Development and Validation of Test Case Prioritization Technique using Genetic Algorithms

A Dissertation submitted in the partial fulfillment for the award of

MASTER OF TECHNOLOGY

IN

SOFTWARE ENGINEERING

by

Divya Tiwari

Roll no. 2k11/SWE/06

Under the Esteemed Guidance of

Dr. Ruchika Malhotra

Department of Computer Engineering

Delhi Technological University

New Delhi

2012-2013

DECLARATION

I hereby declare that the thesis entitled "**Development and Validation of Test Case Prioritization Technique using Genetic Algorithms"** which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree of **Master of Technology in Software Engineering** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any university or institution for the award of any degree.

Divya Tiwari

Department of Computer Engineering

Delhi Technological University,

Delhi.

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

(Govt. of National Capital Territory of Delhi)

BAWANA ROAD, DELHI-110042

Date: ____________________

This is to certify that the thesis entitled **Development and Validation of Test Case Prioritization Technique using Genetic Algorithms"** submitted by **Divya Tiwari (Roll Number: 2K11/SWE/06),** in partial fulfillment of the requirements for the award of degree of Master of Technology in Software Engineering, is an authentic work carried out by her under my guidance. The content embodied in this thesis has not been submitted by her earlier to any institution or organization for any degree or diploma to the best of my knowledge and belief.

<u>Project Guide</u>

Dr. Ruchika Malhotra

Assistant Professor

Department of Computer Engineering

Delhi Technological University, Delhi-110042

ACKNOWLEDGEMENT

I take this opportunity to express my deepest gratitude and appreciation to all those who have helped me directly or indirectly towards the successful completion of this thesis.

Foremost, I would like to express my sincere gratitude to my guide **Dr. Ruchika Malhotra**, **Assistant Professor, Department of Computer Engineering**, **Delhi Technological University, Delhi** whose benevolent guidance, constant support, encouragement and valuable suggestions throughout the course of my work helped me successfully complete this thesis. Without her continuous support and interest, this thesis would not have been the same as presented here.

Besides my guide, I would like to thank the entire teaching and non-teaching staff in the Department of Computer Science, DTU for all their help during my course of work.

DIVYA TIWARI

PUBLICATIONS AND COMMUNICATIONS

Paper published in International Journal

Malhotra, R., Tiwari, D., "Development of a Test Case Prioritization Framework using Genetic Algorithm", ACM SIGSOFT Software Engineering Notes, vol. 38, no. 3.

Paper communicated in International Journal

Malhotra, R., Tiwari, D., "An Empirical Study of Genetic Algorithm Based Test Case Prioritization Framework", Journal of Systems and Software, communicated in May 2013.

Table of Contents

CHAPTER 1 1

INTRODUCTION 1

1.1. REGRESSION TESTING 2

1.2. TEST CASE PRIORITIZATION 5

1.3. MOTIVATION OF THE WORK 7

1.4. GOALS OF THE THESIS 11

1.5. ORGANIZATION OF THESIS 12

CHAPTER 2 15

LITERATURE SURVEY 15

CHAPTER 3 21

GENETIC ALGORITHM 21

3.1. INITIALIZATION 23

3.2. EVALUATION 24

3.3. SELECTION 25

3.4. CROSSOVER 26

3.5. MUTATION 27

CHAPTER 4 29

PROPOSED FRAMEWORK FOR TEST CASE PRIORITIZATION 29

4.1. THE FRAMEWORK 29

4.2. MODIFIED APBC METRIC ($APBC_m$) 30

4.3. ADDITIONAL MODIFIED LINES OF CODE COVERAGE (AMLOC) GRAPH 32

4.4. GENETIC ALGORITHM BASED TOOL 33

4.5. ILLUSTRATION 38

4.5. VALIDATION OF RESULTS 44

4.6. DRAWBACKS OF THE FRAMEWORK 46

CHAPTER 5 ... 48

EMPIRICAL STUDY ... 48

5.1. RESEARCH QUESTIONS ... 48

5.2. EFFICACY MEASURE ... 49

5.3. TEST CASE PRIORITIZATION TECHNIQUE ... 49

5.3.1. MODIFIED FRAMEWORK FOR TEST CASE PRIORITIZATION: OBJECT OF STUDY ... 49

5.3.2. MODIFIED APBC METRIC ($APBC_m$) ... 51

5.3.3. GENETIC ALGORITHM BASED TOOL ... 52

CHAPTER 6 ... 54

DATA COLLECTION AND ... 54

EXPERIMENT DESIGN ... 54

6.1. DATA COLLECTION ... 54

6.1.1. SUBJECT PROGRAMS ... 55

6.1.2. TEST SUITE ... 57

6.1.3. FAULTS ... 58

6.2. EXPERIMENT DESIGN ... 60

CHAPTER 7 ... 62

RESULT ANALYSIS ... 62

***7.1. (RQ1) Is $APBC_m$ a better comparator metric than APBC?* ... 62**

***7.2. (RQ2) Can the proposed framework improve the rate of fault detection?* ... 64**

7.3. DISCUSSIONS ... 66

CHAPTER 8 ... 68

CONCLUSIONS ... 68

REFERENCES ... 72

List of Figures

Figure 1.1. Process of Regression Testing 2

Figure 3.1. Genetic Algorithm Cycle 22

Figure 3.2 Roulette Wheel Selection Strategy 25

Figure 4.1 Framework For Test Case Prioritization [36] 30

Figure 4.2 Roulette Wheel for Sample Data 36

Figure 4.3 AMLOC Graph for T1(Ranked as most fit if APBC is

Used for comparison of candidates in GA) 37

Figure 4.4 Graph for T3(Ranked as most fit if $APBC_m$ is

Used for comparison of candidates in GA) 37

Figure 4.5 Source Code of Triangle Program [34] 39

Figure 4.6 Modified Triangle.c code 42

Figure 4.7 Less AMLOC values per test case in TS1 produced in Experiment 1 44

Figure 4.8 Greater convexity(improved coverage rate) graph using $APBC_m$ 45

Figure 5.1. Modified Framework for Test Case Prioritization 50

Figure 6.1. Experiment Design or Process 60

Figure 7.1. Results of JTopas Project 63

Figure 7.2. Results of Xml-Security Project 63

Figure 7.3. Increase in Fault Deatection Rate for Jtopas Project. 65

Figure 7.4. Increase in Fault Detection Rate for Xml-Security Project. 65

List of Tables

Table 4.1 Block Coverage Matrix 32

Table 4.2 Number Of Modified Lines Of Code Covered By Blocks 32

Table 4.3 Comparisons of Different Test Case

Sequences (Candidates in the TCP problem) 35

Table 4.4 Cumulative, Normalized $APBC_m$ Values for Sample Data 35

Table 4.5 Candidates Selected during some iteration (for sample data) 36

Table 4.6 Original Set of Test Cases for Triangle program [34] 38

Table 4.7 Lines of Code Comprising individual Blocks 40

Table 4.8 Block coverage 41

Table 4.9 Weight of Blocks 43

Table 4.10 Results of Experiment 1 46

Table 4.11 Results of Experiment 2 46

Table 6.1. Subject Programs 56

Table 6.2.Weight Matrix for JTopas Project 56

Table 6.3. Class Coverage Matrix for JTopas Project. 57

Table 6.4. Fault Matrix of JTopas 59

Table 6.5. Fault Matrix of Xml-Security 60

Table 7.1. Results of JTopas project 63

Table 7.2. Results of Xml-Security Project 64

ABSTRACT

Software evolution is a term used for repeated modifications in a software system caused by changing existing requirements, emerging new requirements or bug fixes. A small change in the software system may lead to malfunctioning of the existing software system. Thus, there arises the need for Regression Testing. Regression Testing is the process of testing a software system after it has undergone changes. It aims to detect faults, if any, that may have been introduced into the software system as a result of these changes. It requires rerunning the modified test suite but rerunning may significantly increase the time and effort required for regression testing. Test Case Prioritization aims to reduce the time and effort required in regression testing by prioritizing the test cases so as to increase the rate of fault detection. In this thesis we propose and validate a test case prioritization framework for object oriented systems based on Genetic Algorithm (GA) and using modified Average Percentage of Block Coverage ($APBC_m$) metric as fitness function in GA based tool. The results are obtained using two open source softwares JTopas and Xml-Security. We have used fault coverage criteria to validate the prioritized test case sequence produced by the proposed framework when applied to two open source projects JTopas and Xml-Security. The results show that the framework can be used to obtain better prioritized test case sequences with higher fault detection rate.

