Generating Test Cases using Metamorphic Testing and Genetic Algorithm for Integer Bugs Detection

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

Master of Technology

in

Computer Science and Applications

Submitted By

Deepika Arora

(Roll No 651203001)

Under the supervision of:

Ms. Vineeta Bassi

Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA ï 147004

July 2015

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled "Generating Test Cases using Metamorphic Testing and Genetic Algorithm for Integer Bugs Detection", in partial fulfilment of the requirements for the award of degree of Master of Technology in Computer Science and Applications 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 Ms. Vineeta Bassi 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.

Deepika Arora

(Deepika Arora)

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

(Ms.Vineeta Bassi)
Assistant Professor
CSED

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. S. Bhatia)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

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

I would like to express sincerest thanks to my thesis supervisor Ms. Vineeta Bassi, for her inspiration, guidance, stimulating suggestions, immense help and support throughout the period of this research work. She has provided me with all the necessary resources including motivation and research environment without which it would not been possible to complete this work.

I am equally grateful to Dr. Deepak Garg , Head, Computer Science and Engineering Department, 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.S. Bhatia, Dean, Academic Affairs, for making provisions of infrastructure such as library facilities, immensely useful for the learners to equip themselves with the latest knowledge.

I am also thankful to the authors whose work I have consulted and quoted in this work.

I lack words to express my cordial thanks to all my friends for their useful comments and constructive suggestions during all the phases of my life.

Finally, I convey deep sense of gratitude towards my family members for their moral support and encouragement without which it would not have been possible to bring out this thesis.

Deepika Arora

Deepika Arora

(651203001)

ABSTRACT

In Software Testing, the oracle problem is well understood i.e. the test oracle is either not accessible or if it exists, it is troublesome to apply. Testing software using special values donâ guarantee the correctness of the software. Metamorphic Testing is used to alleviate the oracle problem by developing follow up test cases from initial test cases and software is tested using random values. This testing is very much effective in detecting bugs because by using certain properties of the objective function we develop some metamorphic relations and test the software with follow up test cases against these metamorphic relations. Also in case of testing, it is very important that how effectively the test cases are generated which will test the software using all the execution paths. So to increase the number of test cases, the operators of Genetic algorithm has been applied on the output of Metamorphic Testing to develop more profound test cases which will ensure the correctness of the software under consideration to the maximum extent.

Table of Contents Page No.

List of Figures Page No.

List of Tables Page No.

Chapter 1
INTRODUCTION

Software Testing has been one of the significant problems in ensuring the accuracy of software. Ordinarily used methodology utilizes special values for testing however this regularly lacks in assuring the program rightness. Metamorphic Testing can be utilized to uncover faults in project which couldn't be recognized normally by special test qualities.

The main focus of this thesis is to test a software developed using metamorphic relations under Metamorphic Testing and applying Genetic Algorithm to the test data that crossed Metamorphic Testing so that we can produce new breed of test data which is again tested using metamorphic relations so as to ensure the correctness of the program under test by testing it with more test cases.

This chapter will set context for the remainder of the thesis. It introduces the concept of Software testing, various techniques of testing, Metamorphic Testing, Genetic Algorithm and its various encoding schemes.

1.1 Software Testing

Software Testing is a very crucial step towards assuring quality of the software. It is a type of investigation which is conducted on the software developed with the objective of finding software bugs and to provide different stakeholders with information about the quality of the software under consideration. As indicated by ANSI/IEEE 1059 standard, Testing can be characterized as - A procedure of examining a product to identify the contrasts between the actual and expected conditions (i.e. defects/errors/bug) and to assess the elements of the product [1].

A few software testing strategies have been proposed to provide a template for testing and all have the following characteristics:

- To do testing effectively, software team should organise official meetings and discussions. By this, a lot of errors will be identified before actual testing begins.
- Testing starts at the component level and then go towards integration of the whole system.

- A number of testing techniques are available which can be applied in different scenarios.
- The developer of the software and an independent test group will be responsible for the testing.
- Testing and debugging are different activities but debugging can be accommodated in any testing strategy [2].

A Good test has the following characteristics:

- Valid
- Reliable
- Practical
- Comprehensive
- Relevant
- Appropriate in difficulty [3].

1.2 Why Testing is important?

Below are some of the reasons that will explain the importance of testing:-

- To figure out the deformities those were made amid the developmental stages.
- Customers attain a certain level of satisfaction after seeing the results of testing.
- To ensure quality of the product to be delivered.
- If testing is done properly then the cost of maintenance is also reduced which leads to more reliable results.
- It is obligatory to ensure that software must not collapse as it then becomes very expensive to rebuild the software in later stages.
- It's obliged to stay in the business.
- To ensure that product works as per expectations of the end users.
- Detecting the errors in early phases of software development leads to reduction in cost factor.

1.3 Strategies of Software Testing[2]

The overall process of software development is outlined in the shape of a spiral as shown in Figure1.1. There are four basic levels of testing: Unit testing, Integration testing, Validation testing and System testing.

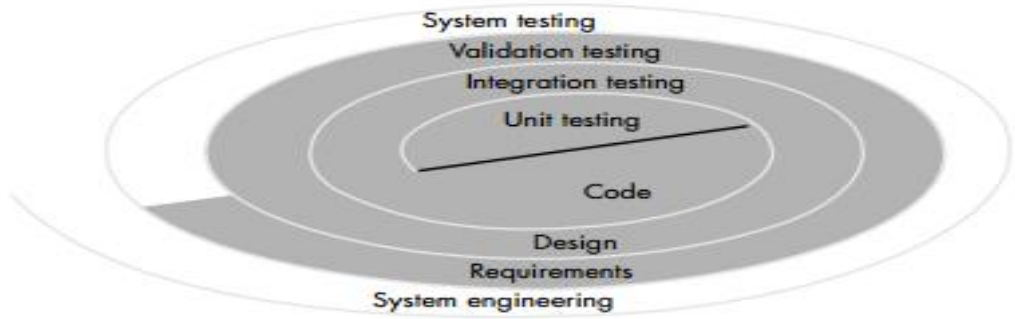


Figure1.1 Testing strategy [2].

1.3.1 Unit Testing

Unit testing also called component testing is meant for testing the individual functionality of the smallest module generated. This level of testing is adopted by the developers while writing code (white-box style), to ensure that the functionality of the module is as per expectations. A module may have to go through many tests, to identity corner cases or run different branches in the code.

Unit testing is alone not sufficient to test a software rather it just ensures that working of individual module is all right and independent of other modules.

A Good unit test is a test which:

- Is ready to be completely computerized.
- Has full control over every piece running.
- Keeps running in memory.
- Reliably gives back the same result when input is same.
- Runs quickly.
- Covers all the paths of the code you want to test.

In Unit testing, the interface of module is tested to ensure that data flows in and out of the system under test as per process and legitimately. The local data structure is checked to ensure that various information that is temporarily stored in these data structure holds the integrity during all the executions of the algorithm. Various boundary conditions are tested to ensure that the behaviour of module is normal at

boundaries also which are meant to cut-off or confine processing. All the control structure are tried to ensure that all instructions in a module have been executed one time. Lastly, all exception handling paths are tested.

1.3.2 Integration Testing

Integration testing is an orderly strategy for developing the system structure while in the meantime leading tests to reveal errors connected with interfacing. The goal is to take unit tested components and manufacture a project structure that has been directed by configuration.

Integration testing is a sort of software testing that tries to confirm the interfaces between components against a software design. Programming parts may be combined in an iterative way or all together ("big bang"). Typically the former is viewed as a superior practice since it permits interface issues to be found all the more rapidly and settled. Integration testing attempts to uncover faults in the interfaces and communication between coordinated parts (modules). Bigger groups of tested programming parts relating to components of the architectural design are continuously incorporated and tried until the software functions as a framework/system.

In other words, when all the individual units are made and tested, we begin joining those "Unit Tested" modules and begin doing the integration testing. So the significance of Integration testing is straight forward- integrate/join the unit tried module one by one and test the conduct as a consolidated unit.

1.3.3 Validation Testing

Validation testing commences after integration testing finishes, when individual segments or modules have been tried and the product is totally packed as a package. Validation succeeds when the product functions in a way that can be sensibly expected by the client. Validation addresses the question: "Are you constructing the right thing?"

Software validation is accomplished with a series of black box tests that confirms to expectations. A test plan traces the classes of tests to be conducted and a test procedure characterizes particular experiments that will be utilized to exhibit conformity with prerequisites. Both the plan and procedure intends to ensure that each behavioural prerequisite is fulfilled, each functional requirement is accomplished, all execution necessities are achieved, documentation is right, and human built and all

other different prerequisites are targeted (e.g., transportability, compatibility, error recovery, maintainability). After every validation experiment has been conducted, one of two scenarios will exist: (1) The function or performance characteristics are as per expectations or (2) a deviation from specification is revealed and a deficiency list is made. Deviation or error found at this stage in a project can seldom be revised before dedicated timeline. It then becomes very important to discuss with the client to set up a technique for resolving defects that are uncovered in this level of testing.

1.3.4 System Testing

System testing consists of multiple test series whose basic aim is to fully check the functionality of computer-based system. Although each test has a different purpose, all work in a direction to verify that system elements have been properly integrated and perform expected functions. A few tests that are worthwhile for software-based systems are:

- Recovery Testing

  It forces the system to fail in a variety of ways and then verifies that recovery is properly performed.
- Security Testing

  This verifies that protection mechanism built into a system will protect it from unnecessary penetration.
- Stress Testing

  Resources are utilized in abnormal quantity, frequency or volume to check the response of the software under stress.
- Performance Testing

  It tests the response time of software and its capability to handle multiple users at a time within the context of an integrated system.

1.4 Test Case

A test case is a document, which has a set of test data, preconditions, expected results and post conditions, to verify that the software developed is as per expectations of the customer.

Test case acts as the starting point for the test execution, and after applying a set of input values; the application has a definitive outcome and leaves the system at some

end point, also known as execution post condition. To maximum extent we should write test cases in a way that only one thing is tested at a time using a test case. Attempt should be made to make test cases atomic, accurate, economical, traceable, repeatable and reusable [2].

Language of writing a test case must

- Be simple and clear that can be easily understood.
- Not be in passive voice.
- Take care of consistency in usage of names.

1.5 Metamorphic Testing

Metamorphic Testing is a software testing strategy that attempts to alleviate the test oracle problem. A test oracle is the method by which an analyzer can figure out whether the working of software is as per expectations or not. A test oracle issue happens when wither the test oracle does not exists or if exists, it is hard to create expected yields from selected test cases, or it is hard to figure out if the actual yields matches with the normal yields.

Metamorphic Testing takes as input an original test case with unknown expected yield and change it into a ñfollow-upò test case. For example, the normal yield of a function that calculates sin x to 100 significant figures is obscure. A Metamorphic relation (MR) is a vital property of the function under consideration like here its sine capacity, for example, ñsin(x) = sin(x + 2´)ò be a metamorphic relation for the sine function. Henceforth, the system under test should have the same expected yield for initial input ñ0.1563ò and a follow up input ñ0.1563 +2´ ò. If the yields from the software/program under test for these two test cases are different, it shows presence of some defect in the implementation of sine function. A MR does not need to be a mathematical statement but rather can be a general property/relation. Thus, Metamorphic Testing is applicable in case of any problem in which a necessary property can be formulated.

1.6 Importance of Metamorphic Testing

In Software Testing, an oracle is something which can ensure the rightness of the software under test .Oracle issue occurs when existence of oracle is not there or if exists; it is too complex / expensive to be utilized in any way. Metamorphic Testing is a methodology which utilizes metamorphic relations, which are properties of the

product to be tested depicted in form of relations among inputs and yields of different executions, to help confirm the rightness of a project. It has been noticed that identification of an adequate number of proper metamorphic relations for testing, even by unpractised analyzers, was conceivable with a little measure of preparation and effort. Moreover, the expense viability of the methodology could be improved through the utilization of more diverse metamorphic relations. The experimental study demonstrates that various metamorphic relations, even those recognized in an ad-hoc manner, had error detection capability to test an oracle, and could accordingly successfully help alleviate the oracle problem.

1.7 Genetic Algorithms

Genetic Algorithms (GAs) are search algorithms which are similar to the process of natural selection and hereditary qualities. As such they represent an intelligent exploitation of a random search used to solve optimization issues. Although randomized, GAs is in no way random rather they use historic data to direct the search into the area of better execution within the search space. The technique of the GAs is intended to replicate/copy the processes occurring in natural systems necessary for development; particularly those follow the principle of Charles Darwin of "survival of the fittest". Since in nature, if we have limited resources available, then there will be a competition among the individuals to attain these resources and this result in the fittest individuals to dominate the weaker ones. Same phenomenon is used in Genetic Algorithm also.

