A Comparative Study of various Mutants Reduction Techniques

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

Master of Engineering

in

Computer Science and Engineering

Submitted By

Rajat Mittal

(Roll No. 851232007)

Under the supervision of:

`

Dr. Ajay Kumar Loura

(Assistant Professor in CSE)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

August 2015

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"AComparative Study of various Mutants Reduction Techniques"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and 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 Dr. Ajay Kumar Loura 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.

Rajat Mittal

(Rajat Mittal)

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

30/8/15

(Dr. Ajay Kumar Loura)

Assistant Professor

Thapar University,Patiala

Countersigned by:

(Dr. Deepak Garg)

Head of Department

Computer Science and Engineering

Thapar University, Patiala

(Dr. S.S.Bhatia)

Dean (Academic Affairs)

Thapar University, Patiala

// Acknowledgement

I would like to take this opportunity to express our gratitude towards all the people who have helped me in various ways in the successful completion of my thesis work.

I must convey my gratitude to my guide **Dr. Ajay Kumar Loura** , Assistant Professor ,CSE department for giving me the constant source of inspiration and help in preparing the thesis, personally correcting my work and providing encouragement throughout the thesis.

I am also thankful to **Dr. S.S Bhatia**, Dean of Academic Affairs, **Dr. Deepak Garg**, Head of Computer Science & Engineering Department and **Mr. Ashutosh Mishra**, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I also thank all my faculty members for steering us through the tough as well as easy phases of the thesis in a result oriented manner with concern attention. I was very fortunate to have an unconditional support from my family. I thank my parent who gave me courage to get my education, supported me in all achievements throughout my life. Last but not least; I would like to thank God for giving me inner strength and courage to achieve my success.

Thanking You

Rajat Mittal

Abstract

Generating huge number of mutants program as possible combinations of various operators, functions, and statements is a great challenge towards the way to success of mutation testing techniques. Programmers and software developers' tries to escape mutation testing cumbersome and expensive procedure due to this reason. If we succeed to decrease the amount of mutants program without disturbing the quality of mutation testing results then it may prove great technique for software testing as well as popular among programmers.

Researchers tries to develop such techniques by which we can decrease the amount of mutants' programs significant level without affecting the quality of the mutation testing. Among mutants reduction techniques; Sampling method, Clustering Method, Selection Method and Higher Order Mutants are very popular. Rating these 4 available techniques on the basis of their mutants reduction capabilities is still a research problem.

In this research we successfully demonstrate and compared the above 4 mutants' reduction techniques. We find the sampling technique outperformer amongst all that could decrease the amount of mutants up to 90.42% while the cluster method could reduce only 69.42% mutants. The Selection method scored 81.52 % reduction score which is greater than cluster method and lower than sampling method. We also compare the relative performances amongst the mutants' reduction methods. We conclude that:

A) Sampling method is 21% more efficient than Cluster method and 8.9% more efficient than Selection method and ;

B) Selection method is 12.1% more efficient than Clustering method. Thus the cluster method is the most expensive and sampling method is most economic amongst the all 3 tested mutants' reduction methods.

Key words: Mutants, Selection Method, Sampling Method, Cluster Method, HOM

Table of Contents

Certificate i

Acknowledgement ii

Abstract iii

Table of contents ....iv-v

Chapter 1: Introduction 1

1.1Mutation Testing…………………………………………………………………… 1

1.2 Various types of mutation testing ………………………………..……….…..... 1

1.3 Software Testing…………………………………………………………………1

1.4 Black box Testing ……………………………………………………….... 3

1.5 Grey box Testing……………………………………………………………….. 3

1.6 Mutation Testing……..………………………………………………………….. 4

1.7 Types of Mutant………………………………………………………………… 5

1.8 Mutation Score………………………………………………………………….. 7

1.9 Test Data Conditions for Killing Mutants………………………………………. 7

1.10 Types of Mutation Testing……………………………………………………….. 7

1.11 Method Level Mutation Operators…………………………………………… 8

Chapter 2: Literature Review…………………………………………………………...10

2.1 Mutants Reduction techniques…………………………………………………10

Chapter 3: Problem Statement………………………………………………………. 13

3.1 Objectives of research………………………………………………………… 13

3.2 Mutation reduction techniques………………………………………………… 14

3.3 Mutation Cost Reduction Techniques…………………………………………..16

Chapter 4 :Proposed Approach……………………………………………………………18

4.1ResearchMethodology……………………………………………………………..18

4.2 Sampling over program elements and variants ………………………………..19

4.3 Operator Selection: Selective mutants…………………………………………20

4.4 Result and Analysis- Program1……………………………………………….. 22

4.5 Result and Analysis –Program 2……………………………………………… 26

Chapter 5: Results and Discussion…………………….…………………………….....31

5.1 Calculations of Mean Performance of Mutants reduction techniques…………31

Chapter 6: Conclusion and Future scope……………………………………………………… 39

Conclusion……………………………………………………………………….39

Future Scope……………………………………………………………………..39

References…………………………………………………………………………………41

Appendix……………………………………………………………………………………..44

Chapter 1

Introduction

1.1 Mutation Testing

Mutation Testing is a fault based testing procedure which gives a testing measure called the "Mutation sufficiency score".

1.2 Variants Types of Mutation Testing

- **Value Mutations:** An effort to alter the program codes to detect faults in the software programs. We usually Mutation one value to a much larger value or one value to a much smaller value. The most common strategy is to Mutation the constants.

- **Decision Mutations:** The decisions/conditions are Mutation to check for the design errors. Typically, one Mutations the arithmetic operators to locate the defects and also we can consider changing of the some of the operators or complete relational operators and logical operators (AND, OR , NOT)

- **Statement Mutations:** Mutations done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.

1.3 Software Testing

Test arrangement portrays the degree, assets, plan, and methodology of testing exercises in a given undertaking. It likewise recognizes the things, the qualities of those things to be tried, and the individual testing undertakings that are to be performed. Test arranging is started instantly after the prerequisites are managed.

There are two classes of inconveniences in Computer programming: flaws or disappointments. A shortcoming is a situation that causes a practical unit to not be up to snuff in performing its imperative capacity. Disappointments emerge in the framework when the passed on administration veered from the expected administration. Programming testing is a routine of deciding shortcomings in the product. Programming testing is done to verify that the rightness, culmination and nature of all product programs must meet. Testing is

performed to check that the calculation utilized as a part of the project is right, and execute accurately in every conceivable condition and it satisfy all the predetermined necessities. Programming testing is the most essential stage in Software Development Life Cycle (SDLC). It starts from the necessities detail stage, proceed to the organization stage.

1.3.1 Objectives of Software Testing

Software testing follows these objectives:

1. Testing is a practice that executes the program with the motive of discovering errors in the program.
2. Test cases are designed to perform testing, and better test cases are one that has strong probabilities of discovery of faults so far undistinguished errors.
3. A successful test case is one that exposes so far undistinguished errors.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.3.4.1 Software Testing Techniques

Software testing techniques are of two types

1.3.4.2 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.

1.3.4.3 Dynamic Testing

For testing the dynamic behaviour 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.

1.3.4.4 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.

1.3.4.5 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.

1.4 Black-Box Testing

Black-Box Testing is same as behaviour 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 Grey-box Testing

Grey-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 issue situated method. It encourages in blend with the ordinary testing methodologies. It accentuation on the test suits that are taken to examine the projects, not care for other testing methodologies that accentuation on exact working of the system. The principle thought process of MT is to make a superior gathering of experiments instead of attempting to find the flaws that are available in the system. Mutation Testing is the essential testing measure for the evaluation of projects as well as for the test suites [6]. Uncovering of deficiencies in the project is a need, e.g., the ESA Ariane five rocket in the year 1996 not succeed on account of an unchecked lapse, the consequence of national duty framework in Britain contained shortcomings more than 10000 invalid duties in the year 2002.

Mutation testing is firstly prescribed by DeMilloet al. in 1978. It concentrates on measuring the agreeability of a test suite to recognize flaws. The fundamental model on which MT works is that the issues are presented by means of the software engineer itself that imply the screw up that developers over and over do. Such blames deliver an arrangement of harmed projects that are entitled as mutant projects. Mutant projects are not the same as the first program by one and only blame. On both the projects i.e., unique and Mutation, test suites are connected. Our principle thought process is to bring about the mutant project to not succeed, hence measuring the acceptability of the test suite. The Mutant that initiates one and only blame in the system at once is characterized as first-request mutant while the other that begins a few deficiencies or Mutations in the project at once is characterized as higher-request mutant.

