:** Body of a method is replaced by another method that has a similar name by OMR operator. Moreover, checks the invocation of an overloaded method that it is correct or not.

Example 1.18

Original Code		**Mutant**
class Car {		class Car {
…. …		……
void sum (int p) {….}		void sum (int p) {....}
void sum (int p, int q) {		void sum (int p, int q) {
……. }}	***	this.sum(p); }}

- **OMD:** Coverage of all overloaded methods is checked by OMD operator and one by one removes each overloading method.

Example 1.19

Original Code		**Mutant**
class Car extends Vehicle {		class Car extends Vehicle {
……		……
void wheel(int p){….}	***	// void wheel(int p){….}
void wheel(float p){….}		void wheel(float p){….}
……..}}		……}}

- **OAN:** The order or number of arguments in method invocations can be changed by OAN operator. The number of arguments is changed in such a way that the new argument list is accepted by other overloaded methods.

Example 1.20

Original Code		**Mutant**
c. wheel(0.6, 3);	***	c.wheel(3, 0.3);
	***	c.wheel(3);
	***	c.wheel(0.3);
	***	c. wheel();

1.15.4 Java-specific Features

It includes the few object-oriented language features that do not occur in other OO languages.

- **JTI:** JTI operator inserts *this* keyword and checks that the member variables are used correctly if they are hidden with the method parameters.

Example 1.21

Original Code		**Mutant**
class Car {		class Car {
char color;		char color;
……		……
void wheel(char color){		void wheel(char color){
this.color =color;	***	this.color= this.color
……}}		……}}

- **JTD**: This operator deletes *this* keyword from the program. It is the reverse of JTI.

Example 1.22

Original Code		**Mutant**
class Car {		class Car {
char color;		char color;
……		……
void wheel(char color){		void wheel (char color){
this.color= color;	** *	color = color
……}}		……}}

- **JSI:** *static* modifier is added for changing the instance variables to class variables.

Example 1.23

Original Code		**Mutant**
private float f=10.6;	***	private static float f = 10.6;

- **JSD:** *static* modifier is removed by JSD operator. It acts reverse of JSI.

Example 1.24

Original Code		**Mutant**
private static float f = 10.6;	***	private float f = 10.6;

- **JID:** Any initialization in the variable declaration and class construction is deleted by JID operator.

Example 1.25

Original Code		**Mutant**
class Car{		class Car {
char color = ‘red’;	***	char color;
Car() {….} }		Car() {….} }

- **JDC:** Any implementation of the default constructor is deleted by JDC that is programmed by programmer.

Example 1.26

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

- **EOA:** Java *clone()* method is used by EOA operator. EOA replaces an assignment of a pointer reference with a copy of the object using Java method and a copy of the contents of the object is created and reference to a new object is returned.

Example 1.27

Original Code		**Mutant**
Car c1, c2;		Car c1, c2;
c1 = new Car ();		c1 = new Car ();
c2 = c1;	***	c2 = c1.clone();

- **EAM:** The accessor method name is changed for another compatible accessor method name.

Example 1.28

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

- **EMM:** Modifier method name is changed for another compatible modifier method name.

Example 1.29

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

1.16 Dissertation Outline

This dissertation is organized into five chapters. Chapter1 describes the basic concepts of Software Testing and Mutation Testing. It also consists a brief introduction of the types, conditions, examples and operators of mutation testing. Chapter2 describes literature survey that has been done during this dissertation. Chapter3 describes the motivation behind the dissertation, discusses the problem statement and its objectives. Chapter4 explains all the results obtain from the algorithm that has been developed for reducing the total time for executing mutants. Chapter5 summarizes the conclusions drawn from the work done along with the directions regarding the future work.

Chapter2

Literature Survey

Mutation Testing is used to design new software tests and evaluate the quality of existing software tests by detecting faults that are inserted in the program that is under test. All the mutants must be killed for achieving high mutation score. MT can be used in various applications. A tool is the finest means to examine practically as it give evidence of experimental outcome and concepts. To sustain automated mutation analysis; a lot of MT applications have been constructed. There are more than 15 executed MT applications of all type.

2.1 Applications of Mutation Testing

Mutation Testing can be applied on several applications. Some of them are as following:

2.1.1 Mutation Testing for Java Database Applications

Zhou *et al.* [43] proposed Java Database Application Mutation Analyzer (JDAMA) that is a mutation testing tool that communicates with the database via Java Database Connectivity (JDBC) interface. Now a day, applications of database like online transaction, shopping, net banking, education systems are taken into major consideration in many software systems. Database application structure is multi-tiered. User interface is at upper, in the middle there is application tier and the lowest is DBMS (database management system). JDAMA tool differentiates the mutation scores that are generated by AGENDA's test generator on database applications. The technique has two phases.

i) ***Static phase*:** This phase runs abstract queries in which few token are exchanged for placeholders that represent values of java variables. Static phase has 3 measures. Initially, the bytecode of the testing application is inspected to estimate the collection of abstract queries that are implemented. Then, by the help of SQLMuation tool, a set of mutants for each query is produced. And at last in the bytecode, instrumentation is pushed to call a function, *i.e.*, MutantChecker that differentiate each query to its mutant at run time.

ii) ***Dynamic phase:*** In this phase, MutantChecker maps a query to the corresponding abstract query, instantiates variables in each abstract mutant of the query, executes each concrete mutant, and compares the result sets of these mutants to the result sets of the original query.

The corresponding abstract mutant is killed if the result sets differ. The (weak) mutation score of the test data can then be computed by dividing the number of abstract queries killed by the number of abstract queries.

At Present, this application is just making use of SELECT statements in SQL queries but in future other queries like UPDATE, INSERT, DELETE can also be used.

2.1.2 Mutation Testing for Web Application

Praphamontripong *et al.* [31] proposed WebMuJava tool. The need of Web application is increasing day by day, we have to find the revolutionary ways to test them. First we have to check that how these applications or software are interacting, *i.e.*, client-server, or server-server. For testing web application, integration testing can be used by using mutation analysis. WebMuJava tool is a mutation testing tool for testing web application. After applying the tool, it gives effective results for finding faults in web application and also shows several things for improvement.

Table 2.1: Steps for Testing Web Application

Step 1. Create mutants using WebMuJava tool.
Step 2. Tests can be generated by hand.
Step 3. Faults are injected into a subject program.
Step 4. Tests are executed on the injected faults.

In WebMuJava tool, 11 web mutation operators are implemented. Mutation analysis is applied to the connection that is between web application elements. Results depict that after applying mutation analysis to web application, tests are created which helps in finding faults. Web applications can interact with databases, so SQL mutation operators can also be implemented further for communicating with database related queries.

2.1.3 Decreasing Test Application Time through Test Data Mutation Encoding

Reda *et al.* [33] suggested compression algorithm that decrease the time required to test scan-based patterns. In this algorithm, the test vector set is compressed by encoding the bits that are required for pulling in the ongoing test data slice for getting the mutated test data slice. This

technique is not only used for stand-alone but it is also used for testing the data that is already compressed, and perform more compression. Compression algorithm is used for decreasing test data and execution time. In the test vectors, changes are encoded, decoders and shift registers are used to give adequate on-chip decompression. To calculate the minimum number of shift bits that are required to pull the needed bits in the test slice, state transition diagram of the shift register are designed. This technique produced the adequate behavior, and is powerful not only for stand-alone techniques but also as an enhancement of ongoing compression techniques. The capacity of this application is also analyzed on the big ISCAS-89 benchmark circuit. It produces a very small hardware overhead and big compression ratio that proves it to be the effective application for reducing test data time.

2.1.4 Interface Mutation for Integration Testing

Delamaro *et al.* [5] proposed that there are several methods developed for checking the effectiveness of tests at the unit level. However, there are lacks of approaches for checking the effectiveness of tests cases that are produced during integration testing. A mutation based approach known as IM (Interface Mutation) is useful at the time of integration testing. With this approach, errors are injected in UNIX sort utility, and the cost of application is measured, and the IM is evaluated. Many types of IM operators are evaluated. When effectiveness is compared after inserting errors on IM test sets that are generated randomly, then we observe that most of the time test sets of IM are superior.