CHAPTER 1

INTRODUCTION

Software maintenance, commonly known as “software evolution”, is a rigorous activity that during which changes are made to the existing software system. Such changes may be the result of a debugging activity, or implementing a new requirement or changing existing requirements. In any case, introduction of a change in a system is followed by several activities like retesting the software and ensuring that no new faults are introduced by such a change. This is known as Regression testing. Looking at the current scenario of regression testing, it is clearly understood that the testers need to improve their practices and strategies for testing in order to deliver a better quality software system in less time and efforts. Our work in this thesis aims to help testers understand the significance of using practical weight factors during test case prioritization so that overall quality of regression testing is improved.

This chapter proceeds with an in depth knowledge of how a software evolves, what is regression testing, what are the problems faced by the testers and developers during regression testing and how the work presented in this thesis, in the field of test case prioritization, contributes to solving these problems and challenges.

1.1. REGRESSION TESTING

***"Software Testing is the process of executing a program with the intent of finding errors"* [1]**. ***"It is an investigation that is conducted to provide stakeholders the information about the quality of the product/service under test"* [2]**. Software maintenance is an essential activity that allows developers to modify an existing software system as and when required, in order to meet certain objective. Such objectives may include fixing the defects that are reported by the clients, after the software system has been delivered and deployed or keeping pace with the changing requirements or emerging new requirements. Regression Testing is the process of testing a software system during maintenance phase. It is a type of software testing that seeks to uncover new software bugs in existing functionality of the software system after it has undergone changes, such as enhancements, bug fixes, configuration changes, etc have been made to them.

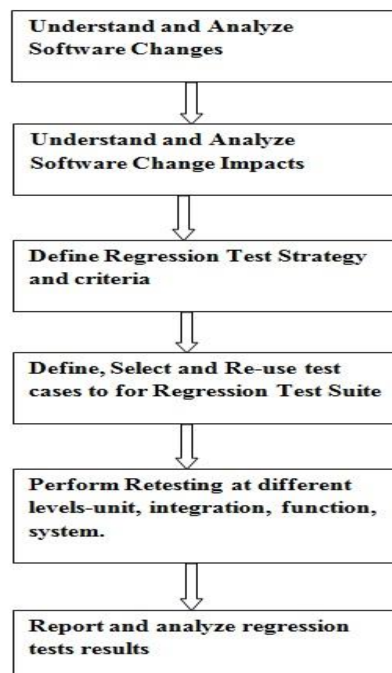

Figure 1.1. Process of Regression Testing

Figure 1.1 above gives a brief introduction to the overall process of regression testing. In our work we try to improve the quality of regression testing process by focusing on step 1 step 3 and step 4. The objective of Step 1 is to identify those parts of the software system that have undergone change and require the attention of tester. There may be several such parts. A tester may therefore face the problem of identifying significant changes. Our idea of assigning weights to different parts of code helps the testers to distinguish some of the significant changes from lesser significant ones. Test case prioritization helps a tester to smartly order the test cases so that when they are executed in that order, most of the faults are exposed earlier. This is achieved in step 4. Step 3 focuses on defining the scope and coverage criteria for testing. The answer to the question when to stop regression testing is formulated during this step. One should always remember that testing smart but not hard, is the key objective of any testing activity.

Regression Testing plays an important role in maintaining the quality of the subsequent releases of the software system and also accounts for large proportion of software maintenance cost. For the same reasons many researchers have focused on reducing the maintenance cost and effort through regression test selection, test case minimization, and test case prioritization techniques. Common methods of regression testing include rerunning previously-completed tests along with the new test cases for the modified parts of the software system and checking whether the program behavior has changed and whether previously-fixed faults re-emerge or not. Regression testing can be used to test a system efficiently by smartly selecting an adequate, minimum set of tests that can achieve certain testing objective like fault coverage, code coverage, etc..

Experience has shown that during software evolution, new faults may emerge as a result of software changes. Often this reemergence occurs because a fix gets lost through poor revision control practices (or simple human error in revision control). Most of the times the code to fix a bug or a problem is "delicate" or prone to changes in future, in the sense that it considers the problem as a specific case when it was first observed but not in more general cases which may arise over the lifetime of the software. Also, the code to fix a problem in one area adversely causes malfunctioning of some other area of the software system. Lastly, it is possible that, when some feature is re-implemented, some of the same mistakes that were done in the original implementation of the feature are repeated unintentionally. Therefore, in most software development scenarios, when a bug is encountered and fixed, it is often considered a good testing practice to record the test case that exposes the bug and re-run that test regularly after subsequent changes to the program. Although most testers do this manually through code instrumentation using programming techniques, it is always a better option to use automated testing tools. There are software tools that provide a testing environment to execute all the regression test cases automatically and some projects even set up automated systems to automatically re-run all regression tests at specific time intervals and report any failures (which may imply a regression or an outdated test). Common strategies are to execute such a system after every successful compilation (for small projects), at regular time intervals like once a week.

Regression testing is an integral part of the software maintenance activities and plays an important role in preserving or enhancing the quality of software system as it evolves over time. In the software industry, it has been observed that regression testing is usually

performed by a software quality assurance team after the development team has completed the development work. However, defects encountered during this stage are the most expensive to fix and therefore regression testing accounts for a large proportion of the software maintenance cost. To address this problem, unit testing practices have been improved. Although developers have always written test cases as part of the development cycle, these test cases have generally been either functional tests or unit tests that verify only intended outcomes. Developer testing compels a developer to focus on unit testing and to include both positive and negative test cases.

1.2.TEST CASE PRIORITIZATION

Software maintenance is an essential activity that allows developers to modify an existing software system as and when required, in order to meet certain objective. Such objectives may include fixing the defects that are reported by the clients, after the software system has been delivered and deployed or keeping pace with the changing requirements or emerging new requirements. Regression testing is the process of testing the modifications in a software system once it has undergone changes and detect the new faults that may have been introduced into the system as a result of these changes. It plays an important role in maintaining the quality of the subsequent releases of the software system but it also accounts for large proportion of software maintenance cost. It requires ample amount of time and effort. In the scenario where developers have to pace up with the increasing competition in the market, smart testing within budget and time is essential. For the same reasons many researchers have focused on reducing the maintenance cost and effort through regression test selection, test case minimization, and test case prioritization techniques. Test case

prioritization techniques allow testers to execute the test cases in an order that achieves some testing objective at a faster rate. There are multiple testing objectives like rate of fault detection, rate of code coverage, etc. The test case prioritization problem as defined by Rothermal et al.[3] is stated below::

Given a Test Suite ***'T'*** ,a set '***PT***' of all permutations of T and a function '***f'*** 'that maps PT to real numbers, test case prioritization technique aims to find a **T'∈ PT** such that **(∀T'') (T'' ∈ PT) (T''≠ T)[f(T') ≥ f(T'')].**

For an efficient and smart regression testing it is important to understand what part of software system actually needs to be focused. This can be achieved through weights. Assigning weights to different parts of code highlights the relative importance of the code to be tested. Statistics show that of all the features provided with a software system only a few are used by the end users and most of the bugs reported by them are associated with the modules implementing those features. Therefore, this fact can be exploited and used to assign weights to different parts of code. Similarly for version specific test case prioritization, testers should focus on parts of code that are highly error prone. Claes et al. [5] in their study revealed that certain programming constructs are more error prone than others and defect data can be used to identify them. Empirical studies performed till now to compare the different techniques of test case prioritization, have used either APBC (Average Percentage of Block Coverage) or APFD (Average Percentage of Fault Detection) metric. Both these metrics reveal the rate at which the faults are discovered or the rate at which the code coverage is achieved. Still there is a drawback of using these metrics as discussed by Elbaum et al. [4]. The APFD metric requires faults to be known prior to prioritization and treats all faults equally severe. Elbaum et al. [4] in their work tried to improve the APFD metric

incorporating the knowledge like fault severity and cost of executing a test case. Similarly APBC metric considers that all blocks are equally likely to contain errors. In their paper [5], authors emphasize that certain programming constructs are more error prone than others**.** The defect data can be used to identify these programming constructs. Thus, in an application, certain blocks contribute more to faults than others. Our work in [36] focuses to exploit this fact and assign weights to blocks. The APBC metric assumes that all blocks are equally error prone. In the work presented in this thesis, we have tried to extend the test case prioritization framework presented in [36] to object oriented systems and validate the new framework using APFD metric thereby ensuring that the performance of test case prioritization techniques improves by including the knowledge about significance of blocks and error proneness of blocks in the form of weights.