Case 1.1:

Unique Program **First-request Mutant**

```
1)      int primary() {            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", whole);

8) return 0; } return 0;}

Here in Example 1.1 system discovers the aggregate of digits of the number and unique project gives yield 6. Be that as it may, if we Mutation one administrator, i.e., supplant % by/then it will give distinctive or flawed yield 13, and it will turn into the first request mutant.
Mutation testing can be characterized in these strides:
Insertion of flaws in the project and building new forms utilizing already characterized Mutation administrators that are called mutant.
Mutant projects are not the same as the first program by one and only blame.

Test suits are connected on the first and mutant projects.
The primary thought process is to bring about the mutant system to not succeed, hence measuring the acceptability of the test suite.

1.7 Types of Mutant

There can be 3 sorts of mutants:
a. Alive/Live Mutant: experiments are not ready to identify the infused deficiency in the project.
b. Comparable Mutant: a mutant that dependably delivers indistinguishable results with the first program.
c. Murdered Mutant: distinctive result created on execution of mutants against with the primary project.

1.7.1 Assumptions in Mutation Testing

Suspicions in Mutation testing are:

i) Competent Developer: software engineer has composed verging on precise system that is needed.

ii) Coupling impact: manages that test suites that crush (murder) straightforward mutant can likewise decimate complex mutant.

1.7.2 Equivalent Mutant

A mutant that dependably delivers indistinguishable results with the first program is called comparable mutant. Acknowledgment of identical mutant is one of the primary issues connected with Mutation testing.

Illustration 1.2:

	Unique Program		Equivalent Mutant
1)	int principle() {		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", aggregate);
8)	return 0; }		return 0;}

There is no all-inclusive experiment which can produce an alternate result from the first and mutant system. In this manner, the first and mutant projects are comparable. These sorts of mutants are characterized as proportionate mutants.

1.8 Mutation Score (MS)

Mutation score is assessed to quantify the proficiency of experiments. The MS is the quantity of recognized flaws over the aggregate number of embedded shortcomings.

MS will be 100% if whole arrangement of mutants would execute or not live and if all the identical mutants are recognized.

1.9 Test Data Conditions for Killing Mutants

Three conditions must be obliged to execute a mutant for fulfilling the test information.
Let a project connotes by P, and a mutant of P on articulation S is implies by M and the test information for P is means by T. The conditions are given as :
i) Reachability condition: mutant is just syntactic Mutation in explanation S, so S must bereachable in light of the fact that alternate proclamations in mutant and unique project are precisely same . In the event that S is not reachable by test information T then, it is difficult to execute mutant M.
ii) Necessity condition: M must come in the state that varies from articulation S of original program P in the wake of applying test information T . It is obligatory that S must be reachable, and conditions of P and M must contrast on articulation S for executing M by test information T.
iii) Sufficiency condition: The last condition of M must be not the same as P. Consequently, the diverse state caused by the need condition must engender through the program's calculation to result in an alternate yield .

1.10 Types of Mutation Testing

Mutation Testing can be delegated Weak Mutation Testing, and Strong Mutation Testing.

1.10.1 Weak Mutation Testing

This satisfies just beginning two states of Mutation Testing

Case 1.3:

Program P	Mutant M with feeble 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){...}

1.10.2 Strong Mutation Testing

Solid Mutation satisfies all the three states of Mutation Testing. Solid Mutation testing is more intense than frail Mutation testing.

Illustration 1.4:

Program P Mutant M with solid 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){...}

1.11 Method Level Mutation Operators

These administrators modify the primary project by supplanting, erasing, embeddings the administrator [39]. A percentage of the administrators are subdivided into parallel, unary and alternate way forms. Subsequently, there are add up to 16 technique level administrator

1.11.1 Arithmetic Operators

Numerical estimations on whole numbers and using so as to skim point numbers are performed number-crunching administrators. In java the accompanying number juggling administrators are upheld:

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

Here "o" connotes operands on which Mutation administrators are connected at system level.

1.11.2 Relational Operators

Social administrators think about the estimation of two operands. In java the accompanying social administrators are bolstered:

- o > o (more prominent than), o >= o (more noteworthy than or equivalent to), o < o (not as much as), o <= o (not exactly or equivalent to), o == o (equivalent to) and o! = o (not equivalent to).

1.11.3 Conditional Operators

Reckonings are connected on the parallel estimations of the operands. In java the accompanying restrictive administrators are bolstered:

- Binary: o || o (restrictive OR), o && o (contingent AND), o | o (bitwise OR), o & o(bitwise AND), and ^ (bitwise XOR).
- Unary: !o (bitwise legitimate supplement).

1.11.4 Shift Operators

Shift administrators control the bits of the first operand in the expression by moving to the estimation of the second operand either to the privilege or the left. In java the accompanying movement administrators are bolstered:

- o << o (marked left move), o >> o (marked right move), and o >>> o (unsigned right move).

1.11.5 Logical Operators

Boolean results are thought about intelligently which are created for expressions that we need to analyse. In java the accompanying sensible administrators are upheld:

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

1.11.6 Assignment Operators

To set the estimations of the operand, task administrators are utilized. On the right hand side of the operand counts are performed, and the worth is doled out to one side hand side operand. In java the accompanying movement administrators are bolstered:

- short-cut: o+= o (expansion task), o-= o (subtraction task), o*= o(multiplication task), o/= o (division task), o%= o (modulus task)

1.11.7 Class Level Mutation Operators

Class level mutants are sorted into four gatherings. Initial three gatherings are taking into account basic components of all item arranged dialects.

Chapter 2
Literature Survey

2.1 Mutants Reduction techniques

There are a few ways to deal with lessening the expense of Mutation examination. These were ordered by Offutt and Untch [23] into three methodologies: do less, more astute, and speedier. The do less methodologies incorporate particular Mutation and mutant inspecting, while feeble Mutation, parallelization of Mutation examination, and space/time Mutation are assembled under the Umbrella of do more intelligent. At last do speedier methodologies incorporate mutant blueprint era systems, code fixing and so on.