Table 2.2: Procedure used for each IM based Test Set

Step 1. A test set size equal to that of the IM based set is generated randomly.
Step 2. Generated test set is executed against each erroneous version to determine whether or not it revealed the error.
Step 3. Only n test cases are considered from the randomly generated test set.
Step 4. Only m test cases are considered in the randomly generated test set.

IM is developed for testing the complete system or subsystems, and it does test the collaboration among pairs of units within a system.

2.1.5 PMACO: A Pheromone- Mutation Based Ant Colony Optimization for TSP

Shokouhifar *et al.* [38] proposed that TSP (Travelling salesman problem) is a very common and complex routing problem. Various algorithms are applied to solve TSP. An algorithm that is based on Ant Colony is applied using mutation operators in genetic algorithms to solve TSP. Ant Colony Optimization (ACO) basically comes in mind due to the activities done by the actual ants to find out the shortest path among their foodstuff and their nest. When ants walk, they leave a substance or chemical on the floor, *i.e.*, Pheromone.

To find the neighbor node of the solution related with two best ants, a mutation technique is applied on all iteration. The population is distributed in two categories: In first category the population that is reproduced according to the prototype that is pheromone in ACO, and the other category is produced by composite anticipated mutation scheme. Moreover, the preliminary population is produced related to the nearby neighborhood prototype by which ants goes in the direction that is needed for enhanced solution and minimize the whole run time. Then performance is compared with both ACO and GA with PMACO.

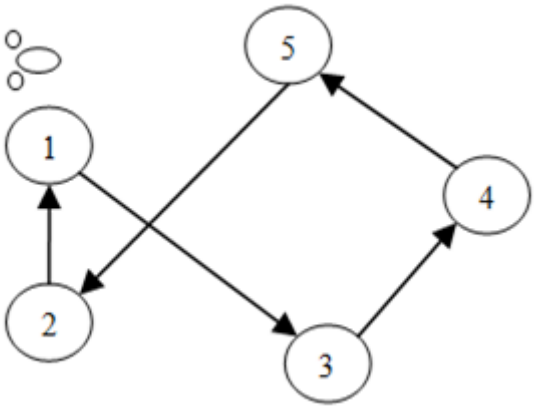


Figure 2.1: Graph Depiction of TSP [38]

In Figure 2.1, initially the ant is at node 1. Moreover, then ant has an option to choose the city, and according to the rule of algorithm, it chooses neighborhood city, it chooses the path 3,4,5,2, and again comes back to node 1 and complete its trip. Ants search its neighbor according to pheromone rule that is defined in Table 2.3.

Table 2.3: PMACO Algorithm Steps

Step 1: Initialize Constraints.
Step 2: Initial solutions are generated by probability based nearest neighbor rule (PNN).
Step 3: Solutions are evaluated.
Step 4: Stopping criteria is checked.
Step 5: Pheromone is updated.
Step 6: New population is constructed using pheromone transition rule and composite mutation scheme.
Step 7: Go to step 3.

By using Composite Mutation scheme, it gives a better solution to TSP.

2.1.6 Mutation Application to Knapsack Problem

Wang *et al.* [40] suggested an enhanced algorithm known as IQGA (Improved Quantum Genetic Algorithm). Mutation testing that is considered by IQGA, defined as MIQGA (Mutation Improved Quantum Genetic Algorithm). Better explore potential and fast convergences are the performance characteristics of MIQGA. To solve 0-1 knapsack problem, a function called greedy repair is used to fix impractical solutions. Experimental outcomes demonstrate that MIQGA have enhanced inclusive performance as compared to other conventional Genetic Algorithms. Chromosomes are coded by IQGA with amplitudes of probability that is shown by cosine and sine functions. Moreover, for updating the population an adaptive tactic of the rotation is used.

Table 2.4: Improved Quantum Genetic Algorithm with mutation (MIQGA)

Step 1: Initialize population.
Step 2: Examine the individual.
Step 3: Use Greedy Repair function to achieve feasible or practical solution.
Step 4: Calculate fitness of each function.
Step 5: Population is updated.
Step 6: Mutation operator is implanted.
Step 7: Return to step 2 until maximum number of evolutionary generation achieves.

2.1.7 Mutation Testing for Finite State Machine

Li *et al.* [15] proposed MT that is used for Finite State Machine (FSM) in the model based Testing. In this technique, test suites are constructed correspondingly to the model that is under supervision. It is done at the design phase to check faults or defects at early stage. Collection of mutation operators are illustrated for FSM as shown in Table 2.5. For testing the models of system in FSM, an algorithm is developed which pick a test suite for MT. The conduct test depicts that MT is more adequate and efficient than the other approaches in FSM testing Technique.

Table 2.5: FSM Mutators [15]

ROT	Reverse of Transition
MOT	Missing of Transition
DOT	Redundant of Transition
COI	Change of Input
MOI	Missing of Input
COO	Change of Output
MOO	Missing of Output
ESC	End State Changed
ESR	End State Redundant
SSC	Start State Changed
SSR	Start State Redundant

MT is a better approach than all other method like D-method, T-method, W-method and other state and transition coverage approaches on the basis of performance of distinguishing faults in FSM.

2.2 Approaches for Improving Mutation Testing

Detection of equivalent mutants is necessary. Equivalent mutant is one that is semantically similar as the original program, but syntactically different. These equivalent mutants lead to increasing the cost of computation. A number of approaches and techniques have been proposed for reducing computation cost and detection of equivalent mutants, as explained below:

i) 'Do Smarter' Approaches

Offutt *et al.* [26] suggested that the cost of computation can be divided among multiple machines by preserving some information. It avoids execution of entire program such as compiled code, so that same code does not get generated repetitively. Let, a program that has 1000 lines of code then by changing the 8^{th} line, 5 mutants are generated. Compiled form of the main code is saved rather than compiling the mutant number of times. As program's compiled form is changing the mutant for a single line only.

***Weak Mutation*:** The mutant and the original program's internal states are compared after the execution of both program mutated part [26]. A mutant is killed if the mutated program and original program's states differs, else the mutant is said to be live.

Distributed Architecture: The cost of computation can be divided over various machines by this approach [26]. Hypercube (MIMD) machines, SIMD machines, Network (MIMD) computers, vector processors are used by this for mutation analysis. Every mutant is considered independent of other mutant and are individually executed.

ii) 'Do Faster' Approaches

Offutt *et al.* [26] suggested that generation and execution of each mutant is done as soon as possible.

Schema-based Mutation Analysis: Metamutant is created by this approach in which all mutants are encoded into single source program. Compilation and execution of metamutant is done once in similar programming environment. For saving the computation cost, repetition is avoided for compilation and execution.

Separate Compilation Approach: Each mutant is created, executed, and runs individually but not in interpretative style [26]. It is 15 to 20 times faster in execution than the interpretative style, *e.g.,* Proteum system.

iii) 'Do Fewer' Approaches

Offutt *et al.* [26] proposed that minimal set of mutants are executed. From the entire collection of mutants, a subset of mutants is selected so that it is enough to find out an excellent set of test cases.

Selective Mutation: May *et al.* [19] suggested that Maximum coverage is achieved by using selected and least number of the mutation operator in selective mutation, *e.g.*, similar mutants are generated by some mutation operator; they are discarded during creation.

Mutation Sampling: The subsets of mutants are arbitrarily selected for testing in mutation sampling. If selected subset is insufficient, another subset is taken [26]. The size of a subset is not previously fixed, it uses Bayesian sequential probability ratio test for appropriate subset sizing.

2.3 Techniques of Mutation Testing

Several techniques have been commenced for cost reduction, time reduction and distinguishing equivalent mutants. These techniques are implemented using some algorithm.

2.3.1 Time reduction using MSG/bytecode Method over Separate Compilation

Ma *et al.* [21] proposed a method known as MSG/bytecode method for reducing the cost of execution of OO programs of MT. Two technologies are used in this method; the one is MSG (Mutant Schemata Generation) and the other is bytecode translation. The existing MSG method is adapted for mutants by which behavior of the program is changed, and bytecode translation is used for the mutants by which structure of the program is changed. The advantage of this method is the performance, as only two compilations are needed, and time for execution and compilation is significantly reduced. A tool for MT is developed according to both the methods and average speed up time is measured. Moreover, they show that MSG/byte code method is faster than separate compilation. Three results were obtained by applying the method. One was a method and mutation operator's collection. The second was a method MSG that is used for testing of OO software, and the last was a tool MuJava for MT and results are obtained by applying the tool. Obtained results depict that MSG and byte code translation can significantly reduce the time of execution over a separate compiled method.