1.3.MOTIVATION OF THE WORK

The need for Test Case Prioritization has its roots in the seven fundamental principles of Software Testing some of which are equally applicable to regression testing. These fundamental principles are as follows:

a. ***Exhaustive Testing is not possible.*** This implies that the entire set of possible test cases cannot be executed. Therefore it is important to minimize and prioritize test cases so that faults can be detected at a higher rate.

b. ***Early Testing.*** This implies that testing activities should be started early and move parallel with the development of software. Thus, test case prioritization should focus on prioritizing the test cases on the basis of requirement specification.

c. ***Testing shows presence of errors.*** This implies that one cannot be assured that a software is free from errors. It shows errors are present but cannot assure their absence.

d. ***Accumulation of errors***. This implies that there is no equal distribution of errors within one test object. All errors may not be localized to same place in code but it is more likely to happen that some errors may be found where one error is found. The testing process must be flexible and respond to this behavior. Thus, all parts of code are not equally error prone. Hence, the need of weightage arises.

e. ***Fading effectiveness.*** This implies that the effectiveness of tests fades over time. If test cases are only repeated, they do not expose new errors. Errors, remaining within untested functions may not be discovered. In order to prevent this effect, test suites must be modified and reworked from time to time.

f. ***Testing depends on context***. No two systems are the same and therefore, can not be tested the same way. Testing intensity, when to stop testing etc. must be defined individually for each system depending on its testing context.

g. ***False conclusion: no errors equals usable system***. Error detection and removal does not guarantee a usable system matching the users expectations. Early integration of units and rapid prototyping prevents unhappy clients and discussions.

Besides, there are several drawbacks of Average Percentage of Block Coverage (APBC) and Average Percentage of Fault Detected (APFD) metric. The APFD metric requires faults to be known prior to prioritization and treats all faults equally severe. Elbaum et al. [4] in their work tried to improve the APFD metric incorporating the knowledge like fault severity and cost of executing a test case. Similarly APBC metric considers that all blocks are equally

likely to contain errors. It does not consider the practical weight factors like significance of the blocks covered. Several factors that can be used to highlight the significance of blocks are as follows:

a. Some blocks of code, like exception handling code, are not frequently executed. Most of the features provided with a software system remain unused throughout the lifetime of the software. Changes in the modules implementing such features of software should be considered least important. Understanding the fact that since such features will not be used in future by the end users, undetected faults in these features shall not be reported. This saves time and effort of testers which can then be devoted to testing of other highly usable modules of the system.

b. Certain programming constructs are more error prone than others**[4].** For instance it is common to commit errors in looping constructs than in simple input output statements. Long and complex expressions are highly prone to logical errors. Therefore, not only the amount of changes but the type of changes made to different blocks of code also affects the error-proneness of that block. It is therefore important to identify these constructs. The defect data can be used to find these programming constructs. In their paper **[4],** the authors have proposed a method to identify programming and design constructs that contribute more than expected to the defect statistics. **Zengkai Ma et al [32]** have shown how analysis of program structure can be used to find important modules (eg. methods) in a source code. **Lei Zhao et al [33]** have presented a methodology using **Control Flow Analysis** to quantitatively analyze how the basic blocks contribute to failures.

c. Some blocks of code contain greater percentage of modified line. Assigning a priority to these blocks shall result in faster exposure of faults since modifications are most likely to contain errors.

d. It is observed that corrective changes are less error prone than adaptive changes. Thus, the kind of software change carried out-adaptive, preventive, corrective also influences the error-proneness of blocks.

In their paper [5], authors emphasize that certain programming constructs are more error prone than others**.** The defect data can be used to identify these programming constructs. Thus, there arises a need to differentiate error prone blocks of code from less error prone blocks of code. Criteria such as the number of modifications made to a block and the type of modifications made to a block can be used as a measure for error proneness. Apart from this complexity is also a measure for error proneness, i.e., highly complex blocks are more likely to contains errors than less complex blocks.

In [36] we had proposed a test case prioritization framework based on Genetic algorithm and assigned weights to blocks of code according to the number of modifications made to the block. It also used a new improved metric $APBC_m$ that used the knowledge of weights to prioritize the test cases. We thereby identified two main problems with the test case prioritization framework presented in [36]. A block in the original Average Percentage of Block Coverage (APBC) metric and modified Average Percentage of Block Coverage ($APBC_m$) metric refers to the basic block, that is a block consisting of all sequential statements such that if first statement in block gets executed then all the consecutive

sequential statements in that block get executed. Since it is not practically feasible to calculate the number of changes at the level of blocks, this metric had limited application to small sized programs and not software systems. Also, most of the software systems developed today use object oriented approach and the framework discussed in [36] was not applicable to the object oriented systems. Therefore, in this work we modifiy the test case prioritization framework presented in [36] by considering a class as a block unit in the modified framework and validate it. That is in the modified framework, the GA based tool uses $APBC_m$ metric which considers a class as a block thus increasing the utility of the framework. For the purpose of validation we used two open source projects, JTopas and Xml-Security and compared the test case sequences produced by the framework using the APFD metric.

1.4. GOALS OF THE THESIS

The goal of the work in this thesis is summarized below:

a. ***To extend the test case prioritization framework presented in [36] to object oriented systems-*** As discussed earlier, the main problem with that framework presented in [36] is that it cannot be applied to the object oriented systems. In this thesis we aim to enhance and modify the framework by considering a class as a block unit so that it can be applied to object oriented systems.

b. ***To validate the proposed framework for test case prioritization using two open source projects through experimentation-*** We also aim to validate the proposed test case

prioritization framework using two open source software projects. By doing so we wish to generalize the results.

c. ***To analyze the prioritized test case sequences produced by the framework using Average Percentage of Fault Detected (APFD) metric-*** As, that fault detection is the main objective of any testing activity, we aim to analyze the results for effectiveness in terms of fault detection.

d. ***To compare the APBC and $APBC_m$ metric used as fitness function in the GA based tool-*** We also aim to compare the two metrics APBC and $APBC_m$ and show that the latter is a better comparator metric. By doing so, we support our claim that use of weights to identify error prone and significant changes in the software system allows testers to shift their focus of testing activity to some specific areas thereby reducing testing time and effort.

It can be observed that our goals are focused on improving the quality of regression testing. We aim to provide a framework for test case prioritization which can help testers in perfectly managing their time and resources during regression testing.

1.5.ORGANIZATION OF THESIS

This thesis is organized as follows:

Chapter 2 discusses the previous work done in the field of test case prioritization. This includes the extensive study of various test case prioritization techniques that have been proposed in the literature so far. It also highlights some of the most relevant works in the direction of field of work presented in the thesis

Chapter 3 gives a comprehensive study of Genetic Algorithm. This chapter is dedicated to a profound study of historical background of Genetic Algorithm including details of it's origin, various phases of Genetic Algorithm, significance and utility of the algorithm. We also exemplified the working of the algorithm with some sample data.

Chapter 4 focuses on the proposed framework for Test Case Prioritization problem including the details of the original framework presented in [36] and the modified framework. It also describes the $APBC_m$ metric, GA based tool and Additional Modified Lines of Code Coverage (AMLOC) graph. It also lays down certain guidelines regarding computation of weight factors that are to be included in $APBC_m$ metric. A detailed study followed by application of framework to a small benchmark program has been shown.