The thought of utilizing a subset of mutants was considered alongside Mutation examination itself. Budd [6] and Acree [1] demonstrated that even 10% examining can accomplish 99% precision for the last score. The thought was further examined by Mathur [16], Wong et al. [29, 28], and Offutt et al. [22] utilizing the Mothra [9] Mutation administrators for Fortran. Mathur [16, 28] recommended compelled Mutation where just two administrators were utilized.

Various studies in the past have taken a gander at the relative benefits of administrator choice and arbitrary inspecting criteria. Wong et al. [28] thought about x% choice of every mutant sort with administrator choice utilizing only two Mutation administrators, and found that both accomplished comparable precision and diminishment (80%).

Mresa et al.[17] utilized the expense of discovery of mutants as a method for choice to de ne an arrangement of administrators. They found that if high Mutation score (near 100%) is needed, x% specific Mutation is superior to anything administrator choice, and, alternately, for lower scores, administrator choice would be better if the expense of mutants is considered.

Zhang et al. [32] contrasted administrator based mutant determination strategies with irregular mutant examining. They found that none of the determination strategies are better than arbitrary testing, with the same number of mutants. They additionally observed that uniform examining of Mutations is more e compelling for bigger subjects contrasted with equivalent inspecting of Mutation administrators and the converse is valid for littler subjects.

As of late, Zhang et al. [30] affirmed that inspecting as few as 5% of mutants was adequate for a high relationship (99%) with full Mutation score, while examining even less mutants has great potential for holding a high exactness of expectation. They explored eight testing techniques on top of administrator based mutant choice and found that examining procedures taking into account program segments (routines specifically) performed best.

A few studies have attempted to locate the arrangement of adequate Mutation administrators that decrease the expense of Mutation however keep up connection with the full Mutation score. Offutt et al. [22] recommended a n-specific methodology with regulated evacuation of administrators with most various Mutations. Barbosa et al.[5] gave an arrangement of rules to selecting such Mutation administrators. Namin et al.[18, 26] planned the issue as a variable lessening issue, and found that only 28 out of 108 administrators in Proteum were adequate.

Utilizing just the announcement cancellation administrator was initially recommended by Untch [27], who found that it had the most noteworthy relationship (R2 = 0:97) with the full Mutation score contrasted with other administrator choice routines, while producing the littlest number of mutants. This was further fortified by Deng et al. [10] who characterized erasure for diverse dialect components, and found that a precision of 92% is accomplished while decreasing the quantity of mutants by 80%.

Zhang et al. [30] augment the extent of their study with a much more extensive scope of Mutation methodologies and base our outcomes on a much bigger arrangement of certifiable ventures.

Mutation investigation is a technique for assessing the nature of test suites. It includes delivering a group of mutants, projects with little respect from the first program, and assessing the electiveness of test suites against these mutants [15, 2]. Past exploration [3] proposes that Mutations hence presented act in a design like genuine deficiencies, concerning the trouble of discovery.

One of the hindrances to more extensive selection of Mutation examination is its high computational expense. The arrangement of basic mutants for even a moderate measured system can be extensive, setting aside a few minutes expending.

A noteworthy strain of examination into expense decrease of Mutation investigation is to pick a littler, agent, set of mutants [23, 14] | regularly called the do less approach. This

methodology can be for the most part separated into particular techniques and examining procedures.

Particular Mutation procedures endeavour to choose an agent subset of Mutation administrators in view of heuristics and measurable examination, and apply this subset of administrators to produce mutants as opposed to applying the entire arrangement of Mutation administrators [19, 27]. Late work recommends that utilizing proclamation cancellation alone can be an elective methodology [27].

Inspecting systems try to arbitrarily choose an arrangement of agent mutants. This was explored byAcree [1] and Budd [6], who proposed utilizing just x% of all mutants delivered. Wong and Mathur observed that arbitrary inspecting with proportion as low as 10% could give precise results [14].

Late work [31, 30] has researched the relative benefits of irregular inspecting methodologies and administrator choice. Irregular examining can execute and in addition or superior to anything administrator choice, in these studies; and a technique of either inspecting in view of project components or one consolidating both system component based testing and administrator choice was best.

On the other hand, as pointed out by Zhang et al. [30], has a genuine lacuna in huge scale research, both in the extent of the projects concentrated on, and in the number and differences of projects, which diminishes in all results. This is valid for specific Mutation studies, examining studies, furthermore for similarly more current studies that endeavour to join routines. This is especially troubling if Mutation investigation is to increase more extensive acknowledgment among testing experts. Further, very much a couple of the studies [22, 20, 28, 17] were led on more established programming dialects, for example, Fortran, with administrators particular to the dialect, and are not specifically material to more up to date dialects, for example, Java. At long last, with byte code based Mutation motors like PIT [7] and Javalanche [25] (a do quicker approach for killing the aggregation venture to pick up execution speed), administrators in view of source code Mutation are no more material, and their counterparts in byte code should be recognized and contrasted and different methodologies.

Chapter 3

Problem Formulation

In mutation testing a huge amount of mutants are produced as a candidate for test cases. This proves overheads and cumbersome exercise while applying for mutation testing. Reduction of unnecessary mutants is an immediate urge for the success of mutation testing techniques. In software industry mutation testing still not get the popularity due to its large number of mutants problem.

The quantity of mutant increments in the proportion of system size and lines of codes.

The quantity of mutants can be diminished without influencing the nature of the testing results. This will support the execution of the Mutation testing.

A few mutants' diminishment procedures are accessible in which 3 systems are extremely famous:

1. **Mutants examining**
2. **Mutants Clustering**
3. **Selective Mutants**
4. **HOM (Higher Order Mutants)**

Presently every mutant's diminishment procedures have its own particular advantages and disadvantages. A suitable procedure choice is key path for diminishment of mutants.

Here we proposed a similar investigation of all the 3 noteworthy mutants' decrease strategies.

3.1 Objective of Research

Objectives of Our research is:

a. To study various types of mutants used in programming languages

b. To study various techniques for the reduction of mutants.

c To rate and recommend the available mutant reduction algorithms on the basis of their performance.

3.2 Mutation Reduction Techniques

Mutation testing is one of the costly testing strategy. Real wellsprings of computational expense in Mutation testing is the natural running expense in executing the extensive number of mutants against the test set. There are essentially four sorts of systems to lessen the mutants One of the significant wellsprings of computational expense in Mutation Testing is the inborn running expense in executing the substantial number of mutants against the test set. Subsequently, decreasing the quantity of created mutants without noteworthy loss of test adequacy has turned into a prominent examination issue. For a given arrangement of mutants, M , and an arrangement of test information T , MST (M) signifies the Mutation score of the test set T connected to mutants M . The mutant lessening issue can be characterized as the issue of discovering a subset of mutants M from M , where MST (M) ≈ MST (M). This area will acquaint four procedures utilized with lessen the quantity of mutants, Mutant Sampling, Mutant Clustering, Selective Mutation and Higher Order Mutation.

3.2.1 Mutant Sampling

It is a approach that randomly choose a small subset of mutants from the entire set of mutants. In mutation sampling all possible mutants are generated and select randomly for mutation analysis and the remaining are discarded.

Mutant Sampling is a simple approach that randomly chooses a small subset of mutants from the entire set. This idea was first proposed by Acree and Budd . In this approach, all possible mutants are generated first as in traditional Mutation Testing. x% of these mutants are then selected randomly for mutation analysis and the remaining mutants are discarded. There were many empirical studies of this approach. The primary focus was on the choice of the random selection rate (x). In Wong and Mathur's studies the authors conducted an experiment using a random selection rate x% from 10% to 40% in steps of 5%. The results suggested that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score. This study implied that Mutant Sampling is valid with a x% value higher than 10%. This finding also agreed with the empirical studies by DeMillo et al. and King and Offutt . Instead of fixing the sample rate, Sahinoglu and Spafford proposed an alternative sampling approach based on the Bayesian sequential probability ratio test (SPRT). In their approach, the mutants are randomly selected until a statistically appropriate sample size has been reached. The result suggested that their model is more sensitive than the random selection because it is self-adjusting based on the available test set

3.2.2 Mutant Clustering

It is proposed by the Hussain . Mutant clustering generate all first order mutants into different based on the killable test cases. Each mutant in the same cluster is killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in mutation testing and the remaining are discarded. Domain reduction techniques introduced by the Ji .

The idea of Mutant Clustering was first proposed in Hussain's masters thesis. Instead of selecting mutants randomly, Mutant Clustering chooses a subset of mutants using clustering algorithms. The process of Mutation Clustering starts from generating all first order mutants. A clustering algorithm is then applied to classify the first order mutants into different clusters based on the killable test cases. Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in Mutation Testing, the remaining mutants are discarded. In Hussain's experiment, two clustering algorithms, K-means and Agglomerative clustering were applied and the result was compared with random and greedy selection strategies. Empirical results suggest that Mutant Clustering is able to select fewer mutants but still maintain the mutation score. A development of the Mutant Clustering approach can be found in the work of Ji et al. . Ji et al. use a domain reduction technique to avoid the need to execute all mutants

3.2.3 Selective Mutation

It seeks to find small set of mutation operators that generate a subset of all possible mutants without significant loss of test effectiveness. It was first proposed by Mathur[17]. Offut[18] extended the work by omitting four and six selective mutation operators. Based on Mothra mutation operators divide them into three categories: statement, operands and expressions. The most recent research work on selective mutant was Namin [19-21] by formulating the selective mutant.

A reduction in the number of mutants can also be achieved by reducing the number of mutation operators applied. This is the basic idea, underpinning Selective Mutation, which seeks to find a small set of mutation operators that generate a subset of all possible mutants without significant loss of test effectiveness. This idea was first suggested as "constrained mutation" by Mathur . Offutt et al. subsequently extended this idea calling it Selective Mutation.

Mutation operators generate different numbers of mutants and some mutation operators

generate far more mutants than others, many of which may turn out to be redundant. For example, two mutation operators of the 22 Mothra operators, ASR and SVR, were reported to generate approximately 30% to 40% of all mutants [131]. To effectively reduce the generated mutants, Mathur suggested omitting two mutation operators ASR and SVR which generated most of the mutants. This idea was implemented as "2-selective mutation" by Offutt et al. .
Offutt et al. [190] have also extended Mathur and Wong's work by omitting four mutation operators (4-selective mutation) and omitting six mutation operators (6-selective mutation). In their studies, they reported that 2-selective mutation achieved a mean mutation score of 99.99% with a 24% reduction in the number of mutants reduced. 4-selective mutation achieved a mean mutation

3.2.4 Higher Order Mutant

Mutants can be divided into first order mutants (FOM) and higher order mutants (HOM). FOM are generated only once by applying mutant operator while in HOM are generated by applying mutation operator more than once. Jia and Harman introduced the concept of subsuming HOMs .
It is not easy to kill all the FOMs from which it is constructed. It is preferable to replace FOMs than the single HOMs to reduce the number of mutants. They also introduced the concept of strongly subsuming HOM (SSHOM) which is only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed. It is partially proved by Polo et al .

3.3 Mutation Cost Reduction Techniques