1.7.1 Importance of Genetic Algorithms

Genetic Algorithm is superior to customary Artificial intelligence (AI) in that it is more vigorous. Dissimilar to more seasoned AI Systems, they don't break easily regardless of the possibility that the inputs get changed marginally or in the vicinity of sensible noise. Additionally, in looking a vast state-space, multi-modular state-space, or n-dimensional one, a Genetic Algorithm serves huge advantages over more typical search of optimization techniques.

1.7.2 Genetic Algorithms Overview

GAs replicates the ñsurvival of the fittestò concept among people over consecutive generation for solving a problem. Every generation consists of a population of character strings that resembles the chromosome that we find in our DNA. Every individual represents a point in a search space and a possible solution. The individuals in the population are then made to experience a procedure of evolution.

GAs takes into account a similarity with the hereditary structure and conduct of chromosomes within a population of individuals utilizing the following establishments:

- Availability of resources is very limited. So there is a competition among the individuals in a population to acquire these resources.
- Those individuals who are stronger than others will produce more off springs.
- Qualities from ñbestò individuals transfers from one generation to next generation that when two parents mate, the new one produced will have even more superior qualities than either of the parent.
- Thus each progressive generation will evolve to be more adapted to the surrounding environment [4].

1.7.3 Search Space

A population of individuals is kept inside a search space for a GA, each representing a possible solution to a given problem under consideration. Every individual is encoded in form of a limited length vector of components or variables or in terms of some alphabet, generally the binary alphabet [0, 1]. These individuals can be compared to the chromosomes in a human being. A chromosome (solution) is made out of several genes (variables). The capacity of an individual to compete is determined by assigning a fitness score to every individual. The individual with the optimal (or generally near optimal) fitness score is selected. The Genetic Algorithms utilize particular `breeding' of the solution to produce ñoffspringò even better than the parents by combining data from the chromosomes.

The GA starts from a population consisting of n solutions each represented by a fitness score. Selection of individuals is done to allow them to mate, reproducing offspring. More profound individuals have more chance to reproduce so that the next generation will also have best qualities as their parents.

As parents mate to produce new ones, provision is always there that fresh debuts are introduced. Individual in a population pass away and new one replaces them. So in a long run, a new era gets produced once all the possibilities of mating get depleted. Better individuals will flourish while weaker one becomes extinct.

New eras of arrangements are created containing more great qualities than an ordinary arrangement in a past era. Each successive era will contain more great ñpartial solutionsò than past eras. In the end, once the populace has focalized and is not delivering offspringôs discernibly not the same as those in past eras, the calculation itself is said to have converged to an arrangement of solution for the current issue.

1.7.4 Implementation Details

After an initial population is randomly generated, the algorithm evolves the through three operators:

- Selection operator - Equivalent to survival of the fittest.
- Crossover operator - Exemplify mating between individuals.
- Mutation operation - Allows random modifications.

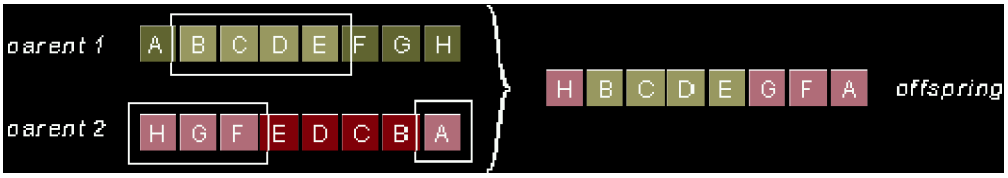
1.7.4.1 Selection Operator

- Offer selection of better individuals, permitting them to transfer their best qualities to next generation.
- The likeness of every individual depends on its fitness score.
- Fitness score may be dictated by an objective function or by a subjective judgement.

1.7.4.2. Crossover Operator

- Two individuals are chosen from the population utilizing the selection operator.
- A hybrid point along the bit strings is arbitrarily chosen.
- The values of the two strings are exchanged at this point.
- If s1=10101010 and s2=01010101 and the hybrid point is 4 then s1'=01011010 and s2'=10100101.

- The two new individuals produced after these mating process are put into the upcoming generation of population.
- By recombining parts of good population, this procedure is prone to result in better offspring.



1.7.4.3. Mutation Operator

- With some low probability, a portion of the new individuals will have some of their bits flipped.
- Its purpose is to maintain diversity within the population and inhibit premature convergence.
- Mutation alone induces a random walk through the search space.

1.7.5 Encoding[5]

Crossover and Mutation are two basic operators of GA. Execution of GA exceptionally rely upon them. Type and implementation of operators relies upon encoding. There are many ways to do encoding for crossover and mutation.

1.7.5.1 Binary Encoding

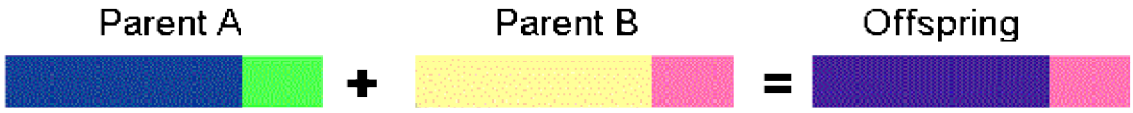
Binary encoding is the most common, as initial works related to GA utilized this type of encoding. In binary encoding every chromosome is represented as a string of binary bits, 0 or 1 as shown below.

Chromosome A	10110010110010101110010 1
Chromosome B	11111110000011000001111 1

Crossover

Single point crossover

- A point where crossover is to be done is selected.
- Binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent.



11011011+11011**111** = **11011111**

Two point crossover

- Two points for crossoverare selected.
- Binary string from starting of chromosome to the first crossover point is copied from one parent, the part from the first to the second point is taken from the second parent and the rest is again taken from the first parent to produce new offspring as shown below.

11010**11** + 11**011**00 = **11011111**

Uniform crossover

Bits are randomly copied from the first or from the second parent.

1**10**011**11** + 1**10****10**101 = **11010111**

Arithmetic crossover

An arithmetic operation is performed on the parents to produce a new offspring.

10110001 + 01011101 =11111101 (OR)

Mutation

Bit inversion - selected bits are inverted

10111001 => **01**111001.

Binary encoding gives many possible chromosomes even with a small number of alleles. On the other hand, this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

1.7.5.2 Permutation Encoding

Permutation encoding is mainly used for problems concerning ordering issues, for example, travelling salesman problem or task ordering problem. In this, each chromosome is represented as a series of numbers.

Crossover

Single point crossover

- One crossover point is chosen
- Till this point the permutation is copied from the first parent, then the second parent is scanned and if the number is not yet in the offspring it is added.

(**1 2 3 4 5** 6 7 8 9) + (4 3 5 **6 8 9 7** 2 1) = (1 2 3 4 5 6 8 9 7)

Mutation

Order changing - two numbers are selected and exchanged

(1 2 3 **4** 5 6 8 **9** 7) => (1 2 3 **9** 5 6 8 **4** 7)

1.8.5.3 Value Encoding

Value encoding is efficient in problems, where some complicated value, such as real numbers, is utilized. Use of binary encoding for this kind of issues would be extremely cumbersome. In value encoding, each chromosome is a string of a few values.

Crossover All crossovers from binary encoding can be utilized here.

Mutation A number is added to the selected values.
(1.34 8.92 **7.86 2.11** 6.89) => (1.34 8.92 **7.73 2.22** 6.89)

1.8.5.4 Tree Encoding

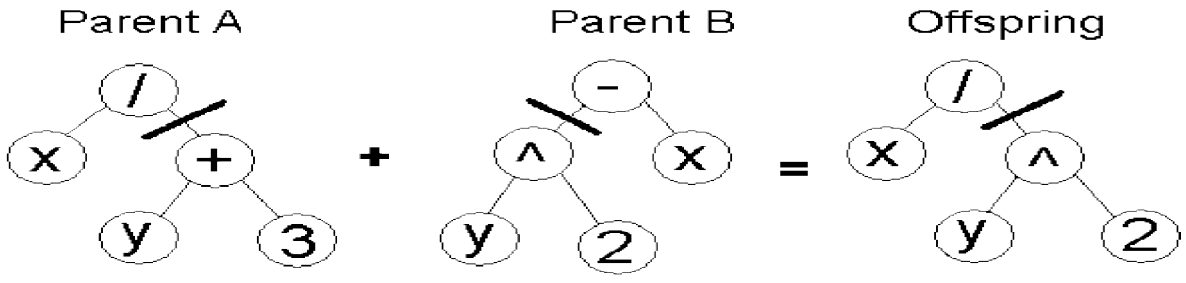
Tree encoding is used mainly for evolving programs or expressions, for genetic programming. In tree encoding every chromosome is represented as a tree of some objects, such as functions or commands in programming language.

Crossover

In both parent one crossover point is selected, parents are divided in that point and exchange part below crossover point to produce new offspring.

Mutation

Changing operator, number - selected nodes are changed.



1.8.6 Effects of Genetic Operators

- Using selection alone will have a tendency to fill the populace with duplicates of the best individual from the populace.

- Using selection and crossover operators will have a tendency to bring about the calculations to meet on a good however sub optimal solution to the problem under consideration.
- Using mutation alone prompts an arbitrary walk through the search space.

1.7.7 Algorithm [4]

Basic steps of Genetic Algorithm involve:

- Initialization of population randomly.
- Calculation of fitness of the population.
- Repetition of steps below.
- Selection of parents from population.
- Crossover is performed on the parents creating more population.
- Mutation is performed.
- Fitness of newly created population is calculated.
- Until the individual created is fittest of all.

1.7.8 Crossover and Mutation probability [6]

There are two fundamental parameters of GA - crossover probability and mutation probability.

Crossover (like selection) is a convergence operation which is planned to draw the populace towards a local minimum/maximum. Crossover fundamentally recreates sexual hereditary recombination (as in human propagation) and there are various ways it is normally executed in GAs. So we indicate crossover probability as a probability of number of couples will be picked for mating (they are normally picked by taking into account selection criteria).Crossover probability shows the frequency of occurrence of crossover. In case of Zero crossover, the newly created individual will be exact copy of parents. If there is a crossover, new ones will contain some parts/qualities of their parents. If crossover is 100%, then offspring will be entirely different from their parents. Crossover is made with the expectation that new chromosomes will have some parts of old chromosomes and perhaps the new chromosomes will be better.

Since the final objective is to make the populace converge, crossover happens more often (regularly every era).

Mutation is a divergence operation. It is proposed to break one or more individuals from a populace out of a nearby least/greatest space and possibly find a superior least/most extreme space.

Mutation probability (or ratio) is fundamentally a measure of the similarity that arbitrary components of your chromosome will be flipped into something else. For instance if your chromosome is encoded as a twofold string of length 100 in the event that you have 1% mutation probability it implies that 1 out of your 100 bits (overall) picked aimlessly will be flipped. Mutation probability says how regularly will be parts of chromosome are changed. If there is no mutation, next generation is taken after crossover (or duplicate) with no change. If mutation transformation is performed, a piece of chromosome is changed. If mutation probability is 100 percent, entire chromosome is changed. Change is made to prevent GA falling into local extreme; however it ought not to happen regularly and influences only a couple of individuals from the population.

There are likewise some other parameters of GA. One likewise vital parameter is populace size. Populace size says what number of chromosomes is in populace (in one era). If there are excessively couple of chromosomes, more crossovers will happen and a huge portion of search space is investigated. GA has less probability to perform crossover and just a little piece of search space is investigated. Then again, if there are an excess of chromosomes, GA. Researches demonstrates that after some breaking point (which depends primarily on encoding and the issue) it is not valuable to expand populace size, on the grounds that it doesn't speed up the solution to a problem.

Chapter 2

Literature Survey

2.1 Oracle

A program is non-testable if either an oracle does not exist or the tester must expend some extraordinary amount of time to determine whether or not the output is correct [7].An Oracle in terms of Software testing is a method by which we can check the validity of the software generated. In different words an oracle is an instrument for figuring out if the project has finished successfully or failed a test.

A complete oracle would have three capacities as follows:

- A generator, to give anticipated outcomes for every test.
- A comparator, to compare anticipated and acquired results.
- An evaluator, to figure out if the examination results are adequately near to be a pass.

One of the key issues with oracle is that they can just address a little subset of the inputs and yields associated with any test. The analyzer may deliberately set the estimations of a few variables, but all the variables in the program have values as well. Arrangement settings, measure of accessible memory, and project alternatives can likewise influence the test outcomes. Therefore, assessment of the test outcomes regarding the test inputs is in light of incomplete data, and may be mistaken.