2.3.2 Criteria's for Reduction of Cost and Increase the Relative Strength of the MT

Wong *et al.* [41] described that several test cases are needed for testing code with the help of mutation testing, but all test sets are not good at revealing faults in the code. Cost consideration may require the generation of only one adequate test set with respect to a given code. The cost of the mutation testing is a major bottleneck in its use by practitioners. Two substitutes are used to overcome the issue of cost which can distinguish the equivalent and non-equivalent mutants:

1) **The randomly selected x% mutation criterion**: According to this criterion the x% of mutants of each mutant type are selected arbitrarily, and remaining mutants are ignored. The variation in percent of selected mutants is among 10% and 100%, then keep on increasing with a constant value to examine the cost effectiveness of mutation testing using a different percentage of mutants.
2) **The constrained mutation criterion:** According to this criterion only a few specified types of mutants are examined, and the others are ignored. For example, the constrained abs/ror mutation, examines only abs and ror mutants.

At the time of execution of mutation testing according to above stated criteria, test sets are discarded that are not capable to detect at least one non equivalent mutant. A good selected mutant operator can reduce the examination cost significantly without sacrificing the fault detecting capability of mutation testing.

2.3.3 An Evaluation of Selective Mutation

Mresa *et al.* [22] used Selective mutation technique. This technique approximate the mutation testing that saves execution by reducing the number of mutants that are executed, and effectiveness of selective mutation testing over non-selective mutation testing is reflected. The main cost of mutation testing is incurred when mutants are run against the test suites. The numbers of mutant generated for the program are found to be roughly proportional to the product of number of data references and the number of data objects. For performing selective mutation, selective mutants are created for a program and maximum possible mutants are killed, and the adequacy of outcome is measured [25]. For generating mutants, efficient operators are used in selecting the mutants [22]. Those mutation operators are expected, which generate mutants that require the construction of test cases. Test cases that is capable of killing the mutants at a low

cost of the specified operator and most of other operator's mutant also [22]. The Goal of selective mutation is trying to use operators that tend to produce mutants having semantically small faults [25]. The results show that the use of efficient operators can provide significant efficiency gains for selective mutation [22]. It has almost same coverage as that of non-selective mutation with significant reduction in cost.

2.3.4 Mutation Clustering

Mutant's size will get reduce by selecting the mutants from each cluster. The more we cluster the mutants, the more we lower the computational cost [7, 12]. Adequate test sets are generated for the mutants that are selected from the clusters [7]. Both the mutants and test set can be significantly reduced due to integration of mutation analysis and clustering of data [12]. *K*-means clustering algorithm and the agglomerative clustering algorithm are used for clustering. There are many variations in *k*-means algorithm; one of these variations is to select good initial values where as 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. Mutants can be represented by the domain of variables to determine the centroid of a cluster [12]. One of the mutants from each cluster is selected randomly for both algorithms and generates the test set for selected mutant using a greedy approach [7]. CBT and DDR methods are used for the generation of test cases automatically [12]. Both the methods focus on the domains of variables. CBT takes the algebraic expressions as constraint and DDR takes initial test set for each input [12]. Keeping the test set fix, a new test set is generated of same size for the other mutants in the cluster and at last efficiency for each test set is analyzed based on the mutation score [7]. Mutants those are clustered together, subsequently reduce the number of the mutants.

2.3.5 Sufficiency of Mutants

Namin *et al.* [23] considered a smaller subset of mutation operators instead of considering the larger number of mutation operators. That smaller subset can model the behavior of a full set. They proposed statistical techniques for variable reduction, model selection and nonlinear regression. The techniques are:

i) ***Model Selection Algorithms:*** These algorithms construct and search all possible best-fitted linear models that attempt to predict detection ratio of the mutants more accurately.

ii) ***Variable Reduction Techniques:*** These techniques are used to reduce the number of predictor variables to a smaller set by avoiding less important predictors. It repeatedly, identifies a pair of highly similar variables and eliminates the one that generates more mutants, stop when all pairs fall below a certain correlation threshold.

iii) ***Nonlinear Regression Approach:*** By nonlinear regression approach, the construction of a complementary set of nonlinear models is done in addition to the linear ones.

2.3.6 Decreasing the Cost of MT with Second-Order Mutants

Polo *et al.* [30] has presented a technique to decrease the cost of mutation testing by means of combining the first-order mutants. Although powerful, mutation is a computationally very expensive testing technique. In fact, its three main stages (mutant generation, mutant execution and result analysis) require many resources to be successfully accomplished. Thus, researchers have made important efforts to reduce its costs. It describes the results of two experiments by means of combining the original set of mutants and therefore obtaining a new set of mutants. Results lead to believe that mutant combination does not decrease the quality of the test suite, whereas it supposes important savings in mutant execution and result analysis.

Singla and Kumar [35] worked on mutation operators and they further identified how mutation operators corresponding conditions contributing them equivalent. Further the similar behavior of mutants is converted into groups and a single mutation is generated from the group. This causes the reduction in cost to a significant amount [36, 37].

2.3.7 Higher Order MT

Jia *et al.* [9] introduced the concept of subsuming Higher Order Mutants (HOMs). It is harder to kill HOM than the first order mutants (FOMs). Subsuming HOMs denotes the subtle fault combinations. It is preferable to replace the FOMs with the single HOM. It also removes the number myths related to the mutation testing [11]. 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. A set of test cases that kill HOM also kill each and every FOM. Fewer (but better) mutants mean fewer (but better) test cases [11].

They also introduced the search based approach for the efficient identification of 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 a greedy algorithm, a genetic algorithm and a hill climbing algorithm. The results indicate that the genetic algorithm performs best among all others. However, others also improve the quality of results [11]. The higher order mutation testing may turn out to be far more scalable than first order mutation testing.

2.4 Tools of Mutation Testing

Automated tools are used to increase performance of mutation testing. A tool is the best way to test practically, as it gives proof of experimental results and concepts. Moreover, the effectiveness of any approach is shown by that tool [1]. Many mutation testing tools have been built for supporting automated mutation analysis. There are near around 36 implemented mutation tools of all categories.

A. Proteum

Proteum (Program testing using mutants) is a mutation testing tool for c language, that implements 108 mutation operators, and for a small c program with only 137 non-blank, and non-comment lines of code, it generate 4,937 mutants [42]. These operators are divided into 4 categories: statement, operator, variable, and constant mutation operators. Therefore, generating and executing large number of mutants by Proteum is very costly and time consuming.

B. Proteum/AJv2

It is a first unit mutation testing tool for both Java and AspectJ programs, as well as AspectJ-specific mutations [14]. Proteum/AJv2 automates the application of unit, and AspectJ-specific mutation operators to both Java, and AspectJ programs. This is done through graphical user interface. It has two main features: first is the AspectJ-specific mutation operator, it supports unit mutation to both Java, and AspectJ programs. The other feature it brings is the graphical user interface (GUI) that supports comprehensive test project management.

C. Proteum/FL

It is a tool for localizing faults using mutation analysis [27]. By default it applies on all mutant operators to generate mutants. To reduce mutant's execution cost, it implements many

optimizations techniques. It supports many mutant operators that are designed for the C programming language. The tool aims to solve the problem of software debugging activity like fault localization. This tool is not work in integrated environment.

D. *JavaMut*

JavaMut is a Graphical User Interface based tool to perform mutation analysis, and it is developed by Chevalley and Fosse [1].With the help of this tool the mutated code can be viewed by the tester. Compile time reflective system (Open Java) is used for implementing this tool. This tool implements 26 mutation operators, from that six are selective, fifteen are class mutation operators, and five are new operators that are created by author. Mutants those satisfy these two conditions are considered by JavaMut; mutants are successfully compiled, the byte code of the main program and mutant is different.

E. Cistron

Cistron is a mutation testing tool for Aspect J programs, and it is based on MuJava tool [34]. Mutant code is generated automatically, on the basis of mutation operators used by this tool. Testing is applied only to selected mutants for reducing cost. This tool identifies equivalent and non- equivalent mutants from live mutants.