Diminished the quantity of mutants, the computational expense can likewise be lessened by decreasing the mutant execution process. There are three systems to lessen the execution cost that have been considered in writing.

3.3.1 Strong, Weak and Firm Mutation:

Strong mutation is referred as traditional mutation testing which is proposed by DeMillo et al [4]. The mutants are killed only if the output is different from the original program. Optimize the execution of the strong mutation Howden[24] proposed a weak mutation instead of checking mutants after the execution of the entire program, the mutant only need to check immediately after the execution point of the mutant or mutated component. Advantage of

weak mutant is that each mutant does not require a complete execution process; once the mutated component is executed we can check for survival mutants.

Firm mutation was first proposed by Woodward and Halewood[12] in 1988. In firm mutation, disadvantage of strong and weak mutation is removed. It lies between the after execution (weak mutation) and the final output (strong mutation) of the mutation. In 2001 Jackson and Woodward [25] proposed a parallel firm mutation approach for java programs.

3.3.2 Optimization systems for runtime Mutation

In original [26] testing apparatus, translator based method is utilized to improve the Mutation, i.e the aftereffect of a mutant is deciphered from its source code specifically. Mutant advancement is adequate and productive for little mutant projects. To diminish the expense of elucidation, compiler based strategy was proposed [27], in light of the fact that execution of accumulated twofold code is much quicker than translation. In gathered based strategy, the mutant project is ordered into an executable system, then each aggregated mutant is executed by various experiments. Fast confinement because of high gathering expense for extensive projects [28]. DeMillo et al. proposed the new method compiler-incorporated to advance the execution of conventional compiler [29]. The new approach of mutant diagram era decreased the overhead cost of customary mediator based [30-31]. It is difficult to incorporate every one of the mutants, rather than gathering every mutant independently, mutant blueprint produce a Meta projects like "super mutant". This Meta system is have to gather once time to test every mutant. So the expense is ascertained once time assemblage and general run time cost. After assemblage system the new approach is presented as a byte code interpretation strategy which is proposed by the Ma et. al. [32].

Chapter 4
Proposed Work

We propose the selection of best mutant's reduction schema analysis among available 4 mutants reduction techniques i.e.

a) **Mutants sampling**
b) **Mutants Clustering**
c) **Selective Mutants**
d) **HOM (Higher Order Mutants)**

Proper analysis and comparison among the four available techniques can play a key role for mutant's reduction process. As each and every techniques is not suitable in every cases. So it is necessary to analyses critically the above techniques and conclude the results.

4.1 Research Methodology

Statistical analysis, sampling techniques, selection techniques and comparisons are the main strategy for our research. Case studies by taking specific sample program is adopted in C++.

4.1.1 Sampling Criteria

We used several different sampling criteria, some of which has been suggested in the literature before, some which are variants of previously suggested criteria, and a few novel ones. For each sampling criteria, we sampled mutants on a decreasing power scale, sampling 1/2, 1/4, 1/8, 1/16, 1/32, 1/64 of the total mutants.

4.1.2 x% selection

The simplest sampling approach consisted of using x% selection as suggested by Budd [6]. In this criteria, we choose a specific fraction of the complete set of mutants. This criteria also serves as a baseline for verifying the e effectiveness of other criteria.

4.2 Sampling over program elements and variants

Following the suggestion of Zhang et al. [30], we extended x% selection criteria to sample from within different program elements. We sampled in increasing order of scope, line, method and class (project scope is just x% selection). We used the formula by Zhang et al. [30],

Sample(x) = bx + random (0::1)c

to correctly sample decimal numbers. Next, we slightly modified this criteria, and instead of sample () which uses probability to manage the decimal numbers, we applied round() to obtain the nearest whole number, which was used as the number of samples to be chosen from the population. The new strategy ignores program elements with small number of mutants. This strategy can potentially guide mutation generation toward more complex program elements. That is, if the number of mutants dropped below a threshold determined by the mutant reduction ratio, that program element would not contribute any mutants to the final result. Conceptually the idea is that mutants of more complex code have more discriminatory power.

Next, we explored in the other direction, by forcing the program elements to return at least one element by using ceil (). That is, irrespective of the fraction being sampled, the simplest elements always contributed at least one mutant to the total sample, ensuring coverage but also giving priority to complex elements.

4.2.1 Lines per element and variants

Our previous research [13] found that statement coverage was highly correlated with mutation score for a project. This immediately suggests that perhaps choosing one mutant per line may be sufficient to achieve a close approximation of the final mutation score. Further, statement deletion has been researched previously [27, 10] and has been found to decrease the amount of mutants well, with tiny reduction in its effectiveness. This also provides a nice comparison with similar numbers of mutants between operator selection and random sampling of mutants.

We extended sampling to levels coarser than methods, i.e. method, class, and project. We sampled n mutants from a method (class or project), where n is the number of statements in the method.

This provided a test of the hypothesis of how important a mutation sampling's relationship to simple code coverage is. This was extended to class and project scope also.

For the first variant, we applied the x% sampling to the result of the first sampling based on line counts. That is, in the case of methods, we first selected line count mutants each from each method, and applied the x% selection to this result. For example, for 1=2 sampling on methods if one method had 10 lines and another had 20, we sampled 10 mutants from the first, and 20 from the second, and from the combined 30 mutants, we sampled 15. For the second variant, we did the sampling in one go, where we chose line count/x number of mutants from each program element. That is, in the previous example, only 5 mutants from the first and 10 mutants from the second method would be chosen.

4.2.2 One per element

Another simple strategy of sampling we experimented with is to just choose one mutant per program element, in the order line, method, class.

4.2.3 x% selection per operator

This strategy, first suggested by Wong et al. [28] samples an equal percentage of mutants from each operator.