Any of the oracle capabilities may be mechanized. For instance we may produce forecasts for a test from past test results of a project, from the conduct of a previous versions of a system or any other comparatively similar system, from a standard capacity or from a custom model. We may produce these manually, by an apparatus that nourishes data to the reference program and catches yield or by something that consolidates automatic and manual testing. We may rather produce expectations from specifications, from hit and trial or different sources of data that help a human to assess the data to create the forecast.

2.2 Metamorphic Testing

Metamorphic Testing(MT) solves the oracle problem[8]. It is a technique of testing that utilizes expected properties of the objective function to test projects without intervention of human. These properties are known as Metamorphic relations(MRs). Given a particular problem, we can have more than one MR. It is upto the experts to choose more efficient MR which detect defect more effectively.

Figure 2.1 shows that the more number of metamorphic relations, the more number of mutants will be detected.

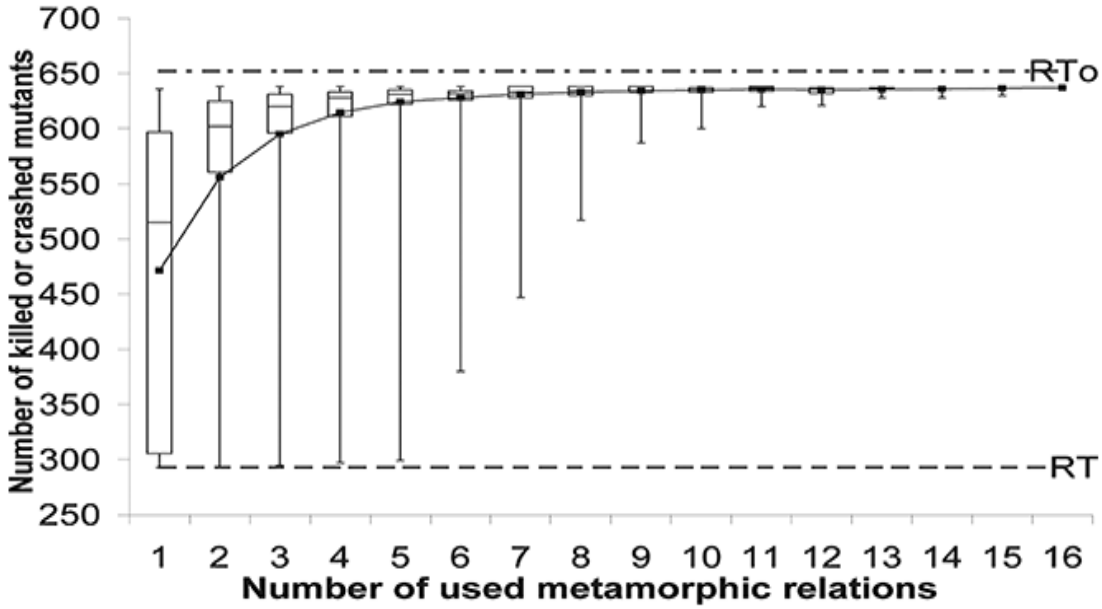


Figure 2.1 Graph between number of metamorphic relations and mutants crashed [9].

2.3 Case Studies

2.3.1 Shortest Path Program [10]

An oracle is a technique using which an analyzer can check whether the outcome of the system is as per expectation or not. A basic issue of software testing is that the oracle is not accessible, if accessible then again excessively complex, making it impossible to apply. Metamorphic Testing (MT) strategy has been proposed to alleviate the oracle problem. MT is an automated testing technique that uses expected properties of the objective function to test projects without human intervention. These properties are called metamorphic relations (MRs). For a given problem, normally more than one MR can be identified. It is consequently fascinating also, extremely helpful for experts to know how to choose effective MRs that is good at identifying program defects.

We are applying Metamorphic Testing on a program ShortestPath that uses Dijkstraôs algorithm to find shortest path between two vertices m and n in a Graph D where D is an undirected graph with positive edge weights. At the point when D is nontrivial, the project is hard to test in light of the fact that no oracle can be essentially applied. All things considered, numerous MRs can be recognized for this issue, with which MT can be performed.

Left Circular Shifts as the MRs :

A property that can be regularly applied in projects of graph theory is the permutation property. Let (D1,m1,n1) be the first input to program ShortestPath. Let (D2,m2,n2) be the second input, where D2 is a graph formed after permutation is applied to D1, vertex m2 in D2 relates to the vertex m1 in D1, and vertex n2 in D2 relates to the vertex n1 in D1. For ShortestPath(D1, m1,n1) and ShortestPath(D2, m2, n2), the value of length of paths found by both must be same.

Let us start with a special kind of permutation, the circular shift. We will consider different shifts of vertices as different MRs. In our analysis, we have utilized 10-vertex graph for experiment. Therefore, we have got 9 MRs appropriate to any experiment, in particular Shift1 , Shift2 , ..., Shift9 , where Shifti refers to the following relation, for i = 1, 2, ..., 9:

ShortestPath(D,m,n).length = ShortestPath(Ű(D), ů(m), ů(n)).length (1)

where Ű(D) depicts the graph produced by circularly moving left the vertices of D i times. If the vertices of D are depicted by (v0, v1, ..., v9), then the same vertices are indicated by (v2, v3,..., v9, v0, v1) in Ű2(D). Vertex ůi(m) in Ű(D) relates to the vertex m in D, and vertex ůi(n) in Ű(D) relates to the vertex n in D, for example, ů2(v1) = v3 in our first sample. "ShortestPath(X).length" will give the path length returned by ShortestPath for input X.

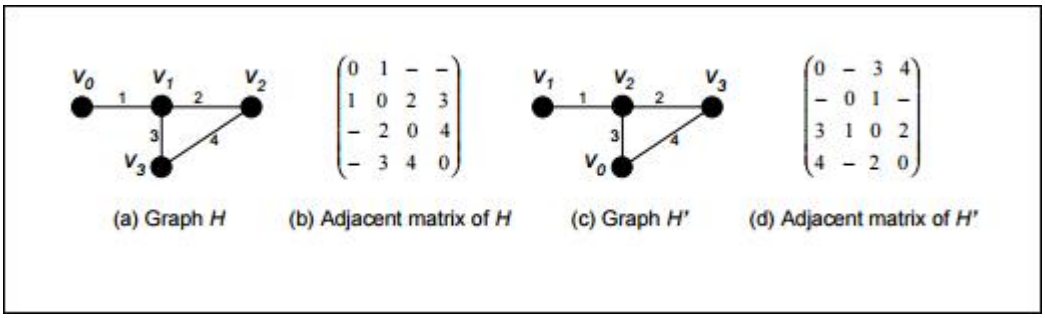


Figure 2.2 Graph and their representations [10].

To estimate the error detecting capability of the MRs, some faults are manually introduced in the source code of program ShortestPath. Each faulty version of the program is called a mutant, and we have introduced 19 mutants. Also the 9 MRs can be categorized into the following 3 classes:

Class1 = [Shift1, Shift3, Shift7, Shift9].
Class2 = [Shift2, Shift4, Shift6, Shift8]. (2)
Class3 = [Shift5].

It is not hard to demonstrate that, for any 10-vertex graph D, the MRs that fit in with the same class is identical to each other. For instance, in Class1, Shift3 can be obtained by applying Shift1 for 3 times, and Shift1 can be achieved by applying Shift3 for 7 times. Moreover, any MR in Class1 implies the various MRs in Class2 and Class3, i.e., Ri ⇒ Rj , where i = 1, 3, 7, 9 and j = 2, 4, 5, 6, 8. Note that Shift5 does not imply some other MR. Subsequently, the 4 MRs in Class1 are the most superior among the 9. We are now going to check the error detecting capacities of all these MRs to see whether the superior MRs have a higher possibility of uncovering a mutant or the scenario is different.

For program ShortestPath(D,m,n), the graph D is represented by an adjacency matrix of size n × n, where n is the number of vertices in graph D. v0, v1... vnī 1 notation is used to indicate the n vertices. If there is an edge (vi , vj) in graph G, where 0 Ò i , j < n, then the (i + 1, j + 1)-th entry of the adjacency matrix contains the weight of this edge; if there is no such an edge, then the (i + 1, j + 1)-th entry of the adjacency matrix will be allotted a special value to designate "no edge". It is additionally assumed that there is edge with weight 0 from a vertex to itself. For chart H demonstrated in subfigure (a) of Figure 2.2, its adjacency matrix is indicated in subfigure (b). In the event that we need to locate the smallest way between vertices v0 and v2 in H, then the information to program ShortestPath (D,m,n) will be D = H, m = 0 and n = 2. Assume (H, 0, 2) is the first experiment. In case of follow up test case in MR, we apply a MR "circularly move left once" to this experiment, then the subsequent experiment will be (HNj1, 3), where HNjis demonstrated in subfigure (c) and (d) of Figure 2.2. The relation is that the path length returned by ShortestPath(H, 0, 2) and the length returned by ShortestPath(HNj1, 3) must be equal.

We initially created an arrangement of introductory test cases = {t_1, t_2... t_{1000}}. To create this test set, we first randomly created 50 graphs as: Each chart has 10 vertices v0, v1... v9. In every chart, every pair of the vertices has a 50% chance of being connected, i.e., the presence of any edge is chosen by tossing of a coin. If two vertices are joined, then the weight of the edge is randomly chosen from numbers 1, 2 ... 50; and a special value is assigned to that position in the adjacency matrix if that edge does not exist. For every graph produced in this manner, we arbitrarily chose 20 unique sets of different nodes as the terminal vertices (take note of that if (m,n) are chosen, then (n,m) won't be chosen). Thus, each graph further created 20 test cases. Accordingly, we have got an arrangement of 20×50 = 1000 test cases T = {t_1, t_2... t_{1000}}.

For each metamorphic relation $Shift_p$, where p= 1, 2, ..., 9, a follow up test case T_i = {$t_{i,1}$, $t_{i,2}$, ..., $t_{i,1000}$} was created taking into account the initial test set T = {t_1, t_2, ..., t_{1000}}, where $t_{i,k}$ in T_i is a follow-up of t_k in T, for k = 1, 2, ..., 1000. For each mutant project $mutant_j$, where j = 1, 2, ..., 19, and for each MR $Shift_p$, where p = 1, 2, ..., 9, $mutant_j$ keeps running on T and T_i, individually. The relation of the yields ($mutant_j$ (t_k), $mutant_j$($t_{i,k}$)) was checked against the MR $Shift_p$, for k = 1, 2, ..., 1000 and p= 1,2, ... , 9. Among the 1000 sets of the yields, we say 640 sets did not fulfil the MR $Shift_p$,

at that point we say the failure rate of $mutant_j$ against $Shift_p$ is 64%.Pseudo code for above process is shown below in Figure 2.3.

```
for i = 1 to 9 do
  for j = 1 to 19 do {
    failureCount = 0;
    for k = 1 to 1000 do {
      if (mutant_j(t_k).length ≠ mutant_j(t_i,k).length), where t_k ∈ T and t_i,k ∈ T_i
        then failureCount = failureCount + 1;
      }
    Print: The failure rate of mutant_j against Shift_i is failureCount / 1000.
    }
```

Figure 2.3 Pseudo code for generation of test cases [10].

Experiment results:

Our test result demonstrates that among the 9 recognized MRs Shift1, Shift2, ..., Shift9 , the hypothetically weakest property Shift5 displayed the most strongest fault detecting capability. For clarity and simplicity of comprehension, we assembled the 9 MRs into 3 classes as per Equation (2). Their normal fault recognition capacities exhibited in the analysis are demonstrated in Figure 2.4.