F. MuAspectJ

MuAspectJ is developed by Jackson *et al.* for AspectJ programs [34]. Tool's effectiveness is determined in terms of, how fast mutant can be generated and executed. This tool provides the time that it will take to get a result but not the indication of quality of generated mutants.

G. AjMutator

Delamare *et al.* [2] proposed this tool for mutation analysis of pointcut descriptor (PCD). Set of mutation operators are implemented for generating mutants by AjMutator, that introduce faults in the PCDs. Mutants are classified according to the set of join points that are matched by the initial PCD. Automatic classification can identify equivalent mutants for a particular class of PCD. Non-equivalent mutants are also classified according to the matched set of join points. The tool can also run a set of test cases on the mutants to give a mutation score.

H. Jester

Moore [1] developed this tool for mutation testing for programs written in java. Jester only supports mutation operators that can be run on a single Unit (class). Jester supports Unit testing of Java programs with the help of JUnit as a base. The tool cannot apply mutation operators that are designed for other system level or integration features, *i.e.*, inheritance and polymorphism. Jester modifies the original program in a variety of ways, and checks whether tests fails for each modification.

I. Bacterio

The tool automates the tasks to perform mutation analysis and is a standard GUI based application that is developed in java [17]. Set of mutation techniques are implemented for reducing the costs and the execution time of mutant. Bacterio helps testers to evaluate the quality of test cases. The tool automatically does these two main tasks to perform mutation analysis: Firstly, this implements almost all the techniques, and make it feasible to perform a mutation analysis for big, industrial system to reduce the cost of mutation testing, and the other task is, the tool implanted flexible weak mutation, that allows testers not only to perform mutation testing at unit level, but also at system and multiclass levels.

J. Judy

Judy is developed in java with AspectJ extensions, and it is an implementation of the FAMTA Light approach [16]. The main features of the tool are high performance, integration with professional development environment tools, advanced mutant generation mechanism and full automation of mutation testing process. Mutant operators supported by Judy are ABS, AOR, LCR, ROR, UOI, UOD, SOR, LOR, COR, ASR, EOA, EOC, JTD, JTI, EAM and EMM.

K. MuJava

MuJava is a tool for mutation testing that is developed for java programs [28]. The tool is developed with the collaboration between two universities, George Mason University in the USA, and Korea Advanced Institute of Science and Technology (KAIST) in South Korea. It comprises of two tools. One is used for generating mutants, and the other one is for running, testing, and killing mutants. Java Mutation operators are classified as class level mutation operators and Method level mutation operators. MuJava Tool applies both types of operators to generate mutants. A standalone application has been made for making the tool more accessible,

that act as the main menu for MuJava tool. The important feature of MuJava menu is that the output of MuJava can be saved as a textual file for further analysis.

L. MILU

MILU is a higher order mutation testing tool for the c language that is optimized at runtime [10]. Two modes of mutation testing are provided by the tool: traditional mode and higher order mode. Traditional mode is designed for first-order mutation testing. Either predefined mutation operators or their succeeded customized mutation operators can be used by user. Distinguishing the results between the original program, and mutants are also needed to automate the testing process. In higher order mode, either predefined search based optimization algorithm is used or users can specify their own algorithm. A user friendly GUI interface is provided by MILU for users running these two modes. MILU uses a novel 'test harness' technique to embed mutants, and their associated test sets into a single-invocation procedure to reduce the runtime cost.

M. EqMutDetect

It is a tool for detecting equivalent mutants in embedded systems [24]. The tool overcomes the disadvantage of other mutation testing tools that cannot detect equivalent mutants. The measured mutation score is wrong, if equivalent mutants cannot be distinguished. To obtain more accurate mutation score, elimination of equivalent mutants is necessary. EqMutDetect is based on a constraint representation of the program, mutation, and constraint solving. The tool allows for computing a distinguishing test case that would kill a mutant. If there is no such test case that kills a mutant, then the tool identifies an equivalent mutant. The tool is only used effectively for measuring quality of test cases for embedded system, and for smaller programs. Because of the limited size of the programs, the scalability of the tool is an open issue.

N. SMT-C

It is a semantic mutation testing tool for C language [4]. SMT is used to capture errors caused by possible misunderstandings of the semantics of the language. Mutants generated by SMT-C are the misunderstandings of the C semantics. TXL is a high-level transformation tool that is used for generating semantic mutants. Mutant generation in SMT-C is highly portable. Nineteen semantic mutation operators are developed in c language for generating mutants. The tool provides an easy to use front-end.

Chapter3

Problem Statement

This chapter includes the problem statement and objectives of the dissertation.

3.1 Problem Statement

MSG/ byte code translation method is faster than separate compilation method, but the speed can still be improved. Thus, we can say that these approaches requires more time for execution of all the mutants. All tools defined so far are only for one language like Cistron, MuAspectJ is for AspectJ programs and MuJava, Jester, JavaMut are for Java language and Proteum, MILU, and SMT-C are only for C language.

By developing an effective algorithm we can provide less time for the execution of mutants. So that it could provide faster performance than other existing algorithms.

3.2 Dissertation Objectives

- We propose an approach that uses string matching algorithm that compares the string and then according to this, mutants are generated using predefined mutation operators.
- Our approach reduces the total time required for executing all mutants. Thus, it improves performance of mutation testing.
- Our approach is used for both Java and C language.

Thus, we will use string matching algorithm that reduce the cost and time consumption.

Chapter4

Proposed Work

This chapter contains the details of work that has been done for solving the problem that is stated in the problem statement. The proposed technique helps to solve the problem of large time and cost consumption for executing the mutants.

4.1 Working of Mutation Testing

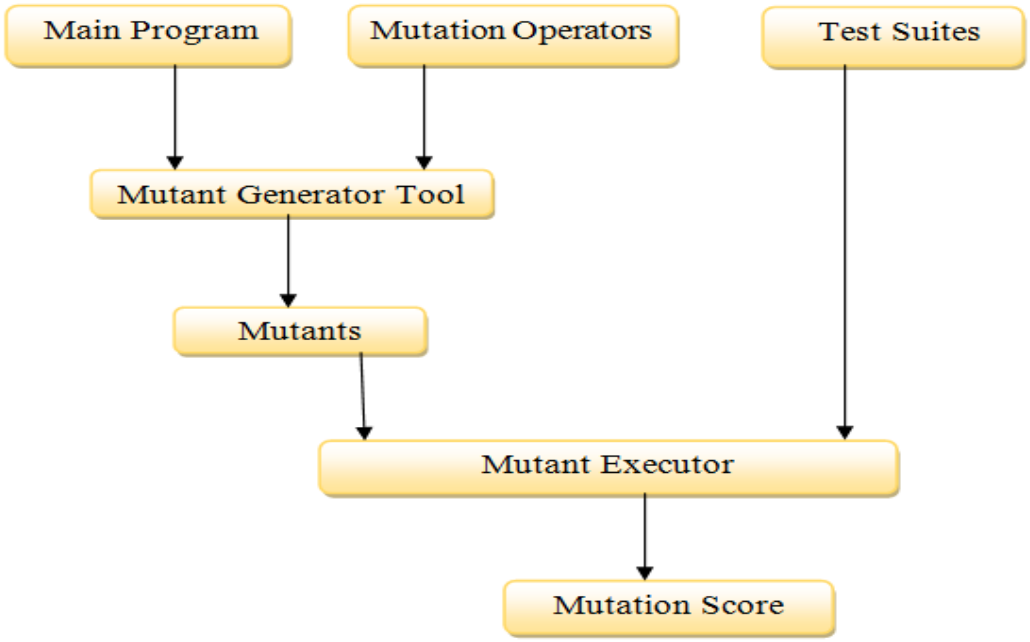


Figure 4.1: Working of Mutation Testing [1]

For performing mutation testing, initially mutants are generated from the main program. A mutant is different from the main program by single fault that is correct according to the syntax [32]. These bugs are introduced with the help of already defined set of mutation operators. After mutant generation, all the test cases are executed related to entire mutants and output is compared with original program's output. If the main program's output is different from the produced output, then that mutant is dead else that mutant is said to be live or not killed. If there is no test case that can differentiate the main program's output with the mutants, then that mutant is called equivalent mutant. Here we have not considered about equivalent mutant.

4.2 Steps for Mutation Analysis

For achieving mutation analysis three steps are executed [27]:

1) **Generation of mutants**

   Multiple copies of the program are created with small syntactic changes in the main program, which are called mutants.