4.3 Operator Selection: Selective mutants

For selective methods, we tried mutation operators suggested by Wong et al. [28], Offutt et al. [20, 10], and, Namin et al. [26]. Since Javalanche [25] utilized operator selection mechanisms, we also compared the Javalanche operators for operator selection. Note that all of these techniques except Javalanche have targeted C programs. Thus, some of these operators may be sensible in C but not in Java. For example, deletion of return statement is tolerated in C, not in Java. Moreover, there were a few operators that were not supported by the PIT, and could not be implemented easily (as mentioned below).

4.3.1 Constrained Mutation

Wong et al. [28]. They selected ROR and ABS from Mothra for mutation analysis. We used operators CB and NC to model ROR operator. There are no comparable operators to ABS in PIT.

4.3.2 E-Selective

Offut et al. [20]. They selected ABS, UOI, LCR, AOR, ROR from Mothra operators. We have used IN, M, CB, and NC. PIT does not have any operator comparable to LCR.

4.3.3 Javalanche

. Javalanche uses Negate Jump Condition, Omit Method Call, Replace Arithmetic Operator, and Re-place Numerical Constant operators. We have used NC, VMC, NMC, M, I C and EMV to model them.

4.3.4 Variable Reduction

Namin et al. [26]. They have tried to reduce the mutation operators of Proteum analysis tool which is for C programs. They suggest 28 operators which many of them are not applicable in Java, and some not in PIT. We used by IN, M, Iand NC.

4.3.5 N-selection

Offut et al. [22]. They suggested removal of n most numerous operators. In our experiment, the order of operators was NMC, NC, RC, DC, RV, IC, CC, EMV, VMC, M, CB, I, RI, RS, ES, and IN. We discarded one at each step and evaluated the effectiveness at each

4.3.6 Statement Deletion

The basic statement deletion was modelled on the work by Deng et al. [10]. The operations on single statements were modelled using VMC, NMC, CC, EMV, and RI for simple statements, and using RC for control structures. RC re-places Boolean conditions with false, resulting in removal of the conditional block. The operator for return values was modelled using RV, which is similar. The operators for a while, for, and if statements were modelled using DC, which replaced the Boolean condition with true, which removed the effect of conditional. The switch statement deletion was modelled using RS which replaced the first 100 labels with a default label, resulting in the switch element being deleted. Due to the constraints of the architecture of PIT only the first 100 labels were replaced. Deleting try/catch was not necessary at byte-code level. Finally, to get an accurate estimation of the

effect of true statement deletion.

The mutants by line, and considered each line a single virtual mutant that is, the entire quantity of mutants is equivalent to the whole number of lines. Further, killing any mutant from a line resulted in marking the virtual mutant for that line as killed. This gave us the mutation score for the virtual statement deletion operator. We note that the approximation of simple statement deletion, especially when arithmetic operators are involved, is not complete. However, the number of mutants produced by M operator is very small. We also note that our approximation does not account for the increase in ease of detection when multiple mutations are combined together due to the coupling effect. This also means that not all the lines may be mutated, since there may be no applicable operators. However, given the constraints of a bytecode based mutation system, we believe that our procedure is reasonable.

4.4 Result and Analysis

We have applied mutant reduction techniques on Program-1 and Program-2 files in Appendix.

In the program we found 93 different versions of mutants which is too high as compared to the line of codes which is merely 25 lines. A suitable mutants reduction operation must be exercised for reduction of cost of mutation testing.

10%0f 94= 9.4

After Rounding off it is 9 .So we have to take about 9 mutants among all available mutants.

Table-1 : No. of different Mutants in Test program-1

Line No.	Syntax of Palindrome program in C++	Operators Mutants	Possible ways of Mutants programs	Data Types Mutants	Possible ways of Mutants programs	Operand / variables Mutants	Possible ways of Mutants programs
1	#include<iostream.h>//						
2	#include<conio.h>//						
3	void main()//			**void**	**5**		
4	{						
5	clrscr();//						
6	inta,b,c,d;//			**int**	**5**	**a,b,c,d**	**4**
7	c=0;//	=	**6**			**c**	**1**
8	cout<<"enter any int :";//						
9	cin>>a;//					**a**	**1**
10	d=a;//	=	**6**				
11	while (a>0)//	>	**6**			**a , 0**	**1, 1**
12	{						
13	b=a%10;//	= , %	**6 , 5**			**A,b, 10**	**3,3,1**
14	c=c*10+b;//	=	**6**			**c,10,b**	**1,1,1**
15	a=a/10;//	= , /	**6,5**			**A,10**	**2,1**
16	}						
17	if (d==c)//	**If , ==**	**2 , 6**			**d,c**	**2,2**
18	{						
19	cout<<"the given number is palindrome						

	" ; //						
20	}						
21	else { cout<<" the number is not palindrome ";//	**else**	**2**				
22	}						
23	cout<<"the reversed number is : "<<c;//					**c**	**3**
24	getch(); //						
25	}						
	Sum Total	**56**		**10**		**28**	
	Total No. of Mutants = 55+10+28= 94						

4.4.1 Reduction of Mutants by Applying Sampling method

We adopt x% sampling method for the above test program. As the program small and no. of total mutants is only 94 so we take 10% of total mutants in randomly for the reduction of mutants .

Table-2 : Mutants categories (Sampling Reduction Technique)						
	Arithmetic Operators	Relational Operators	Program flow Control	Data Types	Constants	Variables
1	2	7	2	2	2	4
Proportionate Distribution of 9 Mutants from each categories						
2	1	3	1	1	1	2
	Total reduced No. of Mutants = 9					

4.4.2 Reduction of Mutants by Applying Clustering Method:

Table-3 : Mutants categories (Cluster method)						
	Arithmetic Operators Mutants	Relational Operators Mutants	Program flow Control	Data Types Mutants	Constants Mutants	Variables Mutants
1	5	42	4	10	2	4
After Clustering						
2	5	6	2	5	2	4
	Total reduced No. of Mutants = 24					

4.4.3 Reduction of Mutants by using Selective Strategy

Statement, operands and expressions

Table-4: Mutants categories (Selection method)			
	Statements	Operands	Expressions
1	5	6	7
	Total reduced No. of Mutants = 18		

4.4.4 Comparisons among Mutants reduction Techniques:

Table-5: Performance Comparisons among Different Mutants reduction Techniques				
		Mutants after reduction	% reduction	
1	Sampling Method	9	90.42%	Best
2	Clustering Method	24	74.46%	Expensive
3	Selection method	18	80.85%	Moderate

4.5 Program 2

Table-6 : No. of different Mutants in Test program-2							
Line No.	**Syntax of Prime Number Program in C++**	**Operators Mutants**	**Possible ways of Mutants programs**	**Data Types Mutants**	**Possible ways of Mutants programs**	**Operand / variables Mutants**	**Possible ways of Mutants programs**
1	**#include<iostream .h>//**						
2	#include<conio.h>/						

	/						
3	#include<math.h>//						
4	void main()//			**void**	**3**		
5	{						
6	intn,i;			**int**	**5**	**n,i**	**2,2**
7	clrscr();						
8	cout<<" enter the number ";//						
9	cin>>n;					**n**	**2**