Chapter 5 presents the research questions that we aim to address in this thesis. We also describe in detail the modified test case prioritization framework.

Chapter 6 comprises of the empirical data collection for two (Free Open Source Softwares) FOSS projects-JTopas and Xml-Security. It also includes the details of the two experiments and how the experiments were performed.

Chapter 7 presents a detailed analysis of the results obtained. In this chapter we compare and assess the results of the experiments and show improvement in results after application of the modified APBC metric in the tool developed for test case prioritization using Genetic Algorithm. We also provide answers to the research questions formulated in chapter 5.

Chapter 8 presents the conclusions of the thesis and future work.

CHAPTER 2

LITERATURE SURVEY

Test case prioritization is dedicated to finding an ideal ordering of test cases for testing, so that certain testing objective is achieved and the tester obtains maximum benefit in terms of saving time and effort and finding maximum number of faults as early as possible, even if the testing is prematurely halted at some arbitrary point. This approach was first introduced by **Wong et al.[41]**. However, it had a limitation that it was only applicable to test cases which were selected by firstly applying some test case selection technique. **Harrold et al.[42]** proposed and assessed a more generalized approach. Thereafter several techniques were developed and analysed for effectiveness. Most of these techniques were focused on version specific test case prioritization using structural metrics and machine learning techniques but the goal remained same that is to maximize early fault detection and achieve certain level of confidence in the software system.

The test case prioritization problem definition given in section 1.2 is a generalized definition and says nothing about the versions of the program under test and the changes carried out from one version to another. Generally, a tester is more interested in the rate of fault detection so the test cases should be executed in the order that maximizes early fault

detection. However, the fault detection information is not known beforehand until the testing is completed. In order to overcome this problem of knowing which tests reveal faults, test case prioritization techniques depend on surrogates, hoping that early maximization of a certain chosen surrogate property will result in increased fault detection rate. If the regression testing is performed in a controlled manner, the result of prioritization can be assessed by executing test cases according to the fault detection rate.

Structural coverage is a metric that is often used as the prioritization criterion **[3-11]**. The intuition behind the idea is that early maximization of structural coverage will also increase the chance of early maximization of fault detection. Although, the goal of test case prioritization remains that of achieving a higher fault detection rate, the prioritization techniques actually aim to maximize early coverage. **Rothermel et al.[3]** presented empirical studies of several test case prioritization techniques. They applied the same algorithm with different fault detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total, and FEP-additional. The branch-total approach prioritizes test cases according to the number of branches covered by individual test cases, while branch-additional prioritizes test cases according to the additional number of branches covered by individual test cases. The statement-total and statement-additional approaches apply the same idea to program statements, rather than branches. Algorithmically, 'total' approaches are essentially instances of greedy algorithms whereas 'additional' approaches are essentially instances of additional greedy algorithms. The FEP of a test case is measured using program mutation. Program mutation introduces a simple syntactic modification to the program source, producing a

mutant version of the program. This mutant is said to be killed by a test case if the test case reveals the difference between the original program and the mutant. Given a set of mutants, the mutation score of a test case is the ratio of mutants that are killed by the test case to the total kill-able mutants. The FEP-total approach prioritizes test cases according to the mutation score of individual test cases, while the FEP-additional approach prioritizes test cases according to the additional increase in mutation score provided by individual test cases. The FEP criterion can be constructed so that it is at least as strong as structural coverage; to kill a mutant, a test case not only needs to achieve the coverage of the location of mutation but also to execute the mutated part with a set of test inputs that can kill the mutant. In other words, coverage is necessary but not sufficient to kill the mutants. It is important to understand that all the 'additional' approaches may reach 100% realization of the utilized surrogate before every test case is prioritized. For example, achieving 100% branch coverage may not require all the test cases in the test suite, in which case none of the remaining test cases can increase the branch coverage. The results are usually evaluated using the Average Percentage of Fault Detection (APFD) metric. Higher APFD values denote faster fault detection rates. When plotting the percentage of detected faults against the number of executed test cases, APFD can be calculated as the area under the curve.

A wide range of metrics for test case prioritization have been proposed and studied. Earliest techniques revolved around coverage metrics like **Statement-total ,Statement-Additional, Branch-total, Branch-Additional, Fault Exposing Potential(FEP)-total, FEP–Additional [3-11]. Jones and Harrold [12]** applied greedy approach for prioritization to **Modified Condition/Decision Coverage.** The main idea behind using coverage based metric was that

maximizing structural coverage of code may maximize the fault detection. The resulting Test Case prioritized sequence produced by various techniques were compared, for effectiveness using APFD metric(Average Percentage Of Fault Detection) or APBC(Average Percentage of Block Coverage). The work in field of test case prioritization technique is not limited to structural coverage. Several other machine learning techniques have also been employed in this area. **Leon and Podgurski [13]** have used clustering techniques to distinguish test cases associated with highly error prone regions of code from those associated with less error prone regions. They prioritized the test cases based on the density of clusters formed by the test cases. Recently **Ryan Carlson et al [14]** also presented a clustering based approach to test case prioritization. They specifically applied four different metrics code coverage, code complexity, fault history, and combination of code complexity and fault detection ratio in order to prioritize test cases within each cluster. **Tonella et al. [15]** used Case Based Reasoning(CBR) to prioritize the test cases. They included human knowledge of test cases for pair wise test case comparison. **Yoo et al. [16]** combined human based prioritization technique (incorporates knowledge of humans about test cases) with clustering technique. **Kim and Porter [17]** took an execution history based approach, borrowing from statistical quality control. **Mirab and Tahvildari [18]** exploited Bayesian networks for test case prioritization. They have used information like fault proneness, code coverage, modified elements in program while providing feedback to Bayesian networks. Several model based test case prioritization techniques have also been proposed in literature including work of **Korel et al. [19-21], Rajib Mall et al. [22].** Relevant to the work presented in this paper is the work of **Siripong Roongruangsuwan et al [23], Elbaum et al. [3]** and several other cost effective Prioritization techniques **[8, 24-27].** The authors **Siripong Roongruangsuwan et**

al. [23], in their paper test case prioritization technique with practical weight factors. In their work they have discussed **Elbaum et al. [3]** emphasizes on improving APFD metric by incorporating the fact all faults are not equally severe. The drawback of APFD metric is that it requires faults to be known beforehand. Similar to his work, the idea presented here focuses to improve the APBC metric, taking into account that not all blocks of code are equally likely to have error. In contrast to most of the cost effective prioritization techniques the $APBC_m$ assigns weights to blocks of code rather than assigning weights to test cases.

Avritzer et al. [40] presented a test case generation technique for the software systems that can be modeled using Markov chains. Although the term "prioritization" is not used by the authors, the technique generates test cases in an order that covers a greater proportion of the software states that are most likely to be reached in the field earlier in testing thereby increasing the chances of faults getting revealed earlier in testing. The work presented by **Malhotra et al. [36]** is inspired by several cost effective prioritization techniques in the past like **Roongruangsuwan et al. [23]** emphasized on the necessity of incorporating practical weight factors. **Elbaum et al. [3]** introduced an improvement to APFD metric to make it more cost effective. Similarly in [36], the authors proposed a framework for test case prioritization using a new metric $APBC_m$ as fitness function in GA based tool. Although this work is inspired by several cost effective techniques in the past it differs from them in the sense that it assigns weights to code and not to test cases. The authors exploited the fact that all blocks of code are not equally significant and error prone. In this work we modify the test case prioritization framework presented in [36] so as to extend the applicability of the framework to object oriented systems and follow a systematic procedure to validate and

analyze it using two open source projects JTopas and Xml-Security. We then compare the test case sequences produced by the framework using the APFD metric.