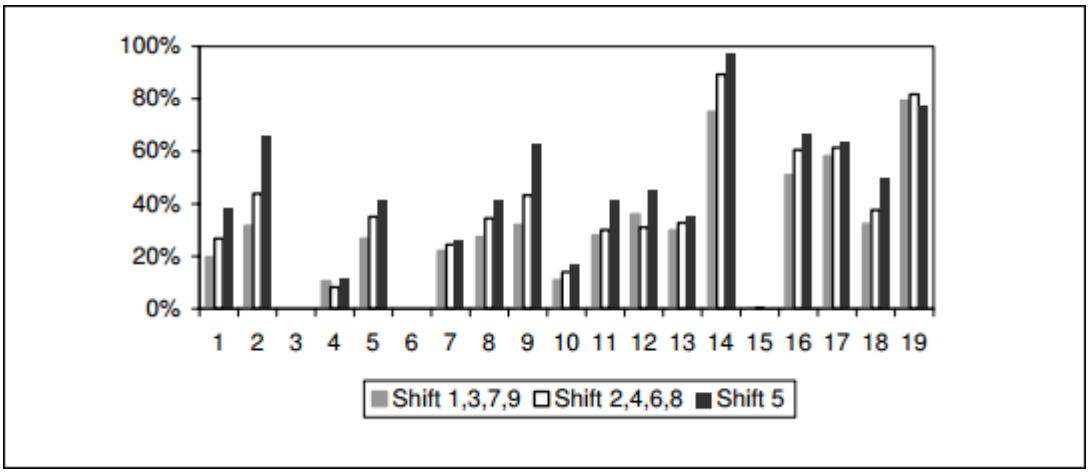


Figure 2.4 An examination of the fault detection capacities of the 3 classes of metamorphic relations on the 19 mutants for the shortest path program [10].

This experiment has shown that theoretically stronger MRs is not necessarily good at detecting program defects. It is suggested, therefore, that selecting MRs from a pure black-box perspective is not adequate.

2.3.2 Matrix multiplication program [11]

In arithmetic, matrix multiplication is an operation that takes two matrices as input and produces another matrix as output. As indicated in Figure 2.5, A is a m × n matrix and B is a n × p matrix. The matrix AB produced as a result is characterized to be the m × p matrix, where every i , j section is given by multiplying the entry A_{ik}(value at i throw and kth column of Matrix A) by the entry B_{kj} (value at kth row and jth column of Matrix B), for k = 1, 2, ..., n, and summing the outcomes over k. This operation is simple if the size of matrices is reasonable. But if the sizes are substantial, it might be hard to check whether the yield of the operation is right or not. So we can say that the program responsible for matrix multiplication suffers from oracle problem when sizes of matrices are large. We make use of Metamorphic Testing in this problem to alleviate the oracle problem.

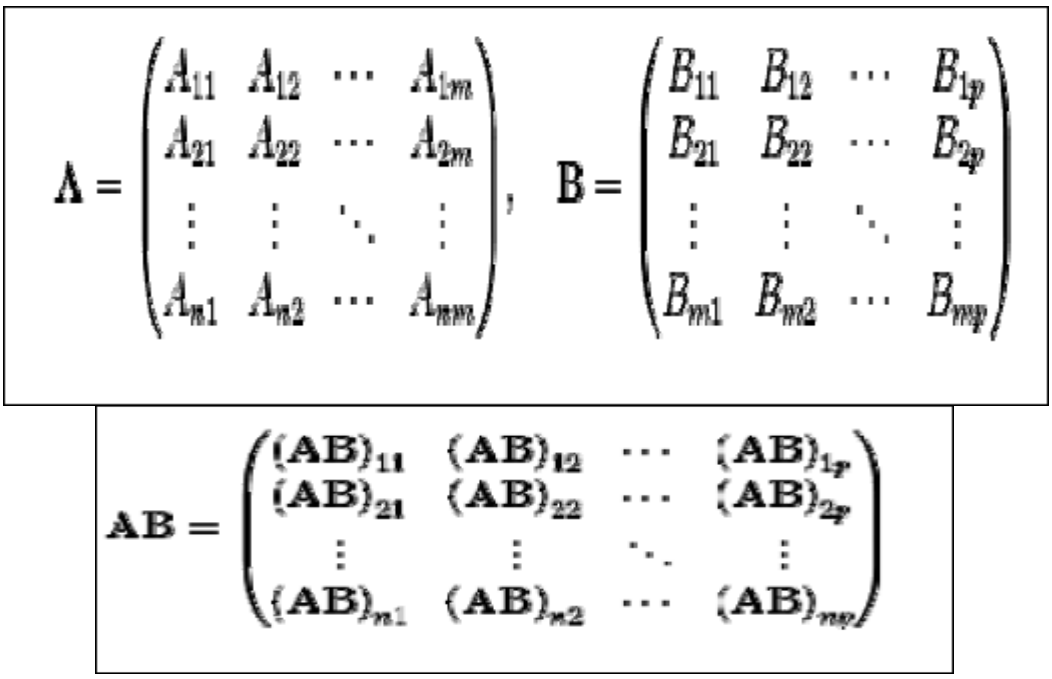


Figure 2.5 The operation of matrix multiplication [11].

On the basis of properties of matrix multiplication, metamorphic relations identified are: (Notation used: A and B are source input matrices. Aô and Bô are follow-up input matrices. Similarly O is output of matrix multiplication of source inputs and Oô is same result with follow up inputs.)

- MR-1: if Aô is equal to B and Bô is equal to A then Oô is not equal to O .This MR corresponds to the fact that matrix multiplication operation is not commutative i.e. A.B is not equal to B.A.
- MR-2: if Aô is equal to A and Bô is equal to B multiplied with I (Identity Matrix) then O is equal to Oô. This MR represents the property of matrix multiplication where A.B is equal to A.(B.I) where I is a identity matrix.
- MR-3: if Aô is equal to A * n where n is a positive integer and Bô is equal to B then O*n is equal to Oô. This MR corresponds to the property of matrix multiplication operation where A. n. B is equal to A.B.n.
- MR-4: if Aô is equal to A * n where n is any negative integer and Bô is equal to B then O*n is equal to Oô. This MR corresponds to the property of matrix multiplication operation where A. n. B is equal to A.B.n where n is a negative integer .
- MR-5: if Aô is a negative of A and Bô is a negative of B then Oô is equal to O. This MR corresponds to the property of matrix multiplication operation where A.B is equal to

  AôBô where Aô is (-1).A and Bô is (-1). B.

To apply the above metamorphic relations we need to create test cases where each test case contains two matrices with random sizes and mutant versions where we have introduced five types of mutants in the source code as shown in the Table 2.1:

Mutant	Correct Version	Mutant Version
Mut-1	for (c = 1 ; c < p ; c++)	for (c = 0 ; c < p ; c++)
Mut-2	if (n < p)	If(n!=p)
Mut-3	for (d = 1 ; d < q ; d++)	for (d = 0 ;d < q ; d++)
Mut-4	if (n < p)	If(n!=p)
Mut-5	Sum=1	Sum=0

Table 2.1 Mutated lines of codes [11].

For each metamorphic relation, another arrangement of test cases (follow up) are produced from the source test cases produced. At that point the source and follow-up test cases are executed against five Mutants, and the outcomes are recorded to see whether faults are recognized or not.

MR/ Mutant	MR-1	MR-2	MR-3	MR-4	MR-5
Mut-1	32350.38	97239.49	97520.02	97742.12	15001.46
Mut-2	32359.93	29999.87	29999.63	29999.44	15001.46
Mut-3	32393.28	67285.10	67565.90	67788.20	67285.10
Mut-4	64753.21	82286.69	82567.49	82789.79	82286.96
Mut-5	32393.28	67285.90	67565.90	67588.20	67285.10

Table 2.2 Average number of pair test cases revealing errors [11].

MR/ Mutant	MR-1	MR-2	MR-3	MR-4	MR-5
Mut-1	32,35%	97,24%	97,52%	97,74%	15,00%
Mut-2	32,36%	30,00%	30,00%	30,00%	15,00%
Mut-3	32,39%	67,29%	67,57%	67,79%	67,29%
Mut-4	64,75%	82,29%	82,57%	82,79%	82,29%
Mut-5	32,39%	67,29%	67,57%	67,59%	67,29%

Table 2.3 Average percentage number of pair test cases revealing errors [11].

Table 2.2 demonstrates that the biggest quantities of test cases with errors for Mutant-1 are ones created by MR-2, MR-3, and MR-4. The biggest quantities of test cases with errors for Mutant-2 are ones created by MR-1, MR-2, MR-3, and MR-4. The biggest quantities of test cases with errors for Mutant-3 are ones created by MR-2, MR-3, MR-4, and MR-5. The biggest quantities of test cases with errors for Mutant-4 are ones created by MR-2, MR-3, MR-4, and MR-5. For Mutant-5, the biggest quantities of test cases with errors are created by MR-2, MR-3, MR-4, and MR-4.
Table 2.3 demonstrates the above observation in percentage form.

2.3.3 Application of MT to Non-numerical problem [12]

Metamorphic Testing is not restricted to numerical projects but can be applied in almost every field. In this segment, a few examples are discussed to illustrate how to utilize MT in non-numerical field.

2.3.3.1 Computer Graphics

When the yields of a project include a lot of information, it becomes costly to check the output whether it is correct or not. For instance, computer graphics programming

produces design and prints them on the screen. It is not possible for a person to verify that whether every last pixel is shown appropriately. In this circumstance, a reasonable methodology is that after checking the rightness of certain measure of individual yields, we apply MT to confirm all the yields in a more efficient and effective manner.

Figure 2.6 shows a chart produced by a realistic-graphics generation programming. For the analyzer, it is not simple to check whether all the pixels in the screen are shown legitimately as its generation involves huge computation and a large amount of pixels are involved. All things considered, some metamorphic relations can be distinguished to check the correctness of output of a computer graphics program.

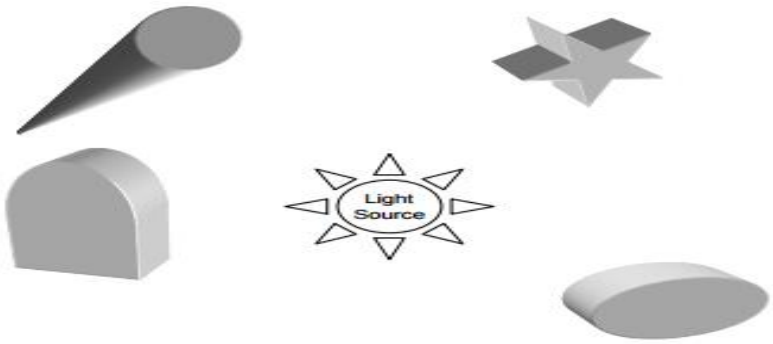


Figure 2.6 Computer Graphics [12].

2.3.3.2 Compilers

Testing compilers is extreme. This is on account of the fact that comparability between the source code and the object code is hard to confirm. We can give a case just to show how to utilize MT to reduce this issue.

Assume pc is a parallelizing compiler. Assume we have the accompanying source code as an experiment:

```
int a, b, c, d;
1 read(a, b);
2 c = a + 1;
3 d = 100;
4 d = d * b;
5 . . .
```

Regardless of the possibility that we don't even know whether the output object code is right, we can identify metamorphic relations to test the compiler. As a straightforward sample, we can find that statement 2 and statement (3, 4) are autonomous of one another. Henceforth, we can build a follow up test case by exchanging position of the statements mentioned above as follows:

```
int a, b, c, d;
1 read(a, b);
2 d = 100;
3 d = d * b;
4 c = a + 1;
```

For the above source code, the parallelizing compiler pc ought to recognize indistinguishable parallelism concerning the first, and this can be confirmed considerably more effectively than the rightness of the object code.

Apart from this, Metamorphic Testing can also be applied to numerical problem like spreadsheets. Test oracle for spreadsheet is very difficult to use, as a spreadsheet contains huge amount of data. Therefore, we use MT for spreadsheet, which is very simple and easy to implement. MT is very effective for end-users because end-user have domain knowledge of spreadsheet that helps to generate good metamorphic relations [13].

Many applications in the field of scientific computing such as computational biology, computational linguistics and other uses Machine Learning algorithms for core functionality that again support solutions in a particular problem domain. These problems are also difficult to test So Metamorphic Testing is used for these applications also [14].

2.3.4 Sine Function [15]

Assume f is a program which provide implementation of sin(ɗ).

This function we are discussing has various understood special test values that can be utilized in testing of f. Input to this program and corresponding output is shown in form of (ɗ, sin (ɗ)). Special values used as a part of the testing are [(0, 0), (´ /6, 1/2), (´ /4, 1/2), (´ /3, 3/2), (´ /2, 1)]. This list of special input values are between 0 and ´ /2. . The program f (Figure 2.7) is a portion of the right program to calculate the sine function.