10	for(i=2;i<n;i++)//	**for , =,< , ++**	**2,6,6,4**			**N,I,2**	**2,2,2**
11	{						
12	if(n%i==0)//	**If,%,==**	**2,6,6**			**N ,i,0**	**2,2,2**
13	{						
14	cout<<" the number is not prime ";//						
15	break;//	**break**	**2**				
16	}						
17	}						
18	if(i==n)//	**If,==**	**3,6**			**i,n**	**2,2**
19	cout<<" the number is prime ";//						
20	getch();//						
21	}						
	Sum Total		**43**		**8**		**22**
	Total No. of Mutants = 43+8+22 = 73						

In the above simple program we found 73 different versions of mutants which is too high as compared to the line of codes which is merely 21 lines. A suitable mutants reduction operation must be exercised for reduction of cost of mutation testing.

Sufficient no. of mutants' calculations

4.5.1 Reduction of Mutants by applying Sampling method

We adopt x% sampling method for the above test program. As the program small and no. of total mutants is only 94 so we take 10% of total mutants in randomly for the reduction of mutants.

10% 0f 73= 7.3

After Rounding off it is 7 .

So we have to take about 7 mutants among all available mutants.

Table—7: Mutants categories (Sampling Reduction Technique)						
	Arithmetic Operators	Relational Operators	Program flow Control	Data Types	Constants	Variables
1	1	4	5	2	4	9
Proportionate Distribution of 9 Mutants from each categories						
2	1	1	1	1	1	2
	Total reduced No. of Mutants = 7					

4.5.2 Reduction of Mutants by Applying Clustering Method

Table-8 : Mutants categories (Cluster method)						
	Arithmetic Operators Mutants	Relational Operators Mutants	Program flow Control	Data Types Mutants	Constants Mutants	Variables Mutants
1	6	24	13	8	2	18
After Clustering						
2	6	6	5	5	2	2
	Total reduced No. of Mutants = 26					

4.5.3 Reduction of Mutants by using Selective Strategy

Statement, operands and expressions

Table-9 : Mutants categories (Selection method)			
	Statements	Operands	Expressions
1	6	4	3
	Total reduced No. of Mutants = 13		

4.5.4 Comparisons among Mutants reduction Techniques: Test program-II

Table-10 : Performance Comparisons among Different Mutants reduction Techniques				
		Mutants after reduction	**% reduction (Out of 73 Mutants)**	
1	Sampling Method	7	90.41%	Best
2	Clustering Method	26	64.38%	Expensive
3	Selection method	13	82.19%	Moderate

5.1 Calculations of Mean Performance of Mutants reduction techniques: `

Test-I Results

Table-11: Performance Comparisons among Different Mutants reduction Techniques				
		Mutants remains after reduction	% reduction	
1	Sampling Method	9	90.42%	Best
2	Clustering Method	24	74.46%	Expensive
3	Selection method	18	80.85%	Moderate

Test-II results

Table-12 : Performance Comparisons among Different Mutants reduction Techniques				
		Mutants after reduction	**% reduction (Out of 73 Mutants)**	
1	Sampling Method	7	90.41%	Best
2	Clustering Method	26	64.38%	Expensive
3	Selection method	13	82.19%	Moderate

Calculations of Combined result Sample program-I (Palindrome) and Sample program-II (Prime number)

Table-13 Comparisons of performance among Mutants Reduction Techniques Test-I & Test-II Combined														
		Total no. of LOC	**Mutants reduced**	**Mutants Left**	**Mutants Reduction %**	**Mutants reduced**	**Mutants Left**	**reduction %**	**Mutants reduced**	**Mutants Left**	**reduction %**		
			Sampling Method			**Clustering Method**			**Selection Method**				
Test -I	**Palindrome program in**	94	85	9	90.42%	70	24	74.46%	76	18	80.85%		
Test-II	**Prime Number**	73	66	7	90.41%	47	26	64.38%	60	**13**	82.19%		
Sum		167	151	16	180.83	117	50	138.84	136	31	163.04		
Test-I	**Mean T1&T**	**83.5**	**75.5**	**8**	**90.42**	**56.5**	**25**	**69.42**	**68**	**15.5**	**81.52**		

Table-14 : Result Comparisons			
% reduction of Mutants	**Sampling Method**	**Cluster Method**	**Selection Method**
% Reduction	**90.42%**	**69.42%**	**81.52%**
Rating	**Best**	**Expensive**	**Moderate**

Table-15 : Relative Comparisons			
Comparisons Between Sampling and Cluster Methods			
% reduction of Mutants	**Sampling Method**	**Cluster Method**	**Difference**
Comparisons in %	**90.42%**	**69.42%**	**21%**
Comparisons Between Sampling and Selection Methods			
	Sampling Method	**Selection Method**	
Comparisons in %	**90.42%**	**81.52%**	**8.9%**
Comparisons Between Selection and Cluster Methods			
	Selection Method	**Clustering Method**	
	81.52 %	**69.42%**	**12.1%**
Rating	**Best**	**Expensive**	**Moderate**

The above analysis clearly indicates that:

A) Sampling method is 21% more efficient than Cluster method.

B) Sampling method is 8.9% more efficient than Selection method

C) Selection method is 12.1% more efficient than Cluster method.

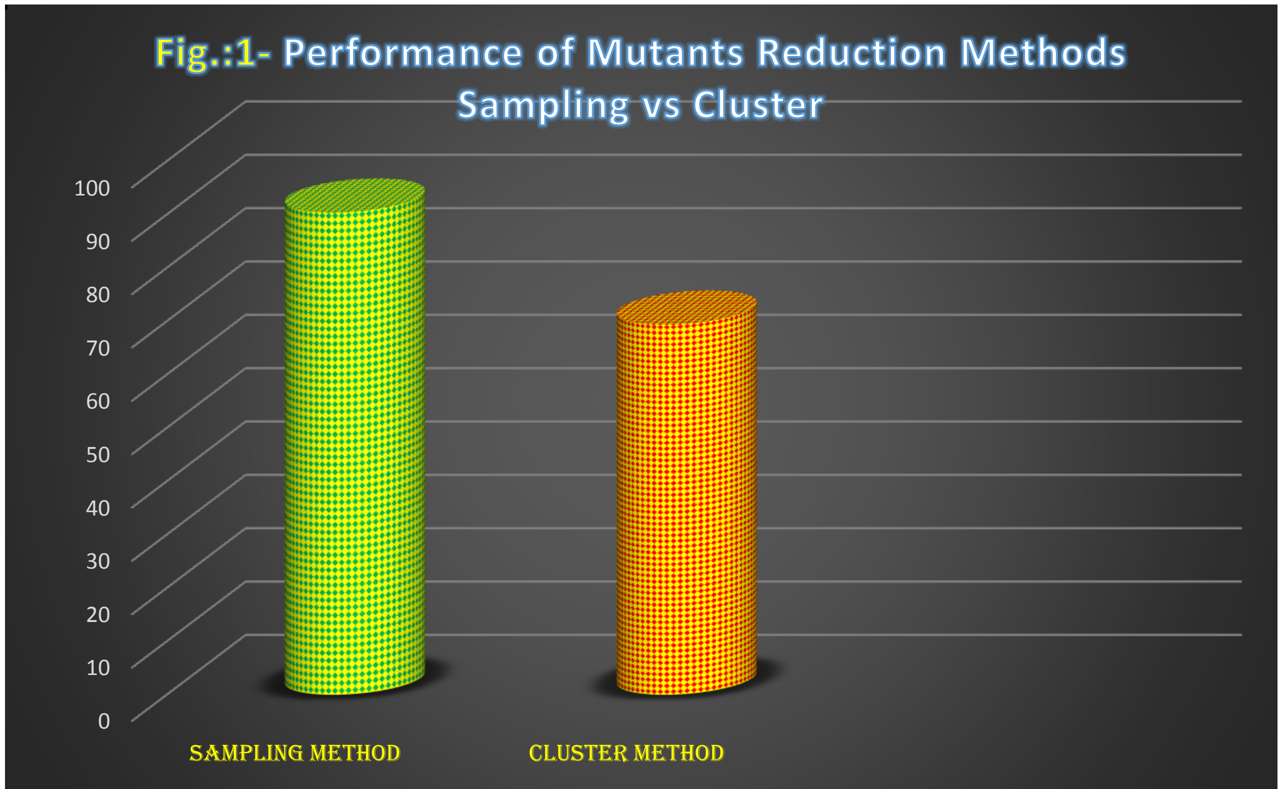


Fig.1 Performance of mutants reduction methods Sampling vs Cluster

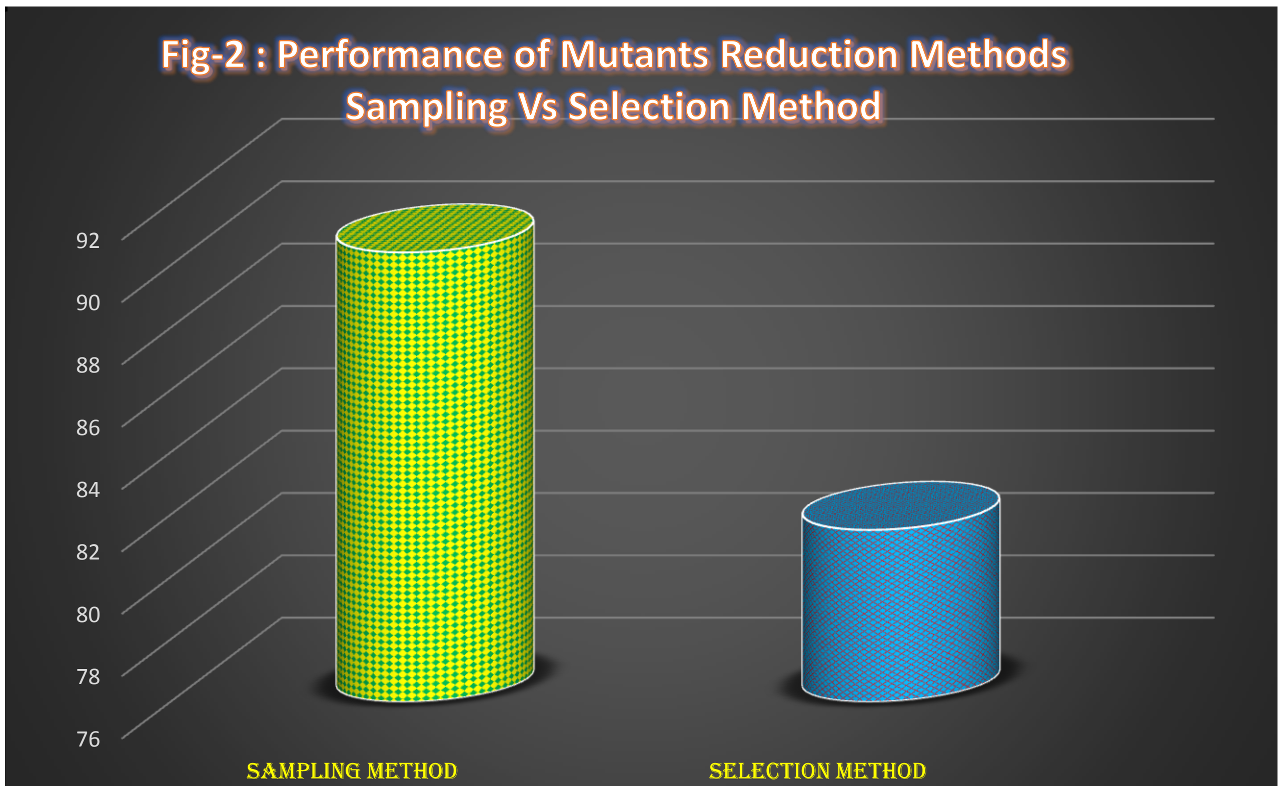


Fig-2 Performance of Mutants Reduction Methods Sampling vs selection method.

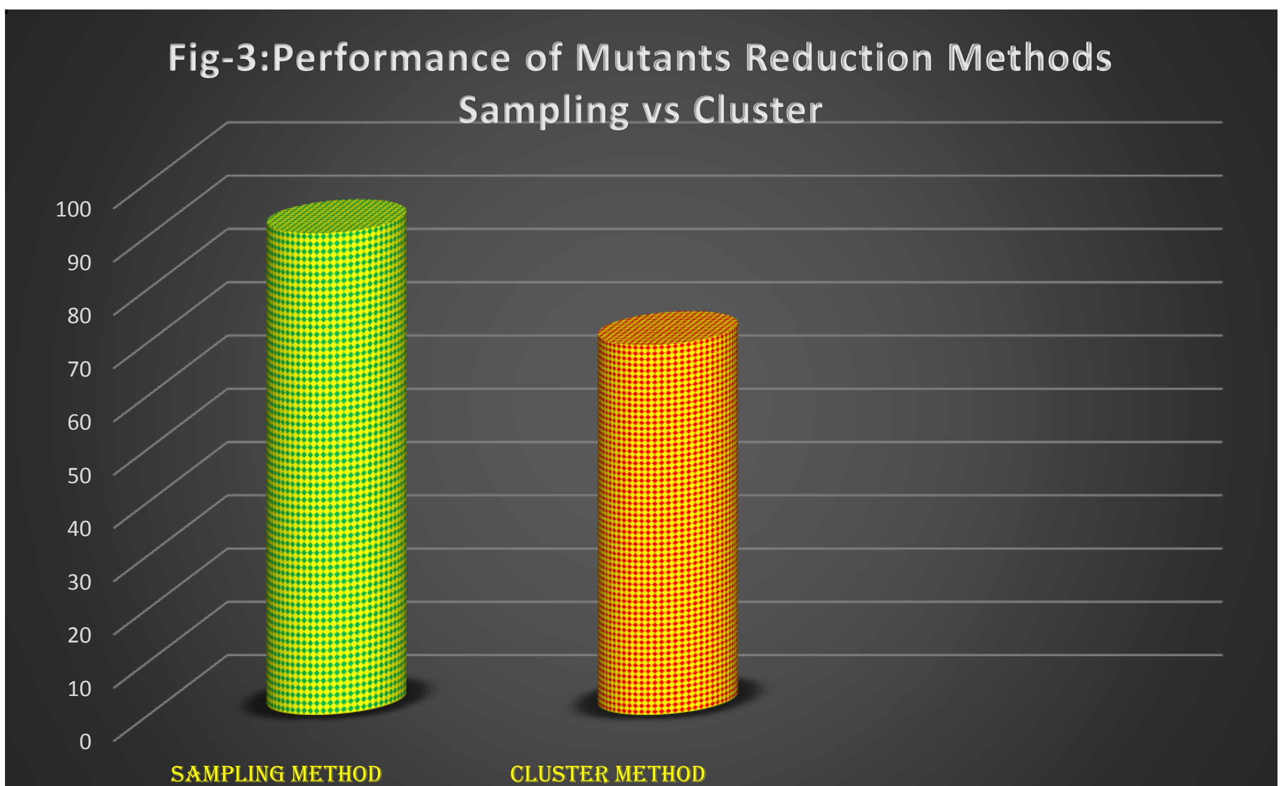


Fig-3 Performance of mutants reduction methods sampling vs cluster.

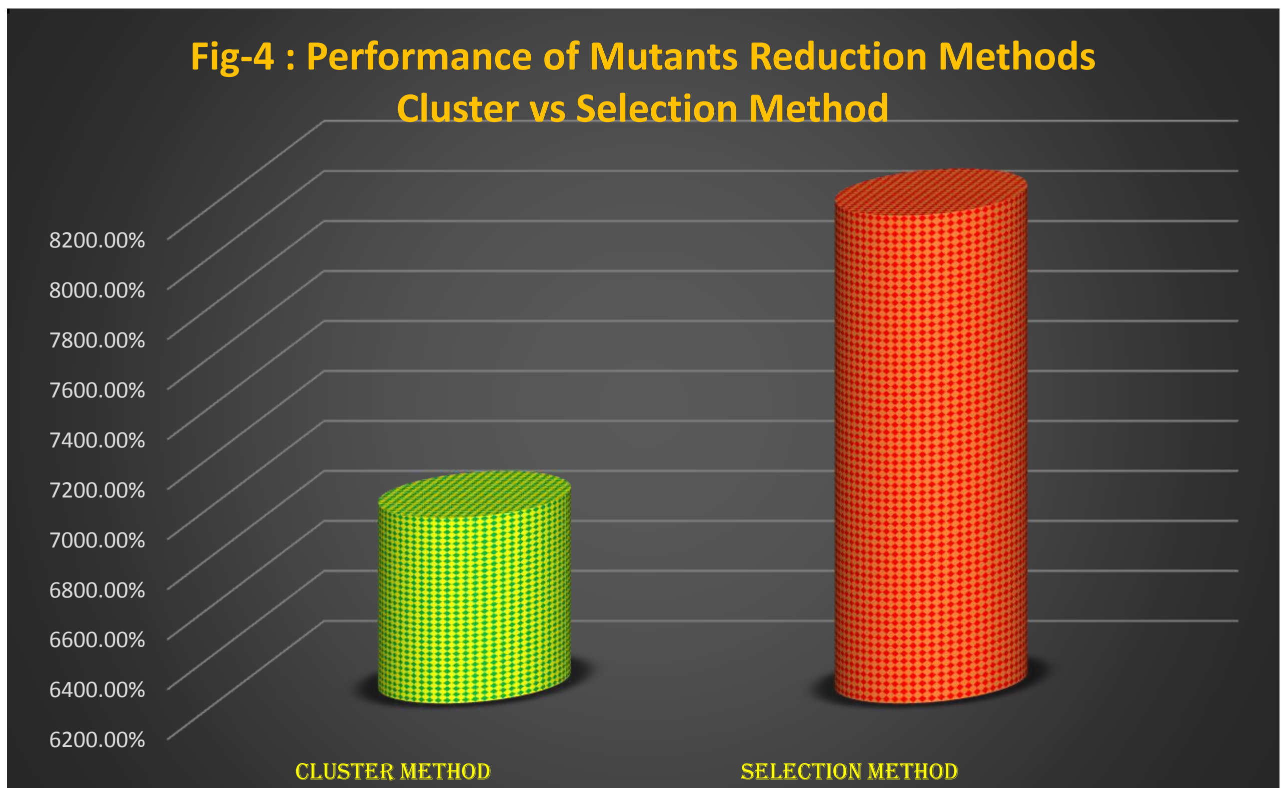


Fig-4 Performance of mutants reduction methods vs selection method

The above discussion clearly indicated that the sampling technique is the best suited and least expensive for mutants' reduction process. Sampling method succeed up to 90.42% of mutants reduction which is highest among all three. Whereas the clustering method is the most expensive. The Selection method scored a moderate value between the sampling and selection methods i.e. 69.42%

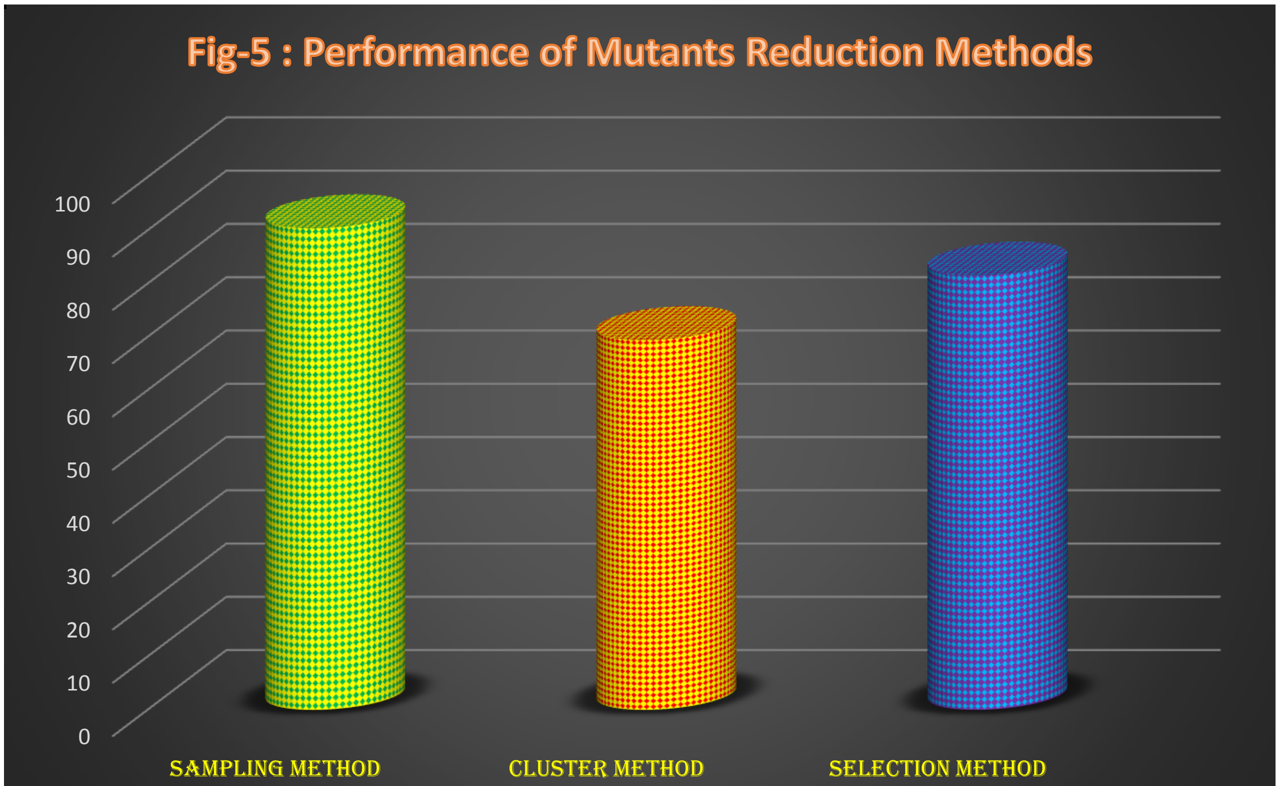
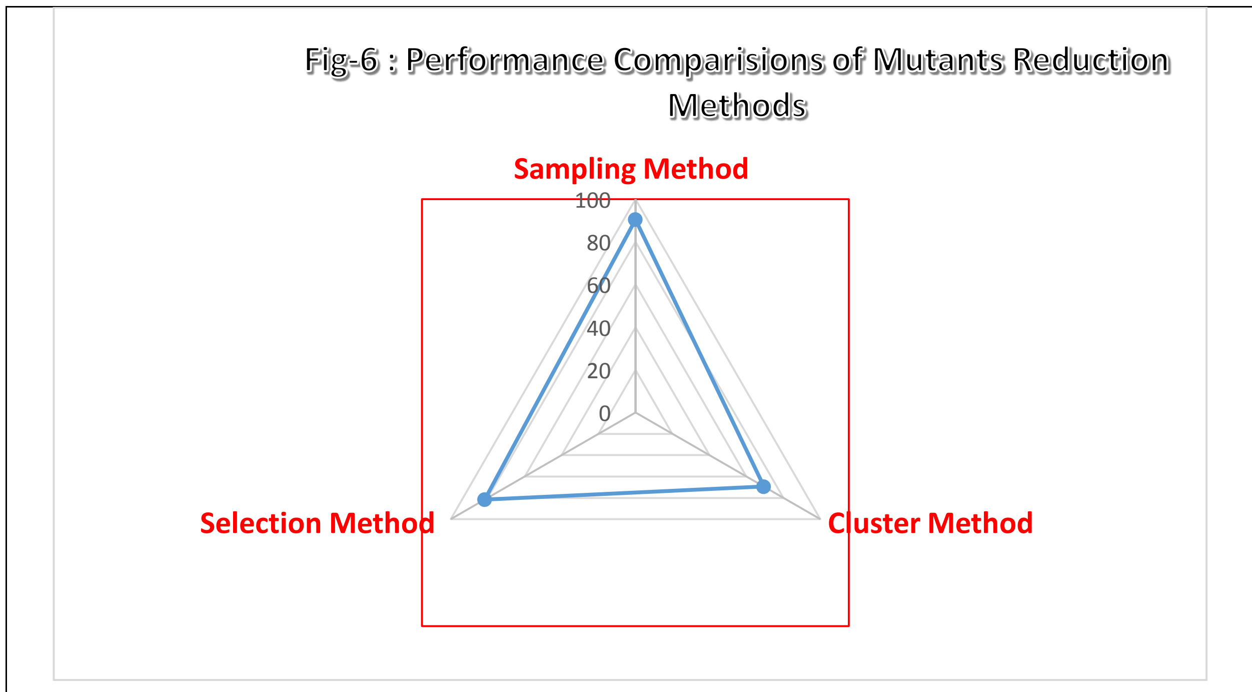


Fig-5 Performance of mutants reduction methods



% reduction of Mutants	Sampling Method	Cluster Method	Selection Method
% Reduction	90.42%	69.42%	81.52%
Rating	Best	Expensive	Moderate

After comparative study of all the four available mutants reduction techniques the Sampling techniques outperforms. However it is not the best technique for each and every case. Sampling Mutants selection is best when the lines of codes are very large. Whereas in small size of programs the selective method is more effective.

Chapter-6
Conclusion and Future Scope

Conclusion

Generating huge number of mutants program as possible combinations of various operators, functions, and statements is a major challenge in the way to success of mutation testing procedures. Programmers and software developers' tries to escape mutation testing cumbersome and expensive procedure due to this reason. If we succeed to decrease the amount of mutants program without affecting the value of mutation testing results then it may prove great technique for software testing as well as popularise among programmers.

Researchers tries to develop such techniques by which we can reduce the number of mutants' programs significant level without affecting the quality of the mutation testing. Among mutants reduction techniques; Sampling method, Clustering Method, Selection Method and Higher Order Mutants are very popular. Ratings these 4 available techniques on the basis of their mutants reduction capabilities is still a research problem.