CHAPTER 3
GENETIC ALGORITHM

The term genetic algorithm, almost universally abbreviated nowadays to GA, was first used by John Holland, whose book *"Adaptation in Natural and Artificial Systems"* of 1975 was instrumental in creating something which is now a flourishing field of research and application that goes beyond the original GA. Many people now use the term evolutionary computing or evolutionary algorithms (EAs), in order to cover the developments of the last 10 years. However, in the context of meta-heuristics, it is probably fair to say that GAs in their original form encapsulate most of what one needs to know. Holland's contribution and influence in the development of the topic has been very important, but several other scientists with different backgrounds were also involved in developing similar ideas. In 1960s in Germany, Ingo Rechenberg and Hans-Paul Schwefel developed the idea of the Evolutionsstrategie (in English, evolution strategy), while also in the 1960s Bremermann, Fogel and others in USA implemented their idea for what they called evolutionary programming. The common thread in these ideas was the use of mutation and selection—the concepts at the core of the neo-Darwinian theory of evolution. Although some promising results were obtained, evolutionary computing did not really take off until the 1980s. Not the

least important reason for this was that the techniques needed a great deal of computational power.

The term Genetic Algorithm has its origin in the Biological Sciences. It works on the famous **Darwins Theory** which emphasizes on the survival of the fittest. The work presented here emphasizes on using the genetic algorithm with **modified APBC** as fitness function to search for the fittest candidate (a test case sequence). Genetic algorithm explains the notion of evolution. The fittest candidates in a population are carried to the next generation of population. The Genetic Algorithm is a heuristic search. The input of the algorithm is a collection of some permutations of the test suite and output of the algorithm is a prioritized test case sequence. The figure 3.1 below gives an overview of the working of Genetic algorithm.

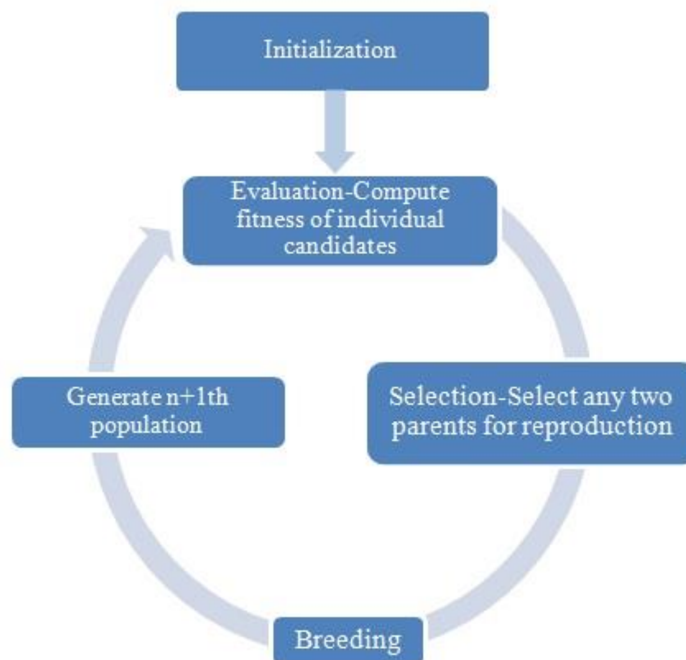

Figure 3.1. Genetic Algorithm Cycle

The following sections explain the various phases of genetic algorithm. The genetic algorithm has four phases primarily Initialization, Evaluation, Selection, Breeding (Crossover and Mutation). These phases are explained below.

3.1. INITIALIZATION

The first phase focuses on initializing the population. It is important to identify the candidate solutions for a problem and the way they are encoded in the population. Various encoding strategies like binary encoding, permutation encoding, real value encoding, tree encoding,etc. A brief description of various strategies for encoding is given below.

a. **Binary Encoding-** In this a chromosome is usually represented as a string of 0's and 1's. This had been the most commonly used form of encoding strategy mainly because of its simplicity. The binary digits usually represent presence or absence of some property in the chromosomes. For instance the knapsack problem uses this kind of encoding.

b. **Permutation Encoding-** A candidate encoded using this strategy is represented as a sequence of numbers which usually denotes a permutation. The ordering problems like Travelling Salesman problem and in our case test case prioritization problem uses this kind of encoding.

c. **Value Encoding-** In this a chromosome is represented using a string of values like real numbers, names, complicated objects, etc. for instance the problem of finding the optimal weights for the neural network uses this kind of encoding.

d. **Tree Encoding-** In this a chromosome is represented as a tree of objects such as functions, commands or operators in a programming language, etc. An example is an S-Expression tree. The problem using this kind of encoding is-finding a mapping from given inputs to known outputs.

Different techniques are applied to randomly generate a few candidates. The convergence of the algorithm depends upon these candidates, better the candidates faster the algorithm converges. We have used ***Johnson-Trotter*** **[28-30]** algorithm to generate the permutations of the test suite and use this as initial population.

3.2. EVALUATION

A ***fitness function*** is used to evaluate each of the candidates in the current population. There are several fitness criteria that have been proposed in the literature for the purpose of test case prioritization. In the framework proposed in this work and in [36], we emphasize on using a new metric, **$APBC_m$ (modified APBC)** as explained in section 4.2, as they used the knowledge of modifications unlike **APBC** metric. This metric is a modified form of traditional APBC (Average Percentage of Block Coverage) metric which is evaluated as follows:

$$APBC = 1 - \frac{TB1 + TB2 \ldots + TBn}{nm} + \frac{1}{2n} \quad (3.1)$$

where "n" is the number of test cases in the input test case set and "m" is the number of blocks in the program to be tested. TB_j denots the location, in the test case sequence, of the test case that first finds the block 'j'. Consider the test case sequence A,C,E,D,B. Suppose

block 3 (in program code) is covered by 2 test cases{E,B},then the value of TB_3 =3(location of 'E' in the test case sequence under consideration). APBC tries to achieve the block coverage at a faster rate.

3.3. SELECTION

Few candidates are selected on the basis of their fitness function. These are propagated to the next generation intact or their offsprings, generated after breeding, are propagated. Several methods have been proposed for selecting the good candidates. We have used the ***Roulette Wheel Strategy.*** The roulette-wheel selection, is a genetic operator used in genetic algorithms for selecting potentially fit candidates for mutation If f_i is the fitness of ***ith*** individual in the population, its probability of being selected is $p_i = \frac{fi}{\sum_{j=1}^{j=N} fj}$ where **N** is the number of individuals in the population. This could be imagined similar to a Roulette wheel in a casino. The figure 3.2 gives an overview of the roulette wheel.

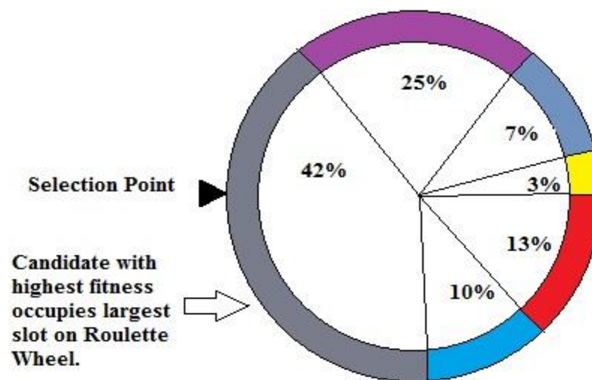

Figure 3.2 Roulette Wheel Selection Strategy

The candidate that is more fit occupies a larger area of the wheel so that it's chances of getting selected is higher. In order to select N candidates, the wheel is rotated N times.

3.4. CROSSOVER

Crossover is a breeding process. A crossover operator is used to recombine two individuals to get a better string. In crossover operation, recombination process creates different individuals in the successive generations by combining material from two individuals of the previous generation. In reproduction, good individuals in a population are probabilistic-ally assigned a larger number of copies and a mating pool is formed. It is important to note that no new individuals are formed in the reproduction phase. In the crossover operator, new individuals are created by exchanging information among individuals of the mating pool.

The two individuals participating in the crossover operation are known as parent individuals and the resulting individuals are known as offsprings. It is intuitive from this construction that good attributes (test cases in this case) from parent can be combined to form a better child string, if an appropriate site is chosen. With a random site, the children produced may or may not have a combination of good features from parent individuals, depending on whether or not the crossing site falls in the appropriate place. But this is not a matter of serious concern, because if good individuals are created by crossover, there will be more copies of them in the next mating pool generated by crossover. It is clear from this discussion that the effect of cross over may be detrimental or beneficial. Thus, in order to preserve some of the good individuals that are already present in the mating pool,

all individuals in the mating pool are not used in crossover. Whether parents selected will be used in crossover operation or not depends on Crossover Probability $\mathbf{P_c}$.