2) **Execution of mutants**

   Entire test suites are executed corresponding to the main program, and whole mutants. Moreover, killed and live mutants are detected.

3) **Result analysis**

   Mutation score is calculated that gives the quality of the executed tests.

4.3 Proposed Technique

In the proposed technique, a tool *MuString* for mutation testing is developed using string matching algorithm that significantly reduce the total execution time of mutants. Mustring tool provides fast performance than other existing previous systems. MuString tool works on both Java and C language. While other tools like Proteum, MILU, and SMT-C are used only for C language. Cistron, MuAspectJ are used for AspectJ programs. MuJava, Jester, JavaMut are used for Java language.

For any program P, each mutant of P is formed according to the result of single changes to some statement in P. Thus, each mutant of P differs from the main program by only one mutated statement. The manner in which the statements are modified is defined by the collection of mutation operators applied.

For example, the operator AOR (arithmetic operator replacement) replaces each occurrence of an arithmetic operator by each of the other possible arithmetic operators. If in program there is a statement L+M then L +M yields the following four mutated statements:

Result = L - M;

Result = L * M;

Result = L /M;

Result = L % M;

In general we can represent these mutations as

Result = L AOR M;

Or we can represent this as

Result=AOR (L, M);

Thus, other operators like relational, logical, unary operators can also be used like this.

4.4 Proposed Algorithm

This section explains how this algorithm works. We are providing a pseudo code which explains the working of this algorithm. Input programs are applied on MuString tool and give a reduction in time for execution of mutants and effective mutation score.

Table 4.1: Proposed Algorithm

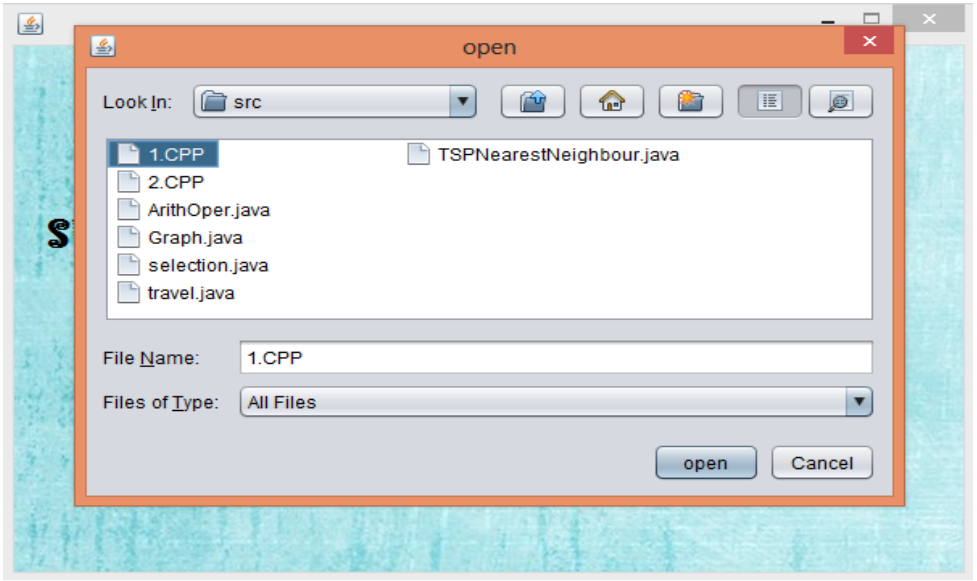
Input: Programs in C or Java language and values of variables. **Output:** Efficient time of execution of mutants and mutation score *Algorithm: Mustring_algo*
Step 1: Initially, the whole program is taken into strings. And those strings are stored in string builder class. **Step 2:** String Arraylist is used, where we divide whole source code into subparts based on newline and stores it in an Arraylist. **Step 3:** Then every index is taken one by one and applies string match function that is *String. Equal* with ignore case function (consider both upper and lower case as the same) using the stored operators that are in another Arraylist. **Step 4:** Based on the found values, we store the values in Master Arraylist. **Step 5:** After storing the values in Master Arraylist, test cases as well as mutants are displayed from Master list that are generated after applying stored operators. **Step 6:** Finally mutant generation and execution time is displayed. **Step 7:** Adequate Mutant Score is calculated using the formula.

4.5 MuString tool

Initially, when we execute the code following window appears.

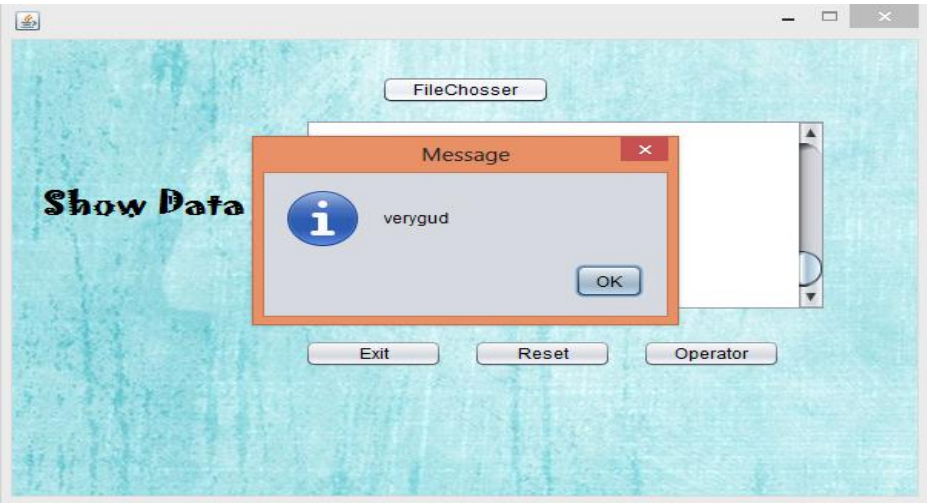
Snapshot 4.1: Initial Window

Then, after clicking on File chooser we can choose a program that can be in C or Java language.



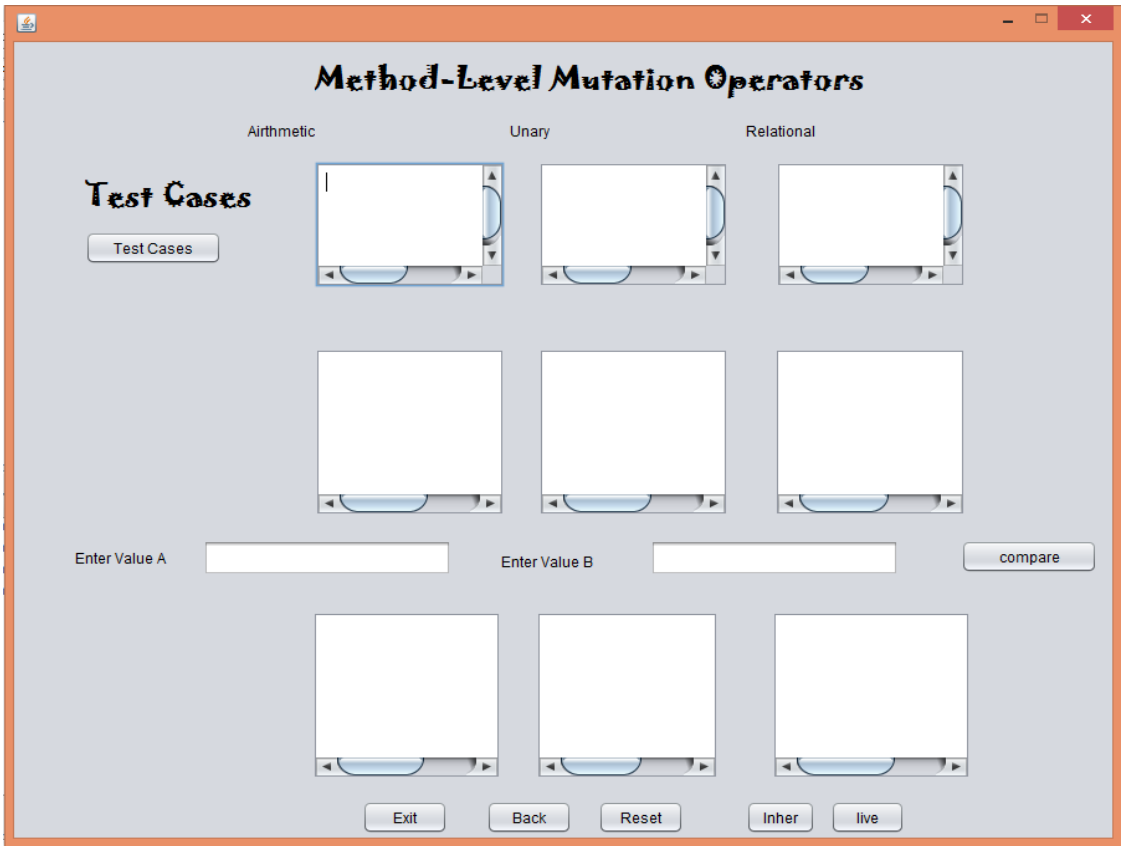
Snapshot 4.2: Choosing a Program

After choosing the program if it is in correct form, *i.e.*, in C or Java language, then the dialog box appears with “Very Good”, otherwise it gives “Wrong file”.