A mutant (fault) is introduced in the system f to acquire faulty project fm (f with mutant) which is shown below. Faulty system fm is exercised to show that utilizing extraordinary test values are not sufficient for testing the sine function. The defect is a basic variation of a statement in the program and this error is also possible while doing coding which might go undetected.

```
1 double sin(double x, unsigned int qoff)
2 {
3       short a1;
4       a1=_Dtest(&x);
5       switch (a1)
6       {
7       case NAN:
8               errno = EDOM;
9               return (x);
10      case 0:
11              return (qoff ? 1.0 : 0.0);
12      case INF:
13              errno = EDOM;
14              return (_Nan._D);
15      default:
16              {
17              double g;
18              long quad;
19
20              if ( x < -HUGE_RAD|| HUGE_RAD < x) {
21                      g = x/twopi;
22                      _Dint(&g,0);
23                      x -= g * twopi;
24                      }
25              g=x*twobypi;
26              quad = (long)(0 < g ? g + 0.5 : g - 0.5);
27              qoff += (unsigned long)quad & 0x3;
28              g = (double)quad;
29              g = (x - g * c1) - g*c2;
30              if ((g <0.0 ? -g :g) < _Rteps._D) {
31                      if (qoff & 0x1)
32                              g = 1.0;
33                      }
34              else if (qoff & 0x1)
35                      g = _Poly(g*g,c,7);
36              else
37                      g *= _Poly(g*g,s,7);
38              return (qoff & 0x2 ? -g :g);
39              } /* default */
40      } /* case stmnt */
41 } /* sin() */
```

Figure 2.7 Program to implement sine function [15].

In the above code we have changed "quad & 0x3" in line 27 to "quad & 0x1".

Since metamorphic relations are basically the properties which the function f must hold, the properties mentioned below must hold true for fm if fm is working correct.

Rsin1 : $\sin(x) = \sin(x + 2\pi)$

Rsin2 : $\sin(x) = -\sin(x + \pi)$

Rsin3 : $-\sin(-x) = \sin(x)$

Rsin4 : $\sin(x) = \sin(\pi - x)$

Rsin5 : $\sin(x) = -\sin(2\pi - x)$

Rsin6 : $\sin(x) + \sin(y) + \sin(z) - \sin(x+y+z) = 4 * \sin((x+y)/2) * \sin((x+z)/2) * \sin((y+z)/2)$

Rsin7 : $\sin2(x) + \sin2(\pi / 2 - x)=1$

Rsin8 : $\sin(3 * x) = 3 * \sin(x) - 4 \sin3(x)$

Rsin9 : $\sin(x)2 - \sin(y)2 = \sin(x + y) * \sin(x - y)$

Rsin10 : $\sin(5 * x) = 16 * \sin5(x) + 5 * \sin(3 * x) - 10 * \sin(x)$

	= expected result	R_{sin1}	R_{sin2}	R_{sin3}	R_{sin4}	R_{sin5}	R_{sin6}	R_{sin7}	R_{sin8}	R_{sin9}	R_{sin10}
Special Test Values											
0	T	T	T	T	T	T	F	T	T	F	T
π/6	T	T	F	T	F	T	F	T	T	T	F
π/4	T	T	F	F	F	T	F	T	F	T	F
π/3	T	T	F	F	T	F	F	T	F	T	F
π/2	T	T	F	F	T	F	F	T	F	F	F
Random Test Values											
1.71E-05	T	T	F	T	F	T	F	T	T	F	T
1.00E+00	T	T	F	F	T	F	F	T	F	T	F
2.13E-05	T	T	F	T	F	T	F	T	T	F	T
8.24E+00	T	T	F	F	T	F	F	T	T	F	F
1.29E-01	T	T	F	T	F	T	F	T	T	T	T
7.83E-01	T	T	F	T	F	T	F	T	T	T	F

Table 2.4 Results from Metamorphic Testing of sine program fm [15].

Table 2.4 demonstrates the outcome of testing fm utilizing values from 0 to ´/2 and MRs. The 1st column depicts values input in the program. Here both special and random values are used. The 2nd column, fm(x) = expected result, is the outcome of testing fm using special values to confirm against expected results. Every other column $Rsin_i$ is consequence of testing the ith metamorphic relations. "T" in the table above shows that the relation holds True and "F" shows that the relation holds False. Computations of fm(x) utilizing random test values as inputs can't be confirmed as it can be confirmed when utilizing special value testing. From the test results (indicated in Table 2.4), various extremely helpful conclusions can be drawn. When special test values are utilized to test the faulty program fm, all tests yield the normal results means fault is not detected by testing with these special values. However, through Metamorphic Testing, mistakes can be revealed utilizing special value as test data. Metamorphic Testing can be completed utilizing random test values along with special test values. Rsin6 distinguishes fault with every test case. It utilizes the sine function seven times and the repeated execution of program with different parameters increases the probability to uncover defects. Four MRs, named as Rsin6 , Rsin8 , Rsin9 and Rsin10, make use of the sine function more than 2 times and they have reliably uncovered faults in fm. The other six metamorphic relations make utilization of the sine function precisely 2 times, two of them didn't uncover any flaw and the other four have uncovered fault in fm. The suggestion is that a metamorphic relation who calls the function more is more favourable to detect errors .

Then again, for a particular fault, ability of each MR to uncover the fault is dependent on the way fault has been introduced and the test cases. Metamorphic Testing uses black box testing methodology and it is not known before testing is executed that

what kind of errors it may recognize, thus it is valuable to perform Metamorphic Testing with all possible relations and all possible test values. The utilization of random test values over special test values is of great benefit as these don't have any suggested property inside the test data and are hence more inclined to test different paths of program under test. Similarly we have test cases and metamorphic relation for cos function also [16].

2.3.5 Integer Bug detection [17]

The integer bugs assume an essential part in functionality and security of software. For the Oracle issue, whole integer bugs are constantly overlooked as program will otherwise throw exception. In this case study a metamorphic relation is given which is best way to deal with invisible integer bugs without oracle [18]. It is demonstrated that this technique can recognize faults which are hard to be found in conventional approach and enhance the effectiveness of integer bugs detection.

Integer bugs detection as of now contains three techniques. It comprises of prior condition, error detection and post conditions. Among them, the prior condition is utilized to check whether an error occurs before performing operation. For instance a prior state of division by zero is that the divisor is 0. Error detection is needed to figure out if the mistakes happen during the implementation. Since integer errors are created by limit of the machine, the operating system and compiler can give a mechanism to find overflow errors. Post conditions are to do the operations first and after that to get the conclusion by contrasting the actual result and the expected result. Post condition is commonly used techniques to find integer bugs.

Methods of integer bugs detection based on Metamorphic Testing: For the 'Test Oracle' issue in the strategy for the integer bugs identification, system for the integer bugs recognition in light of metamorphic test is proposed, which is basically a identification method based on rightness. If an integer blunder happens, the project must ascertain to get a wrong output and if this wrong output is utilized in the following step, the system will inevitably either fall, or output a wrong or exceptional output. That is, when the inputs of programming meet some certain criteria, the relating yield of programming will likewise meet the comparing criteria.

Formal definitions of the concepts which are required in the method are given below:

Definition 1: Assumption: program P is an implementation of the function p.

x1, x2, é , xn, (n>1) are inputs for function p and p (x1), p (x2), é , p(xn) are corresponding outcomes of function p. If x1, x2... xn satisfy the relation r1 and p (x1), p(x2)... p (xn) satisfy the relation r2, (r1,r2) is said to be a metamorphic relation of program P .

Definition 2: For the same program P, Metamorphic relations (r, rm) which we need to verify or to extract always are not just one. It is demonstrated that Ri = (r_i, rm_i) means the i-th metamorphic relations of the system P and that S (R) = {R1, R2, é } signifies the arrangement of metamorphic relations of program P. System for the integer bugs recognition in light of Metamorphic Testing and metamorphic relations incorporates three stage:

Step 1: Select the source test cases. For project P, I1, I2, é , In are chosen as inputs comparing to x1, x2, ..., xn in system P. That is, source test cases (I1, I2, é , In) are picked up.

Step 2: Select the right metamorphic relation to create follow up test cases. We pick the fitting metamorphic relation R = (r, rm) of project P. It is accepted that the project P is right and afterward r (I1, I2, é , In) =>rm(P (I1), P (I2), é , P (In)) is created from definition 1. It implies that follow-up test cases are derived from the source test cases. In the event if there are many relations, we can pick a number of metamorphic relations, some metamorphic relations R1, R2, ..., Rn can be chosen to make many follow up test cases.

Step 3: Compared results both from source test cases and from follow-up test cases in order to judge whether metamorphic relations is complied. In the event that program P is right, P obeys r (I1, I2, é , In) = >rm(P (I1), P (I2), é , P (In)) in which (P (I1), P (I2), é , P (In) are the relating yield So if the test case which is running does not meet the formula above, the assumption is not and it implies that the program has errors.

Figure 2.8 shows process of Integer bug detection based on Metamorphic Testing.

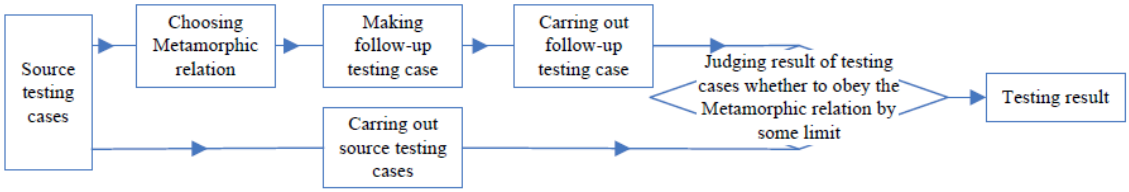


Figure 2.8 Process of integer bugs detection based on Metamorphic Testing [17].

For Mission critical programs, Integer overflow is one of the most dangerous faults. Metamorphic Testing is applied to detect Integer Overflow defect and solve the oracle problem in critical program of Traffic Collision Avoidance System (TCAS) [19].

Chapter 3

Problem Statement

3.1 Problem Definition

It is impossible to test software with all the conceivable inputs. Successful test cases do not reveal error, so these test cases are not considered and discarded by testers. But these successful test cases do carry useful information which remains buried and unused [20].

One more limitation of Software testing is the oracle problem. The test oracle has always been restricting the development of software testing. In some cases the test oracle does not exists or if it exists, it is too complex to be used. Also we usually test the software using special values which do not ensure the correctness of the software. So even after testing a software using special test values we are not satisfied with the testing results.

3.2 Problem Analysis and Resolution

As a solution to the limitations of Software testing stated above, Metamorphic testing is proposed. It solves the oracle problem, takes into account successful test cases and uses these to generate follow up test cases which checks the objective function using its important properties. In other words, the correctness of the program is judged by checking whether it satisfies some necessary properties of the objective function which are called metamorphic relations.

Also to test a program efficiently, there is always a constraint on the availability of test data. To generate a large amount of test data, we make use of Genetic Algorithms with Metamorphic Testing so as to ensure that software is tested with more test cases which guarantee the correctness of the software to the maximum extent.

To solve the problem stated above, we have adopted below steps which will be described in detail in next chapter.

A case study of a program for calculating area on Cartesian coordinate is used for research on method of integer bug detection based on Metamorphic Testing. A mutant

is introduced in the program and data is input in the faulty version of the program to check the outcome. Program with a mutant introduced in it is tested at various values in the range of integer and out of these input values, a data set is extracted for which program computes correctly instead of presence of a mutant.

This data set will become the source test data for the metamorphic relations under Metamorphic Testing. After doing Metamorphic Testing we observed that still some test cases are there for which the faulty version of program computed correct results.

We again reused these successful test cases and applied Genetic Algorithm on these test cases. In Genetic Algorithms, by using crossover and mutation operators a new breed of test data is produced which is again tested using Metamorphic Testing.

We observed that after completion of this complete, the number of successful test cases get reduced by which efficiency of testing gets increased. In other words, now more number of test cases is detected by the program which ensures that testing is conduced efficiently.

Chapter 4

Proposed Solution and Implementation

4.1 Applying Metamorphic Testing

To provide implementation of the problem statement and describe the resolution steps stated in Chapter 3, calculation of graphics area on Cartesian coordinates is used as case studies for research on integer bug detection based on Metamorphic Testing and Genetic Algorithm.

The problem can be described as: vertex coordinates of a triangle are generated using loop with range in the integer's boundary and saved in a text file.

A mutant is introduced in the program. The program reads then text file and calculates area and perimeter using a correct program and a program with mutant. After that comparison of the output from correct program and faulty program is made and that input coordinates which have same area and perimeter in both cases is recorded in a separate text file. Now these are those successful test cases for which even faulty program computed correctly.

Step by Step demonstration is given below:

Step1:

A program to calculate area of a triangle given three vertices is:

```
Double area(double x1,double y1,double x2,double y2,double x3,double y3)
{
double d1,d2,d3,s,area;
 d1= sqrt((x2-x1)^2 + (y2-y1)^2);
 d2= sqrt((x3-x2)^2 + (y3-y2)^2);
 d3= sqrt((x1-x3)^2 + (y1-y3)^2);
s=d1+d2+d3/2;
area=sqrt(s*(s-d1)*(s-d2)*(s-d3));
return area;
}
```

The statement ñs=(d1+d2+d3)/2ò is modified as ñs=(d1+d2+(int)d3)/2ò. This is introduction of a mutant in the program under test.