The process of parent individual m, n cross-generation offspring of individual p, q is as follows:

(1) Generate a random crossover point k, k is bigger than 1, less than n (n is the number of test sequences in the test case).

(2) Copy the first k test cases of m into p.

(3) Remove k test cases in n, and then copy the rest into p

(4) similar to generate p , individual q consists of first k test cases in the n, and n-k test cases m which is removed of the k test cases.

3.5. MUTATION

Mutation adds new information in a random way to the genetic search process and ultimately helps to avoid getting trapped at local optima. It is an operator that introduces diversity in the population whenever the population tends to become homogeneous due to repeated use of reproduction and crossover operators. Mutation may cause the chromosomes of individuals to be different from those of their parent individuals.

Mutation in a way is the process of randomly disturbing genetic information. They operate at the feature level (test cases in this case); when the features are being copied from the current individual to the new individual, there is probability that each feature may become mutated. This probability is usually a quite small value, called as mutation probability. A

coin toss mechanism is employed; if random number between zero and one is less than the mutation probability, then the bit is inverted, so that zero becomes one and one becomes zero. This helps in introducing a bit of diversity to the population by scattering the occasional points. This random scattering would result in a better optima, or even modify a part of genetic code that will be beneficial in later operations. On the other hand, it might produce a weak individual that will never be selected for further operations.The mutation is also used to maintain diversity in the population.

Mutation operation process is as follows:

(1) Generates a random between 0 and 1, if the random number is less than mutation probability Pm, then do the mutation operation.

(2) Randomly select two test cases in the test sequences, and exchange its location.

Whether mutation is performed or not depends on mutation probability $\mathbf{P_m}$**.** The process of Mutation causes two test cases in the offspring produced by crossover to be exchanged. This produces diversity in the population.

CHAPTER 4

PROPOSED FRAMEWORK FOR TEST CASE PRIORITIZATION

In this chapter, we describe the framework proposed by us originally in [36] for prioritizing test cases and illustrate it's working with the help of an example.

4.1. THE FRAMEWORK

The proposed framework includes three major components, the Test Case Prioritizing Tool (based on Genetic Algorithm), Modified APBC Metric ($APBC_m$) and the Additional Modified Lines Of Code Coverage (AMLOC) graph. The initial input comprises of a test suite permutation that serves as initial population to prioritization tool. The tool uses the metric ($APBC_m$) to compare between two permutations and decide which candidate permutation is better and should be carried to next generation of population. The final output of the tool is a prioritized test case sequence that maximizes the $APBC_m$. The Figure 4.1 shown below gives an overview of the proposed framework for **Test Case Prioritization**.

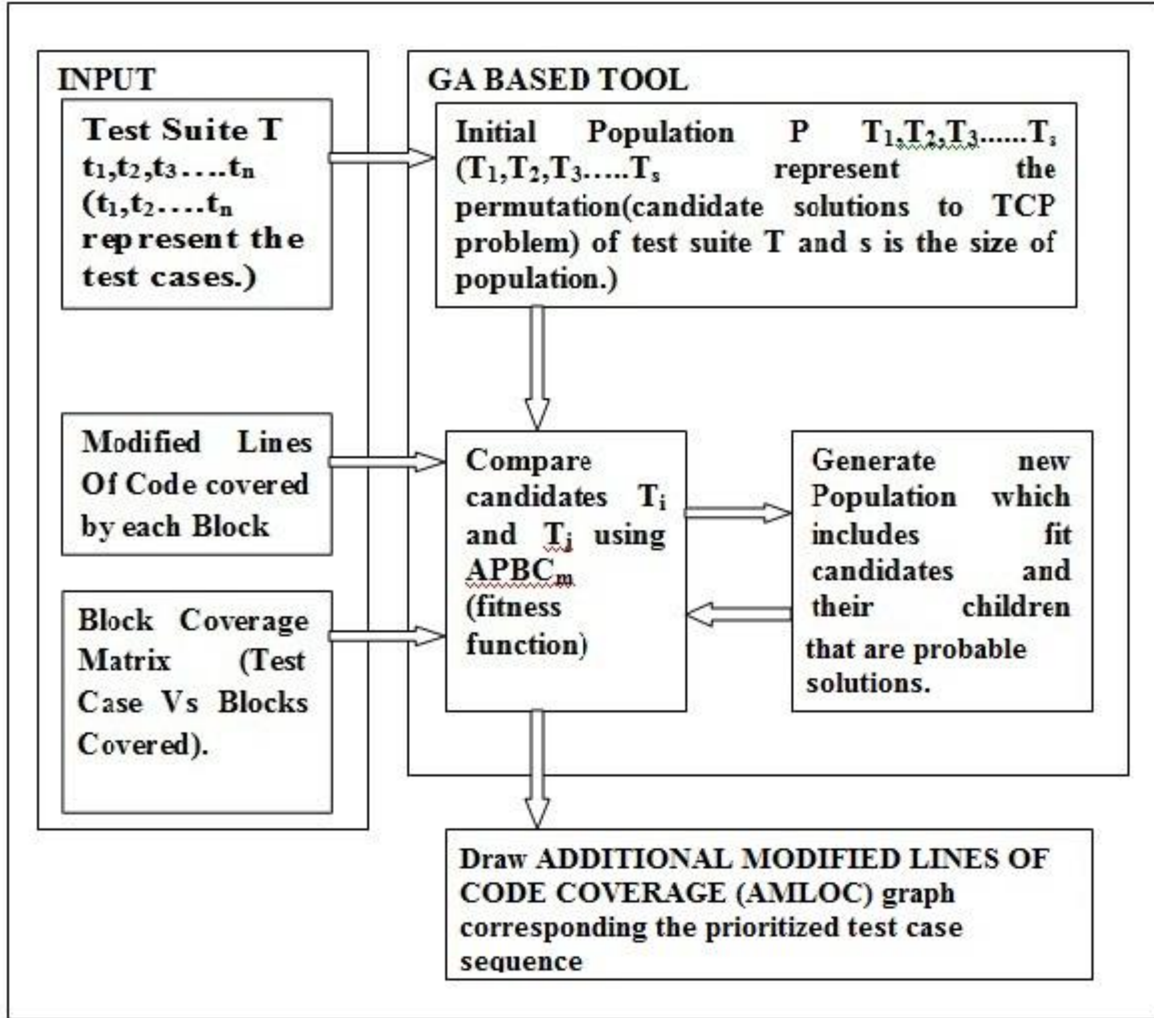

Figure 4.1 Framework For Test Case Prioritization [36]

4.2. MODIFIED APBC METRIC (APBC$_m$)

In this chapter we present a framework for test case prioritization. In this framework we used a Genetic Algorithm Based Tool for prioritizing test cases and used ***modified Average Percentage of Block Coverage*** (**APBC$_m$**) for evaluating the fitness value of an individual in the population, in Genetic Algorithm. This metric is evaluated as follows

$$APBC_m = 1 - \frac{(w_1 * \mathrm{TB_1}) + (w_2 * \mathrm{TB_2}) + \ldots\ldots\ldots + (w_m * \mathrm{TB_m})}{n * (\sum_{i=0}^{m} w_i)} + \frac{1}{2n} \quad \textbf{(4.1)}$$

where “n” is the number of test cases in the input test case set and “m” is the number of basic blocks in the program to be tested. TB_j denotes the location, in the test case sequence, of the test case that first finds the block ‘j’. Consider the test case sequence A,C,E,D,B. Suppose block 3 (in program code) is covered by 2 test cases{E,B},then the value of TB_3 =3(location of ‘E’ in the test case sequence under consideration). APBC tries to achieve the block coverage at a faster rate. However, the APBC metric has the following problem:

a. APBC metric does not consider the practical weight factors like significance of the blocks covered. Several factors that can be used to highlight the significance of blocks are as follows:
    i. Some blocks of code, like exception handling code, are not frequently executed. Certain programming constructs are more error prone than others**[4].** The defect data can be used to find these programming constructs.
    ii. In their paper **[4],** the authors have proposed a method to identify programming and design constructs that contribute more than expected to the defect statistics. **Zengkai Ma et al [32]** have shown how analysis of program structure can be used to find important modules (eg. methods) in a source code. **Lei Zhao et al [33]** have presented a methodology using **Control Flow Analysis** to quantitatively analyze how the basic blocks contribute to failures.
    iii. Some blocks of code contain greater percentage of modified line. Assigning a priority to these blocks shall result in faster exposure of faults since modifications are most likely to contain errors.