Snapshot 4.3: Checking of Program

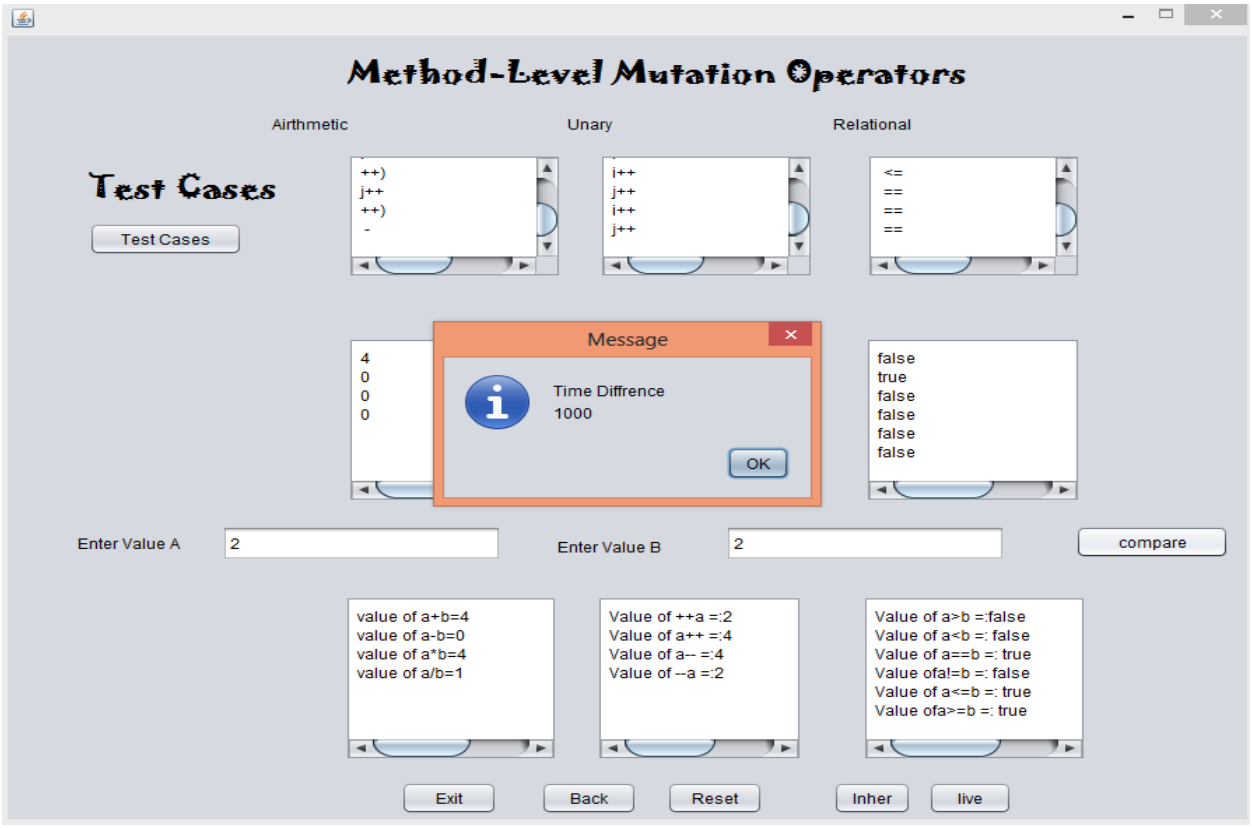
After selecting a program we click on the operator, it will give following window of Method Level Operators.



Snapshot 4.4: Method Level Mutation operators

Then, string comparison algorithm start working and the tool will provide all the mutants that are possible for that program. The mutants for method level operators can be arithmetic, unary, logical and relational. Mutants are generated, and time for execution of mutants is displayed.

Values of variables are given at runtime after mutant generation, and then values of all mutants are generated by the tool. Moreover, it compares the value of the program with mutants.



Snapshot 4.5: Execution time for Mutants

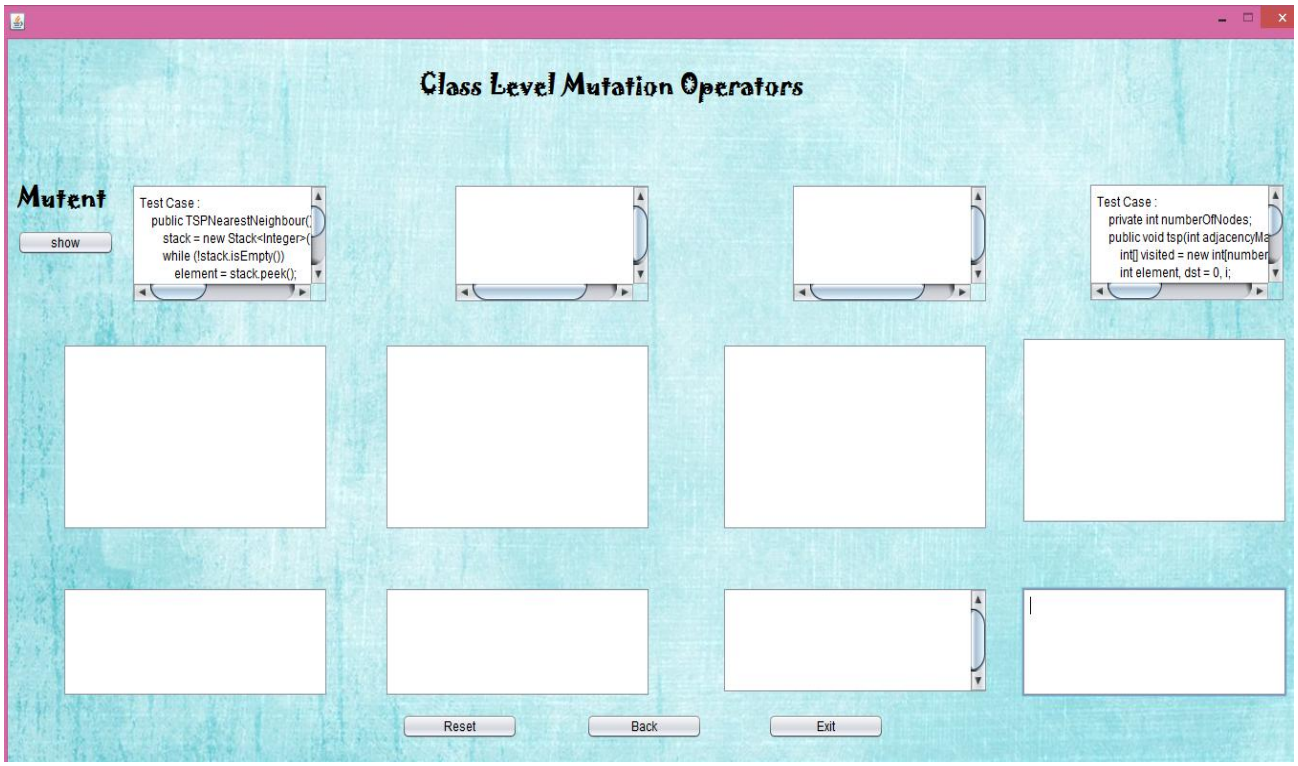
After this, if we want to know how many mutants are killed, or how many mutants are live then we click on live button and it displays live and killed mutants for all defined types of operators and mutation score is calculated according to following formula.

$$Mutation\ Score = \frac{100 * Dead\ Mutants}{Total\ no.\ of\ mutants - no.\ of\ equivalent\ mutants}$$

Here we are not considering equivalent mutants, so we consider its value as 0.