The program with mutant and the program without mutant are run with the source data (generated using simple loop) to calculate area of a triangle and both the areas are compared. After comparison if both the areas are equal we get our test input (input.txt) as below in Table 4.1 on which we will apply Metamorphic Testing as for these input coordinates, the behaviour of program with mutant and the program without mutant is same i.e. this data set has passed normal testing even after presence of fault and we are not able to detect fault using normal testing for this data set.

	----------V1 ----------		----------V2----------		----------V3----------	
Input	x1	y1	x2	y2	x3	y3
S1	65000.000000	64678.000000	65123.000000	63890.000000	64568.000000	65173.000000
S2	65000.000000	64678.000000	65123.000000	63890.000000	64568.000000	65254.000000
S3	65000.000000	64678.000000	65123.000000	63890.000000	64568.000000	65343.000000
S4	65000.000000	64678.000000	65123.000000	63890.000000	64568.000000	65488.000000
S5	65000.000000	64678.000000	65123.000000	63890.000000	64571.000000	65138.000000
S6	65000.000000	64678.000000	65123.000000	63890.000000	64571.000000	65250.000000
S7	65000.000000	64678.000000	65123.000000	63890.000000	64571.000000	65378.000000
S8	65000.000000	64678.000000	65123.000000	63890.000000	64571.000000	65406.000000
S9	65000.000000	64678.000000	65123.000000	63890.000000	64574.000000	65246.000000
S10	65000.000000	64678.000000	65123.000000	63890.000000	64575.000000	65338.000000
S11	65000.000000	64678.000000	65123.000000	63890.000000	64576.000000	65473.000000
S12	65000.000000	64678.000000	65123.000000	63890.000000	64577.000000	65242.000000
S13	65000.000000	64678.000000	65123.000000	63890.000000	64580.000000	65119.000000
S14	65000.000000	64678.000000	65123.000000	63890.000000	64580.000000	65191.000000
S15	65000.000000	64678.000000	65123.000000	63890.000000	64580.000000	65238.000000
S16	65000.000000	64678.000000	65123.000000	63890.000000	64580.000000	65315.000000

S17	65000.000000 64678.000000 65123.000000 63890.000000 64580.000000 65353.000000
S18	65000.000000 64678.000000 65123.000000 63890.000000 64580.000000 65510.000000
S19	65000.000000 64678.000000 65123.000000 63890.000000 64580.000000 65529.000000
S20	65000.000000 64678.000000 65123.000000 63890.000000 64583.000000 65234.000000

Table 4.1 Source test Input

Note: We have taken 1000 coordinate set as test data for implementation purpose but for report we are showing only 20 records for reference.

Step2:

Metamorphic relation 1 and 2:

By using similar triangles property on Figure 4.1 where

$$\left\{ \begin{array}{l} \frac{V1+V2}{2} = V2\hat{o} \;,\; \frac{V1+V3}{2} = V3\hat{o} \\ \frac{V1+V2\hat{o}}{2} = V2\hat{o}\hat{o} \;,\; \frac{V1+V3\hat{o}}{2} = V3\hat{o}\hat{o} \\ \frac{V1+V2\hat{o}\hat{o}}{2} = V2\hat{o}\hat{o}\hat{o} \;,\; \frac{V1+V3\hat{o}\hat{o}}{2} = V3\hat{o}\hat{o}\hat{o}\hat{o} \end{array} \right\} \Rightarrow (1)$$

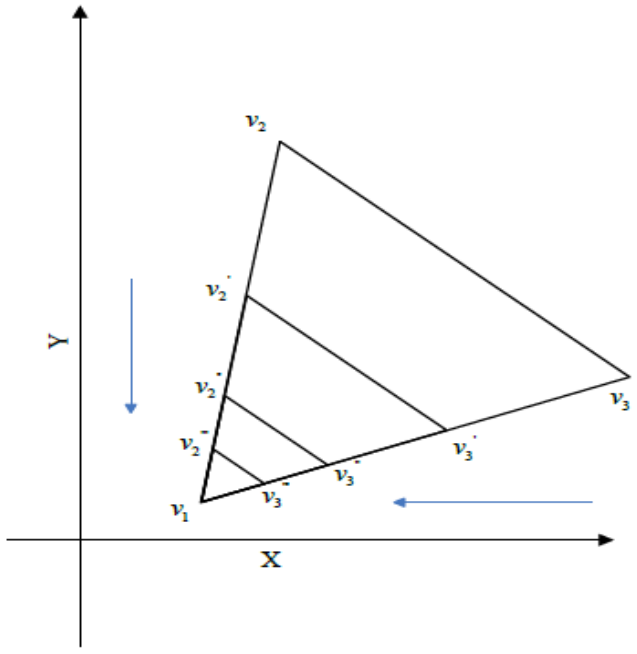


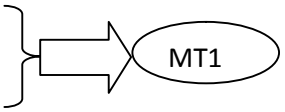
Figure 4.1 Area and perimeter are proportional to side length in similar triangles

We have four triangles in the above figure whose sides are proportional to each other. According to similar triangle property we have:

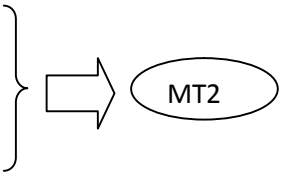
Triangle(v1,v2,v3),triangle(v1,v2',v3'),triangle(v1,v2'',v3'') and triangle(v1,v2''',v3''')

And v1v2=2v1v2'=4v1v2''=8v1v2''', v1v3=2v1v3'=4v1v3''=8v1v3'''

$$\frac{area(v1,v2,v3)}{area(v1,v2',v3')} = \frac{area(v1,v2',v3')}{area(v1,v2'',v3'')} = \frac{area(v1,v2'',v3'')}{area(v1,v2''',v3''')} = 4$$



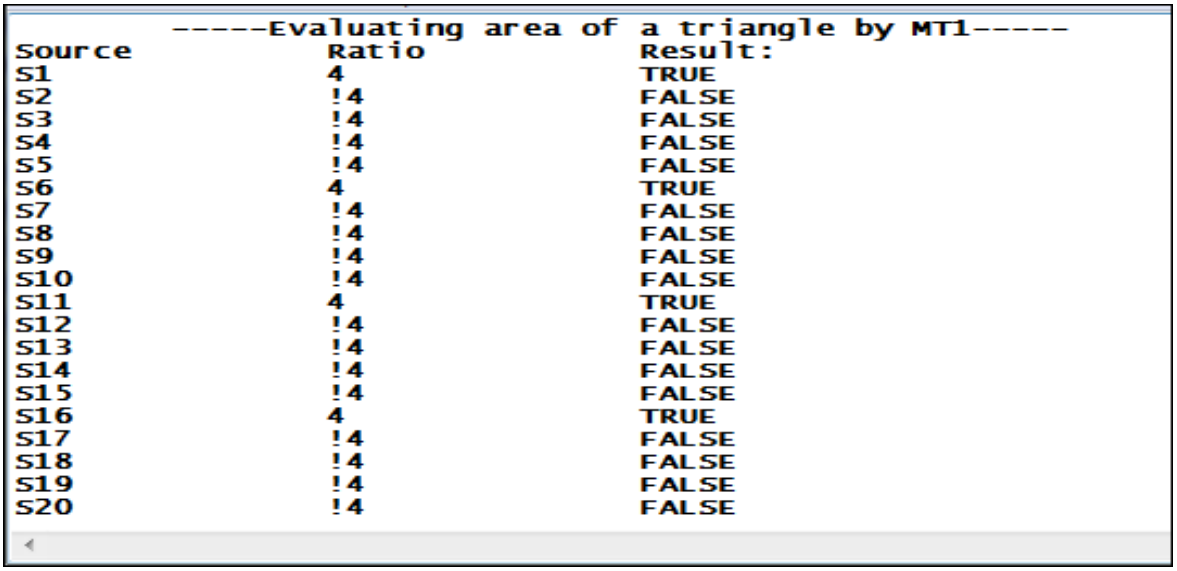
$$\frac{perimeter(v1,v2,v3)}{perimeter(v1,v2',v3')} = \frac{perimeter(v1,v2',v3')}{perimeter(v1,v2'',v3'')} = \frac{perimeter(v1,v2'',v3'')}{perimeter(v1,v2''',v3''')} = 2$$



The test data mentioned in Table 4.1 is input in the below algorithm output of which is shown in Figure 4.2 and Figure 4.3.

Algorithm:-

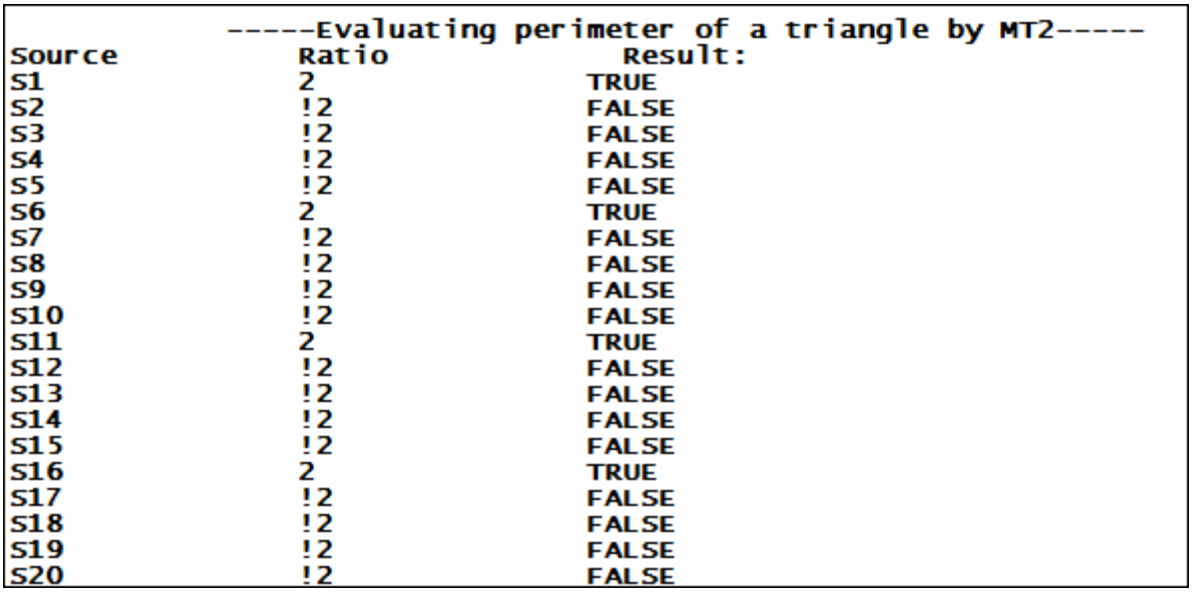
1. Read input.txt as coordinates x1, y1, x2, y2, x3, y3.
2. Break the triangle with coordinates read in step 1 into four triangles as marked in 1.
3. Apply MT1 and MT2 on the four triangles formed.
4. Record the output of comparison of areas in a separate text file (result1.txt ï Figure 4.2 and result2.txtð Figure 4.3)
5. Repeat step 1 to 4 till file end.
6. Exit.



-----Evaluating area of a triangle by MT1-----

Source	Ratio	Result:
S1	4	TRUE
S2	!4	FALSE
S3	!4	FALSE
S4	!4	FALSE
S5	!4	FALSE
S6	4	TRUE
S7	!4	FALSE
S8	!4	FALSE
S9	!4	FALSE
S10	!4	FALSE
S11	4	TRUE
S12	!4	FALSE
S13	!4	FALSE
S14	!4	FALSE
S15	!4	FALSE
S16	4	TRUE
S17	!4	FALSE
S18	!4	FALSE
S19	!4	FALSE
S20	!4	FALSE

Figure 4.2 Result of Metamorphic Relation 1.