For the purpose of illustration we have used the third factor as a measure of weights. We have used this metric as a fitness evaluation function in the Genetic Algorithm based tool and compared the results with those produced by the tool when APBC metric was used.

4.3. ADDITIONAL MODIFIED LINES OF CODE COVERAGE (AMLOC) GRAPH

For a test case sequence TS' $t_1,t_2,t_3,\ldots t_{p-1},t_p\ldots t_n$, the AMLOC value corresponding to a test case t_k, with respect to TS', is the ratio of the total number of unique modified lines of source code that are covered or reached by executing the test cases $t_1,t_2,\ldots t_k$, where $k<=n$, to the total number of modified lines of source code. For example consider the data (hypothetical) given in the following tables. Table 4.1 represents the format of Block Coverage Matrix and Table 4.2 shows the number of modified lines of code contained in each block alongwith the block weights. The block weights are taken as the fraction of modified lines of source code covered by the block.

Table 4.1 Block Coverage Matrix

Blocks / Test Cases	B1	B2	B3	B4	B5
t_1	X		X	X	
t_2		X			X
t_3		X		X	

Table 4.2 Number Of Modified Lines Of Code Covered By Blocks

Blocks	Number of Modified Lines Covered (NMLOC)	Weight of Blocks NMLOC / (∑NMLOC)
B1	2	0.1
B2	6	0.3
B3	5	0.25
B4	2	0.1
B5	5	0.25

Consider the test case sequence TS'' t_1, t_3, t_2. t_1 covers blocks B1,B3,B4 and hence a total of 9 (2+5+2) modified lines of source code. t_3 covers blocks B2 and B4 but B4 has already been executed by test case t_1. So the unique blocks covered by t_1,t_3 together are B1,B2,B3,B4. Hence, total number of unique modified lines of source code covered by t_1,t_3 is 15 (2+6+5+2). Similarly t_2 covers blocks B2 and B5 but B2 has already been covered by t_3. So the unique blocks covered by t1,t3,t2 collectively are B1,B2,B3,B4,B5. Hence, the total number of unique modified lines of source code covered by t_1,t_3,t_2 collectively is 20 (2+6+5+2+5). Therefore the AMLOC values corresponding to test cases t_1, t_3 & t_2 with respect to test case sequence TS'' is 0.45 (9/20) or 45%, 0.75 (15/20) or 75% and 1.0 (20/20) or 100% respectively. The GA based tool uses the AMLOC values of the output test case sequence (prioritized test case sequence) to draw the graph. This graph is then used for comparative analysis of results. Greater the convexity of graph, faster is the rate of modified code coverage. Modified code coverage has a considerable impact on version-specific test case prioritization.

4.4. GENETIC ALGORITHM BASED TOOL

We have presented a framework that employs a test case prioritization tool, developed in Java Language using Eclipse IDE, based on Genetic Algorithm explained in Chapter 3 of this thesis. The tool also uses a new metric ***modified APBC($APBC_m$),*** which is a modified form of original APBC metric. The tool produces the Prioritized Test Case Sequence and an Additional Modified Lines Of Code Coverage (AMLOC) graph for the same. In this section we will explain the working of our framework. The discussion throughout this section

proceeds with an understanding of the way this framework uses the input data to produce a prioritized test case sequence.

The tool takes as input the block coverage matrix and the weight of blocks. The GA algorithm uses $APBC_m$ as fitness function. The initial population for Genetic Algorithm is taken to be the permutations (T1, T2,…..Tn) of the initial test suite T. The fitness value for each of these sequences (candidate solution) is then computed using $APBC_m$. The candidates that are more fit are selected for the breeding process. Let us assume that the selected candidates form the pool **D'**. Whether crossover should be performed on any two parents present in **D'** or not is decided by $\mathbf{P_c}$ (crossover probability)**.** This process is repeated half the times of initial population. The next population (set of test case sequences that will serve as population for next iteration of GA) comprises of the parent candidates if crossover is not performed and children candidates if crossover is performed. After crossover, mutation is performed based on $\mathbf{P_m}$ (mutation probability). This completes one cycle of GA. The convergence criteria used in the tool is the maximum fitness of any individual contained in a population. Consider sample input data (hypothetical) given in Table 4.1 & Table 4.2 to understand the concept underlying the framework.

The Table 4.3 shows that while comparing the individual candidates, GA based tool marked T3 as the fittest candidate when $APBC_m$ was used as fitness function (Experiment 1). It also shows that when APBC was used as fitness function, the GA based tool marked T1 as the fittest candidate (Experiment 2). The ranking produced during evaluation phase has a tremendous effect on the following selection and breeding processes. During the selection

process the candidates are sorted in ascending order of their fitness values and their cumulative $APBC_m$ value is computed (the values are normalized to range [0,1]). A random number 'r' is generated between 0 and 1. The candidate whose cumulative $APBC_m$ value is greater than 'r' is selected. The process is repeated as many number of times as the population size.

Table 4.3 Comparisons of Different Test Case Sequences (Candidates in the TCP problem)

Permutations of Test Suite T	$APBC_m$ Values	$APBC_m$ /∑ $APBC_m$	Ranking on the Basis Of $APBC_m$ (fittest candidate ranked 1)	APBC Values	Ranking on the basis Of APBC (fittest candidate ranked 1)
$T1(t_1, t_2, t_3)$	0.6503(65%)	0.1857	2	0.7(70%)	1
$T2(t_1, t_3, t_2)$	0.567(56.7%)	0.162	3	0.6336(63%)	2
$T3(t_2, t_1, t_3)$	0.6836(68.4%)	0.195	1	0.6336(63%)	2
$T4(t_2, t_3, t_1)$	0.567(56.7%)	0.162	3	0.5(50%)	4
$T5(t_3, t_1, t_2)$	0.517(51.7%)	0.148	4	0.567(56.7%)	3
$T6(t_3, t_2, t_1)$	0.517(51.7%)	0.148	5	0.5(50%)	4

The Table 4.4 below shows the results after sorting the candidates and computing the cumulative $APBC_m$ values for each candidate. Thereafter, the candidates are plotted on a roulette wheel as shown in figure 4.2.

Table 4.4 Cumulative, Normalized $APBC_m$ Values for Sample Data

Candidate Number	Permutations of Test Suite T sorted in Ascending order of $APBC_m$ values	Cumulative Normalized $APBC_m$ values
1	T5	0.148
2	T6	0.296
3	T2	0.458
4	T4	0.62
5	T1	0.807
6	T3	1.0

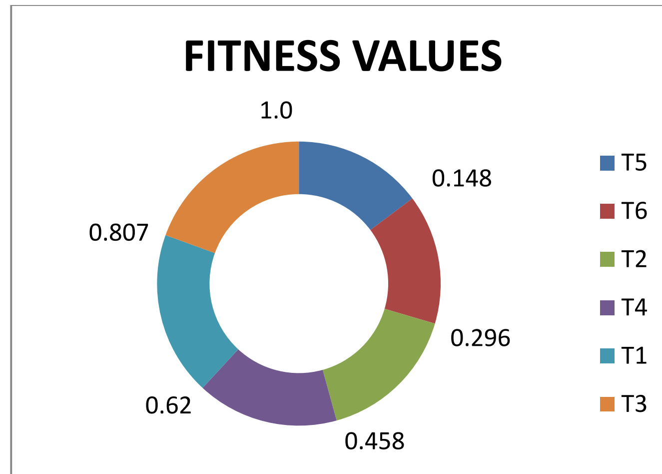

Figure 4.2 Roulette Wheel for Sample Data

The results of selection process have been shown in the Table 4.5.