Snapshot 4.6: Live and Killed Mutants

Then after click on Inh button, we can get class level mutation operators as following:



Snapshot 4.7: Class Level Mutants

Our tool with string matching algorithm gives adequate mutation score with low execution time as compared to previous algorithms.

4.6 Experimental Results

Ma *et al.* used MuJava tool, and MSG/byte code translation method to reduce the total execution time of all mutants [21]. Moreover, a significant amount of time was reduced by MSG/byte code method over the previous method, *i.e.*, Separate Compilation method. Table 4.1 shows the total execution time for separate compilation. 7 programs are compared for all the three methods. Program P1 is RSA, P2 is PlayFair encryption algorithm, P3 is Substitution algorithm, P4 is Snake and Ladder program, P5 is Chessboard program, P6 is DES (Data Encryption Standard) algorithm, P7 is AES (Advance Encryption Standard).

Table 4.2: Total Execution Time of Mutants for Separate Compilation Approach

Program	No. of Lines	Language	Total Execution Time
P1	80	Java	5,852
P2	126	Java	6,727
P3	80	Java	9,634
P4	319	Java	22,183
P5	205	Java	31,598
P6	380	Java	42,942
P7	533	Java	64,079

The execution time shown in tables is in milliseconds. From Table 4.2, it can be observed that the total execution time for separate compilation approach is very large. So MuJava tool with MSG/Byte code translation method is used for reducing this time. Table 4.3 shows the total execution time for MSG/Bytecode translation method. Speedup is calculated by dividing the time for separate compilation, by the time for the MSG method.

$$Speedup = \frac{Time\ for\ Separate\ compilation\ approach}{Time\ for\ MSG/bytecode\ Method}$$

Table 4.3: Total Execution Time of Mutants for MSG/bytecode Approach

Program	No. of Lines	Language	Total Execution Time	Speedup
P1	80	Java	1,671	3.50
P2	126	Java	2,038	3.30
P3	80	Java	3,222	2.99
P4	319	Java	4,983	4.45
P5	205	Java	8,476	3.73
P6	380	Java	5,110	8.40
P7	533	Java	7,696	8.33

From Table 4.3, it can be observed that the total time to generate and execute mutants with the MSG/Bytecode translation approach was faster than the separate compilation approach. However, the time taken by MSG/Bytecode method was also very large. So we used another algorithm, *i.e.*, string matching algorithm with MuString tool by that generation and execution time of mutants is reduced.

Table 4.4: Total Execution Time of Mutants using String Matching Algorithm

Program	No. of Lines	Language	Total Execution Time	Speedup
P1	80	C	500	3.34
P2	126	Java	1,500	1.36
P3	80	Java	1,000	3.22
P4	319	C	1,500	3.32
P5	205	Java	2,500	3.39
P6	380	C	2,000	2.56
P7	533	Java	4,000	1.92

Table 4.4 shows the total execution time for string matching algorithm. Moreover, it can be observed from the tables that execution time is significantly reduced using string matching algorithm over both previous methods.

$$Speedup = \frac{Time\ for\ MSG/bytecode\ approach}{Time\ for\ String\ Matching\ algorithm}$$

Here, Speedup is calculated by dividing the time for MSG method, by the time for the String matching algorithm.

4.6.1 Result Analysis

From the above results, the following analysis has been made about this algorithm.

- Results obtained from this algorithm prove its efficiency.
- We have received reduced time over both previous algorithms.

Table 4.5: Comparison of Proposed Approach with the Previous Approaches

Program	No. of Lines	Methods	Total Execution Time using Separate Compilation Approach	Total Execution Time using MSG/byte code Approach	Total Execution Time using String Matching Algorithm
P1	80	5	5,852	1,671	500
P2	126	9	6,727	2,038	1,500
P3	80	9	9,634	3,222	1,000
P4	319	14	22,183	4,983	1,500
P5	205	33	31,598	8,476	2,500
P6	380	42	42,942	5,110	2,000
P7	533	37	64,079	7,696	4,000

From Table 4.5, it can be observed that the total execution time of mutants is significantly reduced using String Matching Algorithm on MuString tool. Here time is in milliseconds. For all programs, there are different-different lines of codes and methods. The corresponding total

execution time for all the programs using separate compilation approach, MSG/byte code approach, and String Matching algorithm is compared. Results obtained depict the efficiency of String Matching Algorithm using MuString tool. The efficiency of algorithm can also be observed from the graph as shown in Figure 4.2.

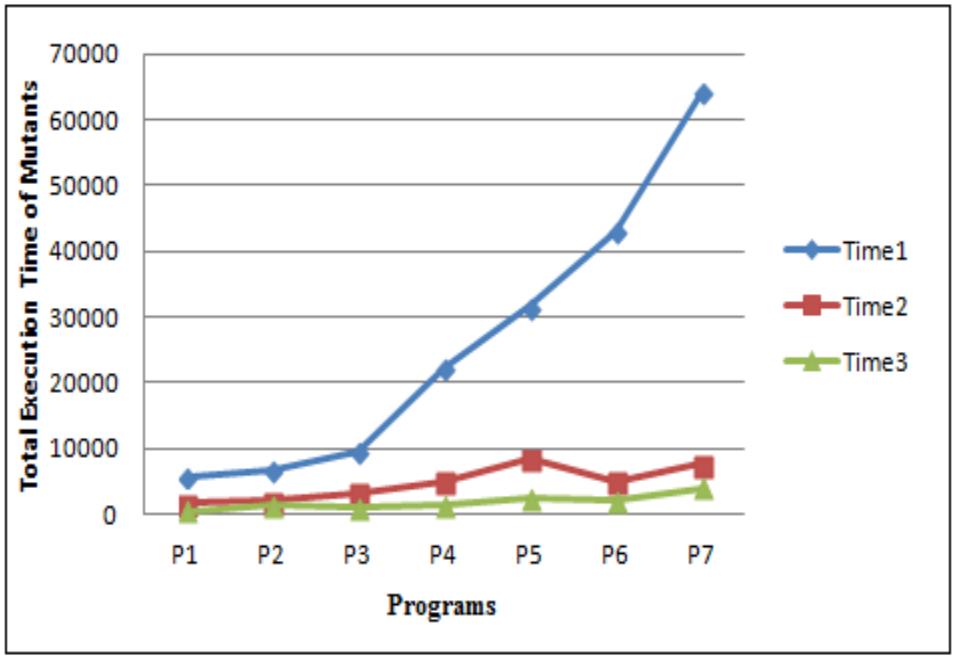


Figure 4.2: Comparison Analysis of Proposed Technique with Previous

- Figure 4.2, Time1 (Blue color line) is showing the total execution time for all programs using Separate Compilation approach. Time2 (Red color line) is showing total execution time using MSG/Bytecode approach. Moreover, Time3 (Green color line) is showing the total execution time for all programs using String Matching Algorithm.

- Figure 4.2 shows the efficiency of String Matching Algorithm as it can be observed that the execution time is reduced for all programs as compared to the existing previous approaches.

Chapter5

Conclusion and Future Scope

5.1 Conclusion

Mutation Testing is a fault based technique that works in conjunction with the traditional testing techniques. It is done by generating faults and observes the effect by going through three conditions: reachability, necessity and sufficiency. However, it has few main problems: detection of equivalent mutant, large effort, time and cost consumption.

This dissertation presents an approach for solving some of these problems at a great extent. The proposed approach is inexpensive for mutation testing. Mutants are generated using method level mutation operators, and class level mutation operators. A new tool *MuString* is developed that use mutation operators and *String matching algorithm* to execute mutants.

By using String matching algorithm with MuString tool, it reduces the total execution time required to execute the mutants. Moreover, the mutation score is calculated for mutants. A mutant is killed; if it gives different output from the original program, otherwise it is live. The proposed algorithm helps in reducing time. Additionally, it helps in the calculation of the mutation score more accurately.