In this research we successfully demonstrate and compared the above 4 mutants' reduction techniques. We find the sampling technique outperformer amongst all that could decrease the amount of mutants up to 90.42% while the cluster method could reduce only 69.42% mutants. The Selection method scored 81.52 % reduction score which is greater than cluster method and lower than sampling method. We also compare the relative performances amongst the mutants' reduction methods. We conclude that:

A) Sampling method is 21% more efficient than Cluster method and 8.9% more efficient than Selection method and ;

B) Selection method is 12.1% more efficient than Clustering method. Thus the cluster method is the most expensive and sampling method is most economic amongst the all 3 tested mutants' reduction methods.

Future Scope

The present work explores the best available mutants' reduction technique amongst the available techniques. Now each techniques have several methods to implement. For example in the sampling technique there are several sampling techniques are available:

a) X% selection

b) Sampling over program elements and variants

c) Line Per element and variants

d) One per element criteria

e) X% selection per operator

This research can be extended for analysis of best suited sampling methods amongst above listed methods for best sampling results.

Similarly the selective method of mutant reduction has some methods for selection like:

a)Constrained Selection
b)E-Selective
c)Java lance
d)Variable Reduction
e)N-Selection
f)Statement Deletion

We can study separately the best suited selection method for mutant reduction amongst available techniques.

References

[1] A. T. Acree, Jr. On Mutation. PhD suggestion, Atlanta, GA, USA, 1980. AAI8107280.

[2] P. Ammann and J. Offutt. Introduction to programming testing. Cambridge University Press, 2008.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation a fitting gadget for testing examinations? In Software Engineering, 2005. ICSE 2005. Systems. 27th International Conference on, pages 402-411. IEEE, 2005.

[4] Apache Software Foundation. Apache master endeavor. http://maven.apache.org.

[5] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Around the determination of sufficient mutant heads for c. Programming Testing, Verification and Reliability, 11(2):113-136, 2001.

[6] T. A. Budd. Mutation Analysis of Program Test Data. PhD recommendation, New Haven, CT, USA, 1980. AAI8025191.

[7] H. Coles. Pit Mutation testing. http://pittest.org/.

[8] H. Coles. Pit Mutation testing: Matadors. http://pitest.org/quickstart/mutators.

[9] R. A. De Millo, D. S. Guindi W. McCracken, A. Offutt, and K. Ruler. A widened chart of the Mutation programming testing environment. In Software Testing, Verification, and Analysis, 1988. Procedures of the Second Workshop on, pages 142-151. IEEE, 1988.

[10] L. Deng, J. Offutt, and N. Li. Precise appraisal of the declaration crossing out Mutation chairman. In IEEE 6th International Conference on Software Testing, Verification and Validation., Luxembourg, 2013.

[11] GitHub Inc. Programming storage facility. http://www.github.com.

[12] R. Gopinath. Replication data for: A relationship of Mutation strategies. http://dx.doi.org/10.7910/DVN/24936.

[13] R. Gopinath, C. Jensen, and A. Groce. Code scope for suite appraisal by architects. In 36th International Conference on Software Engineering, 2014.

[14] Y. Jia and M. Harman. An examination and survey of the Mutation of Mutation testing. Programming Engineering, IEEE Transactions on, 37(5):649-678, 2011.

[15] R. J. Lipton. Insufficiency finding of PC tasks. Particular report, Carnegie Mellon Univ., 1971.

[16] A. Mathur. Execution, sufficiency, and relentless quality issues in programming testing. In Computer Software and Applications Conference, 1991. COMPSAC '91.,

Proceedings of the Fifteenth Annual International, pages 604-605, 1991.

[17] E. S. Mresa and L. Bottaci. Adequacy of Mutation chairmen and specific Mutation procedures: A precise study. Programming Testing Verification and Reliability, 9(4):205-232, 1999.

[18] A. S. Namin and J. H. Andrews. Finding sufficient Mutation managers by method for variable diminishment. In Techniques of the second Workshop on Mutation Analysis (MUTATION'06), page 5, 2006.

[19] A. Offutt, G. Rother Mel, and C. Zapf. An exploratory appraisal of specific Mutation. In Software Engineering, 1993. Methods. Fifteenth International Conference on, pages 100-107, 1993.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C.Zapf. An exploratory determination of sufficient mutant managers. ACM Trans. Delicate. Eng. Method. 5(2):99-118, Apr. 1996.

[22] A. J. Offutt, G. Rothermel, and C. Zapf. An exploratory appraisal of specific Mutation. In Proceedings of the fifteenth overall meeting on Software Engineering, pages 100{107. IEEE Computer Society Press, 1993.

[23] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In Mutation testing for the new century, pages 34-44. Springer, 2001.

[24]C. Pacheco and M. D. Ernst. Randoop arbitrary test time. http://code.google.com/p/randoop.

[25] D. Schuler and A. Zeller. Javalanche: Efficient Mutation testing for java. In ESEC/FSE '09: Proceedings of the seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 297-298, Aug. 2009.

[26] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Satisfactory Mutation chairmen for measuring test adequacy. In Proceedings of the 30th worldwide meeting on Software outlining, pages 351-360. ACM, 2008.

[27] R. H. Untch. On decreased neighbourhood Mutation examination using a singular mutagenic director. In Proceedings of the 47th Annual Southeast Regional Conference, ACM-SE 47, pages 71:1-71:4, New York, NY, USA, 2009. ACM.

[28] W. Wong and A. P.Mathur. Decreasing the cost of Mutation testing: A test study. Journal of Systems and Software, 31(3):185 - 196, 1995.

[29] W. E. Wong. On Mutation and data stream. PhD hypothesis, Cite soothsayer, 1993.

[30] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Head based and subjective mutant determination: Better together. In IEEE/ACM International Conference on Automated Software Engineering. ACM, 2013.
[31] L. Zhang, S.- S. Hou, J.- J. Hu, T. Xie, and H. Mei. Is head based mutant decision superior to anything unpredictable mutant determination? In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 435-444, New York, NY, USA, 2010. ACM.
[32] L. Zhang, S.- S. Hou, J.- J. Hu, T. Xie, and H. Mei. Is overseer based mutant decision superior to anything sporadic mutant determination? In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 435-444. ACM, 2010.

Appendix

Appendix-I

//C++ Program for testing Palindrome of a number//

```
#include<iostream.h>//
#include<conio.h>//
void main()//
{
clrscr();
inta,b,c,d;//
c=0;//
cout<<"enter any int  :";//
cin>>a;//
d=a;//
while (a>0)//
{
b=a%10;//
  c=c*10+b;//
  a=a/10;//
}
if (d==c)//
 {
cout<<"the given number is palandrom  " ; //
}
else {  cout<<" the number is not palandrom  ";//
}
cout<<"the reversed number is  : "<<c;//
getch();
}
```

Appendix-II

//C++ program for test whether the is Prime or Not a Prime Number//

```
#include<iostream.h>//
#include<conio.h>//
#include<math.h>//
void main()//
{
intn,i;
clrscr();
cout<<" enter the number  ";//
cin>>n;
for(i=2;i<n;i++)//
{
if(n%i==0)//
 {
cout<<" the number is not prime ";//
break;//
 }
}
if(i==n)//
cout<<" the  number is prime ";//
getch();//
}
```