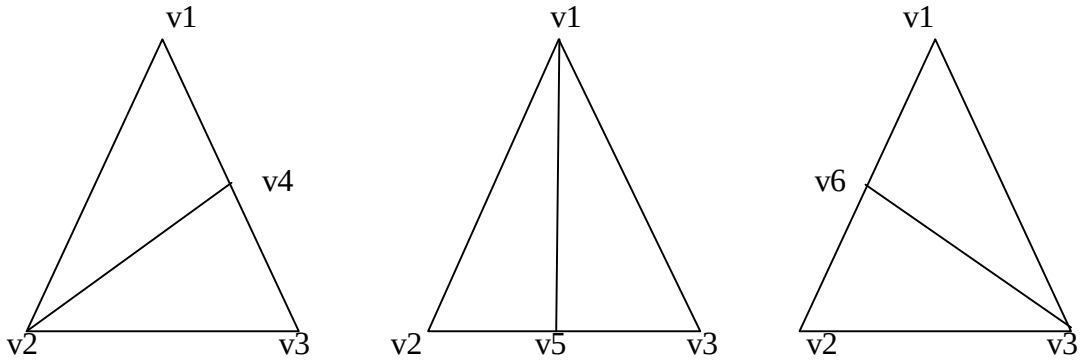
-----Evaluating perimeter of a triangle by MT2-----

Source	Ratio	Result:
S1	2	TRUE
S2	!2	FALSE
S3	!2	FALSE
S4	!2	FALSE
S5	!2	FALSE
S6	2	TRUE
S7	!2	FALSE
S8	!2	FALSE
S9	!2	FALSE
S10	!2	FALSE
S11	2	TRUE
S12	!2	FALSE
S13	!2	FALSE
S14	!2	FALSE
S15	!2	FALSE
S16	2	TRUE
S17	!2	FALSE
S18	!2	FALSE
S19	!2	FALSE
S20	!2	FALSE

Figure 4.3 Result of Metamorphic Relation 2.

Metamorphic relation 3:

In this relation we are considering the fact that partitioning a triangle into two parts by drawing a line from a vertex to the midpoint of opposite side and adding area of both the triangle must be same in all the three cases generated by repeating the same procedure on three sides.



In three triangles shown above

$v4 = \frac{v1+v3}{2}$ $v5 = \frac{v2+v3}{2}$ $v6 = \frac{v1+v2}{2}$ ⇒ 2

area(v1,v2,v4) + area(v2,v3,v4) must be equal to

area(v1,v2,v5)+area(v1,v3,v5) must be equal to

area(v1,v3,v6)+area(v3,v2,v6). ⇒ MT3

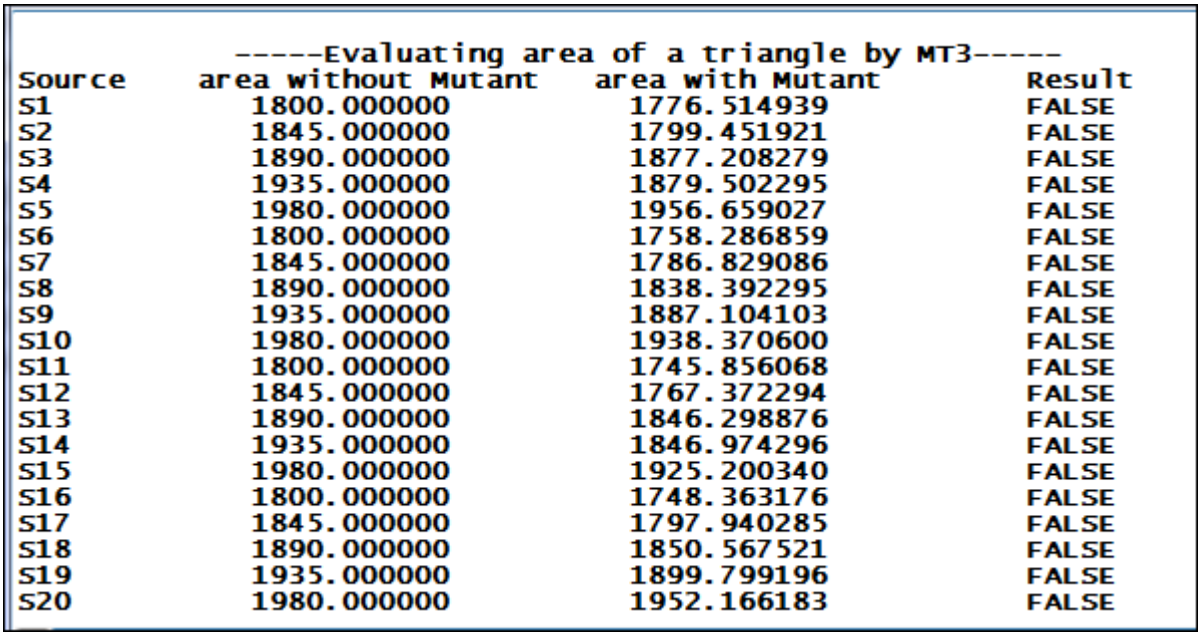
The test data mentioned in Table 4.1 is input in the below algorithm output of which is shown in Figure 4.4.

Algorithm:

1. Read input.txt as coordinates x1, y1, x2, y2, x3, y3.
2. Break the triangle with coordinates read in step 1 into six as marked in 2.
3. Apply MT3 on the triangles formed.
4. Record the output of comparison of areas in a separate text file (result3.txt ï Figure 4.4)

5 repeat step 1 to 4 till file end.

6. Exit.



-----Evaluating area of a triangle by MT3-----

Source	area without Mutant	area with Mutant	Result
S1	1800.000000	1776.514939	FALSE
S2	1845.000000	1799.451921	FALSE
S3	1890.000000	1877.208279	FALSE
S4	1935.000000	1879.502295	FALSE
S5	1980.000000	1956.659027	FALSE
S6	1800.000000	1758.286859	FALSE
S7	1845.000000	1786.829086	FALSE
S8	1890.000000	1838.392295	FALSE
S9	1935.000000	1887.104103	FALSE
S10	1980.000000	1938.370600	FALSE
S11	1800.000000	1745.856068	FALSE
S12	1845.000000	1767.372294	FALSE
S13	1890.000000	1846.298876	FALSE
S14	1935.000000	1846.974296	FALSE
S15	1980.000000	1925.200340	FALSE
S16	1800.000000	1748.363176	FALSE
S17	1845.000000	1797.940285	FALSE
S18	1890.000000	1850.567521	FALSE
S19	1935.000000	1899.799196	FALSE
S20	1980.000000	1952.166183	FALSE

Figure 4.4 Result of Metamorphic Relation 3.

4.2 Applying Genetic Algorithm

In the outcome of Metamorphic testing, we can clearly note that still there are certain test cases/input for which the program is still computing correctly. The result of three metamorphic relations shown above in Figure 4.2, 4.3 and 4.4 still have some cases where the metamorphic relations are unable to find the fault and which passed the Metamorphic Testing also..

The main focus of testing is to have as many test cases as possible because presence of more test cases will ensure that the fault is detected. So we should generate more test cases from the successful test cases which passed the Metamorphic Testing to test the software more effectively.

For this purpose we will apply Genetic Algorithm here. Two operators of Genetic Algorithm which we are using are crossover and mutation. We are applying GA to the results of MR1 only.

Algorithm used for Genetic Algorithm is given below:

1. Extract the test cases for which metamorphic relation failed to identify fault into a text file named MT1_TRUE.txt.

2. Read MT1_TRUE.txt taking two test cases at a time.

(Note: For applying crossover we need two test cases at a time).

3. Apply crossover on the two test cases read in step 2.

   Note: In this case we have six parts of a test case corresponding to three vertices of a triangle in two dimensions. Applying crossover means we are interchanging the positions of the coordinates in the two test cases. This interchange can occur in number of ways like interchanging one part at a time (One Point Crossover), interchanging two parts at a time (Two Point Crossover) and so on. So we need to include all those cases of interchange as more will be the cases more is the probability to detect fault.

4. Apply mutation to the output of crossover.
   Note: Mutation is when we are changing a particular part/value of test case. This is applicable to only one test case at a time. As we have so many ways to do crossover, similarly here also we can change the six parts in many ways (one at a time, two at a time and so on).We are changing the particular coordinates by generating a random number.

5. Apply metamorphic relations on the new genes/test cases produced (implies coordinates here) and save the output in file named result4.txt (Table 4.2).
6. Repeat step 2 to 5 till file end.
7. Exit.

We have used crossover probability of 0.9 and mutation probability of 0.3 for the algorithm above.

Figure 4.5 shows the above algorithm in form of a flow chart for better understanding.

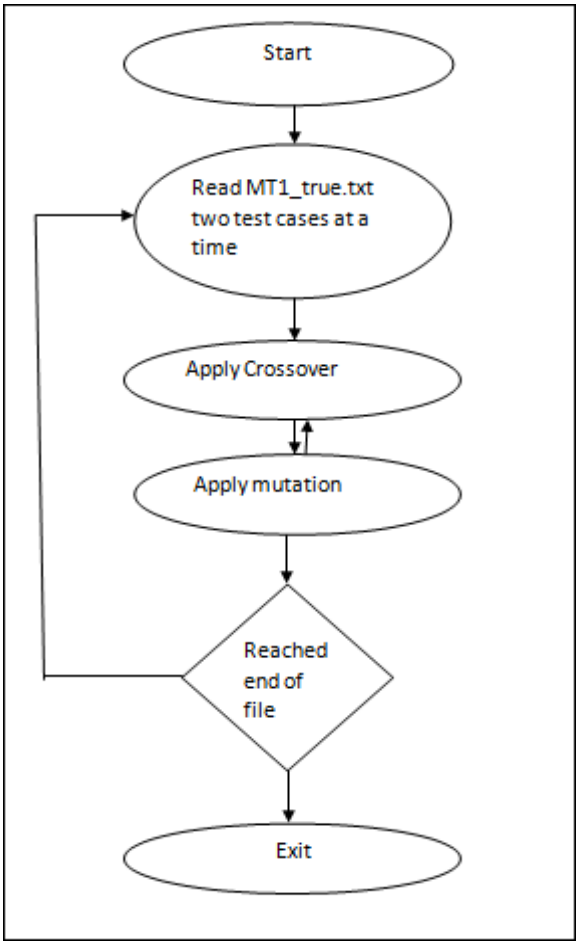


Figure 4.5 Flow chart describing steps to apply Genetic Algorithm.

For Input 1 Metamorphic relation value is False
For Input 2 Metamorphic relation value is False
For Input 3 Metamorphic relation value is False
For Input 4 Metamorphic relation value is TRUE
For Input 5 Metamorphic relation value is False
For Input 6 Metamorphic relation value is False
For Input 7 Metamorphic relation value is False
For Input 8 Metamorphic relation value is TRUE
For Input 9 Metamorphic relation value is False
For Input 10 Metamorphic relation value is False
For Input 11 Metamorphic relation value is False
For Input 12 Metamorphic relation value is False
For Input 13 Metamorphic relation value is TRUE
For Input 14 Metamorphic relation value is TRUE
For Input 15 Metamorphic relation value is False
For Input 16 Metamorphic relation value is False
For Input 17 Metamorphic relation value is False
For Input 18 Metamorphic relation value is False
For Input 19 Metamorphic relation value is False

For Input 20 Metamorphic relation value is False
For Input 21 Metamorphic relation value is TRUE
For Input 22 Metamorphic relation value is False
For Input 23 Metamorphic relation value is False
For Input 24 Metamorphic relation value is False
For Input 25 Metamorphic relation value is False
For Input 26 Metamorphic relation value is False
For Input 27 Metamorphic relation value is False
For Input 28 Metamorphic relation value is False
For Input 29 Metamorphic relation value is TRUE
For Input 30 Metamorphic relation value is False
For Input 31 Metamorphic relation value is False
For Input 32 Metamorphic relation value is False
For Input 33 Metamorphic relation value is False
For Input 34 Metamorphic relation value is False
For Input 35 Metamorphic relation value is False
For Input 36 Metamorphic relation value is False
For Input 37 Metamorphic relation value is TRUE
For Input 38 Metamorphic relation value is False
For Input 39 Metamorphic relation value is TRUE
For Input 40 Metamorphic relation value is False
For Input 41 Metamorphic relation value is False
For Input 42 Metamorphic relation value is False
For Input 43 Metamorphic relation value is False
For Input 44 Metamorphic relation value is TRUE
For Input 45 Metamorphic relation value is TRUE
For Input 46 Metamorphic relation value is False
For Input 47 Metamorphic relation value is TRUE
For Input 48 Metamorphic relation value is False
For Input 49 Metamorphic relation value is False
For Input 50 Metamorphic relation value is False
For Input 51 Metamorphic relation value is False
For Input 52 Metamorphic relation value is TRUE
For Input 53 Metamorphic relation value is False
For Input 54 Metamorphic relation value is False
For Input 55 Metamorphic relation value is False
For Input 56 Metamorphic relation value is False
For Input 57 Metamorphic relation value is False
For Input 58 Metamorphic relation value is False
For Input 59 Metamorphic relation value is False
For Input 60 Metamorphic relation value is False
For Input 61 Metamorphic relation value is False
For Input 62 Metamorphic relation value is False
For Input 63 Metamorphic relation value is TRUE
For Input 64 Metamorphic relation value is False