5.2 Future Scope

In future following works can be done:

- An automated technique can be developed to remove equivalent mutants at the time of their creation.
- The proposed method can only deal with numeric data. This is partly, because generating boundary values are relatively difficult for non-numeric data. For example, what is the "next" value of "a"? It depends on the ordering rule. It may be b or A.
- This technique does not consider all mutation operators like shift, bitwise XOR, bitwise OR and many more. So modifications can be done so that it can work on all operators which are left out.
- There is a scope of integration of mutation testing with other types of testing like unit and integration testing.

REFERENCES

[1] Bashir M. B., and Nadeem A., "Object oriented mutation testing: A survey", *IEEE International Conference on Emerging Technologies, ICET*, 2012.

[2] Baudry B., and Traon Y. L., "AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors", *IEEE International Conference on Software Testing Verification and Validation Workshops*, 200-204, 2009.

[3] Beizer B., "Sotware testing techniques", *Dreamtech Press*, 2003.

[4] Dan H., and Hierons R. M., "SMT-C: A Semantic Mutation Testing Tool for C", *IEEE 5th International Conference on Software Testing, Verification and Validation*, 654-663, 2012.

[5] Delamaro M. E., Maidonado J. C., and Mathur A. P., "Interface mutation: An approach for integration testing", *IEEE Transactions on Software Engineering*, 27(3), 228-247, 2001.

[6] Ferrari F. C., Maldonado J. C., and Rashid A., "Mutation testing for aspect-oriented programs", *International Conference on Software Testing, Verification, and Validation*, 52-61, 2008.

[7] Hussain S., "Mutation clustering", *M. S. Thesis, King's College London, Strand, London*, 2008.

[8] IEEE Standards Association, "829-1998 IEEE Standard for Software Test Documentation", 1998.

[9] Jia Y., and Harman M., "Constructing Subtle Faults Using Higher Order Mutation Testing", *Conference on Source Code Analysis and Manipulation, SCAM*, 249–258, 2008.

[10] Jia Y., and Harman M., "MILU: A Customizable, Runtime- Optimized Higher Order Mutation Testing Tool for the Full C Language", *Testing: Academic & Industrial Conference – Practice and Research Techniques*, 94-98, 2008.

[11] Jia Y., and Harman M., "Higher order mutation testing", *Information and Software Technology*, 51(10), 1379-1393, 2009.

[12] Ji C., Chen Z., Xu B., and Zhao Z., "A Novel Method of Mutation Clustering Based on Domain Analysis", *International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2009.

[13] Khan M., "Different Forms of Software Testing Techniques for Finding Errors", *International Journal of Computer Science Issues, IJCSI*, 7(1), 11-16, 2010.

[14] Leme F. G., Ferrari F. C., Maldonado J. C., and Rashid A., "Proteum/AJv2: A Mutation-based Testing Tool for Java and AspectJ Programs".

[15] Li J., Dai G., and Li H., "Mutation analysis for testing finite state machines", *Second IEEE International Symposium on Electronic Commerce and Security, ISECS*, 1, 620-624, 2009.

[16] Madeyski L., Radyk N., "Judy- A Mutation Testing Tool for Java", *IET Software*, 4, 32-42, 2010.

[17] Mateo P. R., and Usaola M. P., "Bacterio: Java Mutation Testing Tool", *28th IEEE International Conference on Software Maintenance, ICSM*, 646-649, 2012.

[18] May P., Timmis J., and Mander K., "Immune and evolutionary approaches to software mutation testing", *Artificial Immune Systems, Springer Berlin Heidelberg*, 336-347, 2007.

[19] May P., Mander K., and Timmis J., "Software vaccination: An artificial immune system approach to mutation testing", *Artificial Immune Systems, Springer Berlin Heidelberg*, 81-92, 2003.

[20] Ma Y. S. & Offutt, J. "Description of Class-level Mutation Operators for Java", 2005.

[21] Ma Y. S., Offutt J., and Kwon Y. R., "MuJava: an automated class mutation system", *Software Testing, Verification and Reliability*, 15(2), 97-133, 2005.

[22] Mresa E. S., and Bottaci L., "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study", *Software Testing, Verification and*

Reliability, 9(4), 205–232, 1999.

[23] Namin A. S., and Andrews J. H., "On Sufficiency of Mutants", *IEEE International Conference on Software Engineering-Companion, ICSE*, 73–74, 2007.

[24] Nica S., and Wotawa F., "EqMutDetect—A tool for equivalent mutant detection in embedded systems", *Intelligent Solutions in Embedded Systems, WISES*, 57-62, 2012.

[25] Offutt A. J., Rothermel G., and Zapf C., "An experimental evaluation of selective mutation", *Proceedings of the 15th international conference on Software Engineering, IEEE Computer Society Press*, 100–107, 1993.

[26] Offutt A. J., and Untch R. H., "Mutation 2000: Uniting the orthogonal", *Mutation testing for the new century, Springer US*, 34-44, 2001.

[27] Papadakis M., Delamaro M. E., and Traon Y. L., "Proteum/FL: A tool for localizing faults using mutation analysis", *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, 94-99, 2013.

[28] Pecev P., Markoski B., Ratgeber L., Lacmanovic D., and Lvankovic Z., "Making MuJava more accessible", *13th IEEE International Symposium on Computational Intelligence and Informatics*, 147-150, 2012.

[29] Polo M., Tendero S., and Piattini M., "Integrating techniques and tools for testing automation", *Software Testing, Verification and Reliability*, 17(1), 3-39, 2007.

[30] Polo M., Piattini M., and Garcia-Rodriguez I., "Decreasing the Cost of Mutation Testing with Second-Order Mutants", *Software Testing, Verification and Reliability*, 19(2), 111–131, 2009.

[31] Praphamontripong U., and Offutt J., "Applying mutation testing to web applications", *Third IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW*, 132-141, 2010.

[32] Pressman, Roger S., "Software Engineering: a practitioner's approach", *Pressman and Associates*, 2005.

[33] Reda S., and Orailoglu A., "Reducing test application time through test data mutation encoding", *Proceedings of the Design, Automation and Test in Europe IEEE*

Conference and Exhibition, 387-393, 2002.

[34] Singh P. K., Sangwan O. P., and Sharma A., "A Systematic Review On Fault Based Mutation Testing Techniques and Tools for Aspect-J Programs", *IEEE 3rd International Advance Computing Conference, IACC*, 1455-1461, 2013.

[35] Singla T., Kumar A., "Mutation Operator corresponding Conditions Contributing in Deporting them Equivalently", *IJCST*, 4(2), 656-658, 2013.

[36] Singla T., Kumar A., and Garhwal S., "Reducing Mutation Testing Endeavor using the Similar conditions for the same Mutation Operators occurs at Different Locations", *Applied Mathematics & Information Science*, 8(5), 2389-2393, 2014.

[37] Singla T., "Reducing Mutation Testing Endeavor using the Similar conditions for the same Mutation Operators occurs at Different Locations, *M.E. Thesis*, 2013".

[38] Shokouhifar M., and Sabet S., "PMACO: A pheromone-mutation based ant colony optimization for traveling salesman problem", *IEEE International Symposium on Innovations in Intelligent Systems and Applications, INISTA*, 1-5, 2012.

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

[40] Wang R., Guo N., Xiang F., and Mao J., "An improved quantum genetic algorithm with mutation and its application to 0-1 knapsack problem", *IEEE International Conference on Measurement, Information and Control, MIC*, 1, 484-488, 2012.

[41] Wong W. E., and Mathur A. P., "Reducing the Cost of Mutation Testing: An Empirical Study", *Journal of Systems and Software*, 31(3), 185–196, 1995.

[42] Zhang L., Gligoric M., Marinav D., and Khurshid S., "Operator- Based and Random Mutant Selection: Better Together*", Automated Software Engineering, ASE,* 92-102, 2013.

[43] Zhou C., and Phyllis F., "Mutation testing for Java database applications", *IEEE International Conference on Software Testing Verification and Validation, ICST*, 396-405, 2009.

PUBLICATIONS

Accepted

1. Mittal R., and Garhwal S., "Applications of Mutation Testing: A Systematic Review", *International Conference on Industrial Electronics and Computer Science, ICIECS,* 2014.

Communicated

1. Mittal R., and Garhwal S., "Techniques and Tools of Mutation Testing: A Systematic Review", *5th International Conference Confluence, The Next Generation Information Technology Summit, Amity University,* 2014.