For Input 65 Metamorphic relation value is False
For Input 66 Metamorphic relation value is False
For Input 67 Metamorphic relation value is TRUE
For Input 68 Metamorphic relation value is False
For Input 69 Metamorphic relation value is False
For Input 70 Metamorphic relation value is TRUE
For Input 71 Metamorphic relation value is False
For Input 72 Metamorphic relation value is False
For Input 73 Metamorphic relation value is TRUE
For Input 74 Metamorphic relation value is TRUE
For Input 75 Metamorphic relation value is False
For Input 76 Metamorphic relation value is False
For Input 77 Metamorphic relation value is TRUE
For Input 78 Metamorphic relation value is TRUE
For Input 79 Metamorphic relation value is False
For Input 80 Metamorphic relation value is TRUE
For Input 81 Metamorphic relation value is False
For Input 82 Metamorphic relation value is False
For Input 83 Metamorphic relation value is False
For Input 84 Metamorphic relation value is TRUE
For Input 85 Metamorphic relation value is False
For Input 86 Metamorphic relation value is False
For Input 87 Metamorphic relation value is False
For Input 88 Metamorphic relation value is False
For Input 89 Metamorphic relation value is False
For Input 90 Metamorphic relation value is False
For Input 91 Metamorphic relation value is False
For Input 92 Metamorphic relation value is TRUE
For Input 93 Metamorphic relation value is TRUE
For Input 94 Metamorphic relation value is False
For Input 95 Metamorphic relation value is False
For Input 96 Metamorphic relation value is TRUE
For Input 97 Metamorphic relation value is False
For Input 98 Metamorphic relation value is TRUE
For Input 99 Metamorphic relation value is False
For Input 100 Metamorphic relation value is TRUE
For Input 101 Metamorphic relation value is False
For Input 102 Metamorphic relation value is False
For Input 103 Metamorphic relation value is False
For Input 104 Metamorphic relation value is False
For Input 105 Metamorphic relation value is TRUE
For Input 106 Metamorphic relation value is False
For Input 107 Metamorphic relation value is False
For Input 108 Metamorphic relation value is TRUE
For Input 109 Metamorphic relation value is TRUE

For Input 110 Metamorphic relation value is TRUE
For Input 111 Metamorphic relation value is False
For Input 112 Metamorphic relation value is False
For Input 113 Metamorphic relation value is TRUE
For Input 114 Metamorphic relation value is False
For Input 115 Metamorphic relation value is False
For Input 116 Metamorphic relation value is False
For Input 117 Metamorphic relation value is False
For Input 118 Metamorphic relation value is False
For Input 119 Metamorphic relation value is TRUE
For Input 120 Metamorphic relation value is TRUE
For Input 121 Metamorphic relation value is False
For Input 122 Metamorphic relation value is False
For Input 123 Metamorphic relation value is False
For Input 124 Metamorphic relation value is False
For Input 125 Metamorphic relation value is TRUE
For Input 126 Metamorphic relation value is False
For Input 127 Metamorphic relation value is False
For Input 128 Metamorphic relation value is False
For Input 129 Metamorphic relation value is False
For Input 130 Metamorphic relation value is False
For Input 131 Metamorphic relation value is False
For Input 132 Metamorphic relation value is False
For Input 133 Metamorphic relation value is TRUE
For Input 134 Metamorphic relation value is False
For Input 135 Metamorphic relation value is False
For Input 136 Metamorphic relation value is False
For Input 137 Metamorphic relation value is TRUE
For Input 138 Metamorphic relation value is False
For Input 139 Metamorphic relation value is False
For Input 140 Metamorphic relation value is False
For Input 141 Metamorphic relation value is False
For Input 142 Metamorphic relation value is False
For Input 143 Metamorphic relation value is False
For Input 144 Metamorphic relation value is False
For Input 145 Metamorphic relation value is False
For Input 146 Metamorphic relation value is False
For Input 147 Metamorphic relation value is False
For Input 148 Metamorphic relation value is False
For Input 149 Metamorphic relation value is False
For Input 150 Metamorphic relation value is False
For Input 151 Metamorphic relation value is False
For Input 152 Metamorphic relation value is False
For Input 153 Metamorphic relation value is False
For Input 154 Metamorphic relation value is False

For Input 155 Metamorphic relation value is TRUE
For Input 156 Metamorphic relation value is False
For Input 157 Metamorphic relation value is False
For Input 158 Metamorphic relation value is TRUE
For Input 159 Metamorphic relation value is False
For Input 160 Metamorphic relation value is False
For Input 161 Metamorphic relation value is TRUE
For Input 162 Metamorphic relation value is False
For Input 163 Metamorphic relation value is False
For Input 164 Metamorphic relation value is False
For Input 165 Metamorphic relation value is False
For Input 166 Metamorphic relation value is False
For Input 167 Metamorphic relation value is False
For Input 168 Metamorphic relation value is False
For Input 169 Metamorphic relation value is TRUE
For Input 170 Metamorphic relation value is False
For Input 171 Metamorphic relation value is TRUE
For Input 172 Metamorphic relation value is False
For Input 173 Metamorphic relation value is False
For Input 174 Metamorphic relation value is False
For Input 175 Metamorphic relation value is False
For Input 176 Metamorphic relation value is False
For Input 177 Metamorphic relation value is False
For Input 178 Metamorphic relation value is False
For Input 179 Metamorphic relation value is False
For Input 180 Metamorphic relation value is False
For Input 181 Metamorphic relation value is TRUE
For Input 182 Metamorphic relation value is False
For Input 183 Metamorphic relation value is TRUE
For Input 184 Metamorphic relation value is False
For Input 185 Metamorphic relation value is False
For Input 186 Metamorphic relation value is TRUE
For Input 187 Metamorphic relation value is False
For Input 188 Metamorphic relation value is False
For Input 189 Metamorphic relation value is False
For Input 190 Metamorphic relation value is False
For Input 191 Metamorphic relation value is TRUE
For Input 192 Metamorphic relation value is False
For Input 193 Metamorphic relation value is False
For Input 194 Metamorphic relation value is TRUE
For Input 195 Metamorphic relation value is False
For Input 196 Metamorphic relation value is False
For Input 197 Metamorphic relation value is False
For Input 198 Metamorphic relation value is False
For Input 199 Metamorphic relation value is TRUE

For Input 200 Metamorphic relation value is False

Table 4.2 Result after applying Genetic Algorithm.

Result: Out of test data consisting of 1000 coordinate set, for approx 800 inputs coordinate set, the fault was detected. For remaining 200 successful test cases, the fault is not detected even after applying Metamorphic Testing. So we applied Genetic Algorithm and the used Metamorphic Testing on these 200 successful test cases.

On analyzing the output file result4.txt as in Table 4.2 we observed that out of these 200 test cases ,we are able to detect fault for 153 test cases which means out of total 1000 test input, 953 are detected by above procedure to confirm presence of an error in the software. Efficiency of bug detection has increased from 800/1000*100=80% to 953/1000*100=95.3 by using Genetic Algorithm with Metamorphic Testing.

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

This thesis demonstrates the use of metamorphic testing as a complement to special value testing using randomly generated values to test the software. Metamorphic Testing is very helpful in detecting faults when general testing fails to detect the bugs. Along with Metamorphic Testing, Genetic Algorithm is deployed to increase the efficiency of Metamorphic Testing. The case study used in this thesis is related to verify the correctness of a function that find area and perimeter of a triangle. We have introduced a mutant deliberately in the program and then tested the above program using metamorphic relation with normal test cases, follow up test cases and more refined set of test cases generated after applying Genetic Algorithm.

The result of applying Genetic Algorithm along with Metamorphic Testing shows that the number of test cases gets increased and metamorphic relations detect errors more often as compared to before when only follow up test cases are input in the metamorphic relations. This helped in testing the program using all the possibilities which ensures correctness of the program and validates the testing methodology used. To conclude we can say that error detection capacity of Metamorphic Testing has improved by this work.

5.2 Future Scope

In this thesis, we have generated the metamorphic relations under Metamorphic Testing for testing the software manually which requires a lot of resources as well as time. So a mechanism can be devised in future to generate automated metamorphic relations to test the software more effectively and efficiently.

REFERENCES

[1] ñSoftware Testing-Overview,ò http://www.tutorialspoint.com/software_testing/software_testing_overview.htm.

[2] Roger Pressman, Software Engineering, A Practitionerôs Approach, 5th ed. New York: McGraw Hill, ISBN 0073655783,2001.

[3] ñCharacteristics of a good Case,ò https://eltguide.wordpress.com/2011/12/28/12-characteristics-of-a-good-test.

[4] ñGenetic Algorithms,ò http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html.

[5] ñGenetic Algorithm,ò http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php.

[6] ñWhat is the role of mutation and crossover probability in Genetic algorithms?,ò http://www.researchgate.net/post/What_is_the_role_of_mutation_and_crossover_probability_in_Genetic_algorithms.

[7] E.J. Weyuker, ñOn Testing Non Testable Programsò, The Computer Journal, vol.25(4),pp 645-470,1982.

[8] Tsong Yueh Chen, ñMetamorphic Testing: A Simple Approach to Alleviate the Oracle Problemò,Fifth IEEE International Symposium on Service Oriented System Engineering, 2010.

[9] Huai Liu, Fei-Ching Kuo, D. Towey, Tsong Yueh Chen, ñHow Effectively Does Metamorphic Testing Alleviate the Oracle Problem?ò, Software Engineering, IEEE Transactions on , vol.40, no.1, pp.4,22, January 2014.

[10] Tsong Yueh Chen, D.H. Huang, T.H. Tse, et al., ñ Case studies on the selection of useful relations in Metamorphic Testing.ò, Proceeding of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering, JIISIC 2004, Polytechnic University of Madrid, Madrid Spain, pp. 569ï 583 ,2004.

[11] Arlinta Christy Barus, ñMATRIX MULTIPLICATION PROGRAM: A CASE STUDY OF METAMORPHIC TESTINGò, ARPN Journal of Engineering and Applied Sciences, ISSN 1819-6608, VOL. 10, NO. 3, FEBRUARY 2015.

[12] Zhi Quan Zhou, D.H. Huang, T.H. Tse, Zongyuan Yang, Huang Haitao, T.Y. Chen, ñMetamorphic Testing and Its Applicationsò, Proceedings of 8th International Symposium on Future Software Technology (ISFST), 2004.

[13] Balinder Singh, ñImplementation of Metamorphic Testing on Spreadsheet Applicationsò, International Journal of Modern Engineering Research (IJMER) Vol.3, Issue.2, pp-990-995, March-April 2013.

[14] Amanjot Singh, Sandeep Kang, Shavinder Bajwa , ñMetamorphic Testing: Using the properties of Sutò, International Journal of Computer Technology and Applications, ISSN: 2229-6093, vol 2(5),1334-1336, 2011.

[15] Tsong Yueh Chen, Fei-Ching Kuo,Ying Liu and Antony Tang, ñMetamorphic Testing and Testing with Special Valuesò, School of Information Technology, Swinburne University of Technology.

[16] Tarun Khosla, Sushil Garg, ñMetamorphic Testing Effectiveness on Trigonometryò, International Journal of Computer Science and Technology, Vol. 2, Issue 3, September 2011.

[17] Yao Yi, Zheng Changyou, Huang Song and Zhengping Ren, ñResearch on Metamorphic Testing: A Case Study in Integer Bugs Detectionò, Research Journal of Applied Sciences, Engineering and Technology 6(16): 2951-2956, 2013 ISSN: 2040-7459; e-ISSN: 2040-7467,2013.

[18] Yao Yi, Huang Song, Ji Mengyu, ñResearch on Metamorphic Testing for Oracle Problem of Integer Bugsò, Springer Berlin Heidelberg, Volume 1,pp. 95-100,2012.

[19] Zhanwei Hui, Huang Song , Zhengping Ren , Yao Yi, ñ Metamorphic Testing Integer Overflow Faults of Mission Critical Program: A Case Studyò, National High Technology Research and Development Program of China Project and the National Science Foundation of Jiangsu Province ,China,2013.

[20] T.Y. Chen , Fei-Ching Kuo,T.H. Tse , Zhi Quan Zhou, ñMetamorphic Testing and beyondò, Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on, vol. no., pp.94-100, 19-21 Sept. 2003.