Table 4.5 Candidates Selected during some iteration (for sample data)

Rounds	Random number Generated	Candidate Selected
1	0.96512	T3
2	0.73894	T1
3	0.869	T3
4	0.826	T3
5	0.779	T1
6	0.2547	T6

This intermediate population comprising of selected candidates is used for crossover. During crossover phase, a random number ‘p’ is generated between 0 and 1. If $p<=P_c$, crossover is performed on two randomly selected different candidates and the resulting off-springs are carried to the next generation or else the candidates are copied in their original form to the next generation. This process is repeated as many times as half the population size. To preserve diversity in population mutation is performed on the population generated after the crossover phase. A random number ‘q’ is generated between 0 and 1. If $q<=P_m$, mutation is performed on a randomly selected candidate. This completes one cycle of GA.

The figure 4.3 and figure 4.4 shows the AMLOC graph for the candidates T1 and T3. These candidates were marked as fittest by the GA based tool when APBC and $APBC_m$ was used as fitness function respectively. The figures show that the candidate ranked as fittest, by GA based tool (when the $APBC_m$ metric is used), is actually the fittest candidate because execution of the test case sequence represented by this candidate (T3) has a greater rate of modified code coverage as shown by AMLOC graph. In other words, the ranking made by the $APBC_m$ is more efficient as compared to the ranking produced by the APBC metric.

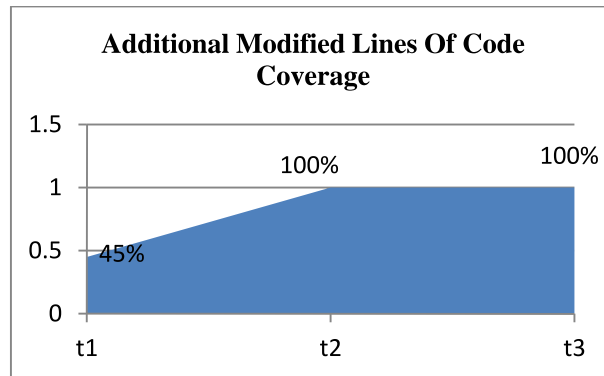

Figure 4.3 AMLOC Graph for T1(Ranked as most fit if APBC is Used for comparison of candidates in GA)

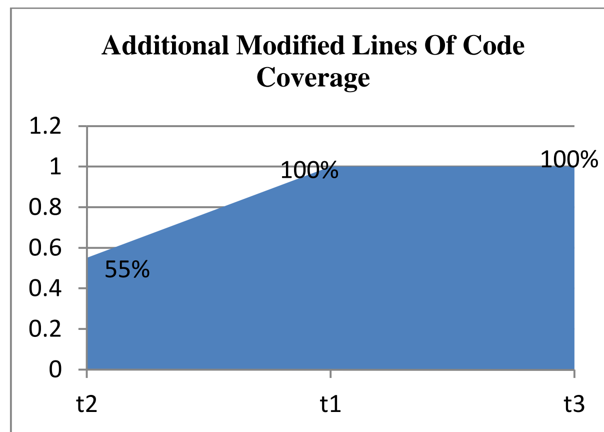

Figure 4.4 Graph for T3(Ranked as most fit if $APBC_m$ is Used for comparison of candidates in GA)

In the upcoming section we demonstrate the working of the proposed framework using a benchmark program. We also present the data used for our demonstration and analyze the results after applying the framework.

4.5. ILLUSTRATION

The ***Triangle*** benchmark program has been used previously by several researchers in Software Engineering studies. Given three sides of a triangle, the program aims to classify it as a scalene, isosceles, equilateral or not a triangle. For the purpose of our illustration, we used the simplified version of original ***Triangle*** program presented in **[34],** translated from FORTRAN into 'C' language. The original set of 14 test cases was minimized to 10 test cases taking into account redundancy of test cases in terms of block coverage. The Table 4.6 shows the original set of test cases followed by the simplified code of Triangle.

Table 4.6 Original Set of Test Cases for Triangle program [34]

Input	Expected Output
0,0,0	4 (Not a triangle)
1,0,0	4 (Not a triangle)
1,1,0	4 (Not a triangle)
1,1,1	3 (Equilateral triangle)
2,2,1	2 (Isosceles triangle)
1,1,2	4 (Not a triangle)
2,1,2	2 (Isosceles triangle)
1,2,1	4 (Not a triangle)
2,1,1	4 (Not a triangle)
3,2,2	2 (Isosceles triangle)
3,2,1	4 (Not a triangle)
4,3,2	1 (Scalene triangle)
2,3,1	4 (Not a triangle)
2,1,3	4 (Not a triangle)

```
1.  int gettri(int side1, int side2, int side3)
2.  {
3.  int triang ;
4.  if( side1 <= 0 || side2 <= 0 || side3 <= 0){
5.  return 4;
6.  }
7.  triang = 0;
8.  if(side1 == side2){
9.  triang = triang + 1;
10. }
11. if(side1 == side3){
12. triang = triang + 2;
13. }
14. if(side2 == side3){
15. triang = triang + 3;
16. }
17. if(triang == 0){
18. if(side1 + side2 <= side3 ||side2 + side3 <= side1 || side1 + side3 <= side2){
19. return 4;
20. }
21. else {
22. return 1;
23. }
24. }
25. if(triang > 3){
26. return 3;
27. }
28. else if ( triang == 1 && side1 + side2 > side3) {
29. return 2;
30. }
31. else if (triang == 2 && side1 + side3 > side2){
32. return 2;
33. }
34. else if (triang == 3 && side2 + side3 > side1){
35. return 2;
36. }
37. return 4;
38. }
```

Figure 4.5 Source Code of Triangle Program [34]

We applied the approach of basic blocks identification algorithm [35] to identify the basic blocks from the source code given in figure 4.5. This algorithm is developed to work on three address code but for simplicity we have used this approach in our high level source code. In this algorithm the code is partitioned in such a way that each line of code falls in exactly one partition. Each partition has exactly one leader which is the first statement of the block. The following rules are used to identify the leaders.

a. The first instruction is a leader.

b. The statements to which control can be transferred from a conditional or unconditional jump statement is a leader.

c. The line immediately following the conditional or unconditional jump instruction is a leader.

Once the leaders are identified the blocks are constructed. A block consists of the leader and all statements following the leader until and not including the next leader.

In the above code 21 basic blocks were identified after applying the algorithm. The identified blocks and block coverage is given in Table 4.7 and Table 4.8 respectively.

Table 4.7 Lines of Code Comprising individual Blocks

Block	Lines Comprising the Block
B1	1-4
B2	5-6
B3	7-8
B4	9-10
B5	11
B6	12-13
B7	14
B8	15-16
B9	17
B10	18
B11	19-20
B12	21-24
B13	25
B14	26-27
B15	28
B16	29-30
B17	31
B18	32[-33
B19	34
B20	35-36
B21	37-38

Table 4.8 Block coverage

Test Cases	Blocks Covered
T1	B1,B2
T2	B1,B2
T3	B1,B2
T4	B1,B3,B4,B5,B6,B7, B8,B9,B13,B14
T5	B1,B3,B4,B5,B7,B9,B13 B15,B16
T6	B1,B3,B4,B5,B7,B9,B13 B15,B17,B19,B21
T7	B1,B3,B5,B6,B7,B9, B13,B15,B17,B18
T8	B1,B3,B5,B6,B7,B9, B13,B15,B17,B19,21
T9	B1,B3,B5,B7,B8,B9, B13,B15,B17,B19,B21
T10	B1,B3,B5,B7,B8,B9, B13,B15,B17,B19,B20
T11	B1,B3,B5,B7, B9,B10,B11
T12	B1,B3,B5,B7, B9,B10,B12
T13	B1,B3,B5,B7, B9,B10,B11
T14	B1,B3,B5,B7, B9,B10,B11

From the Table 4.8 it can be seen that the test cases T1, T2, T3 and T11, T13, T14 are redundant in terms of block coverage. The minimized test set therefore includes T3,T4,T5,T6,T7,T8,T9,T10,T11,T12 and discards T1,T2,T13,T14. For the purpose of our illustration we took another modified version of ***Triangle.c*** by introducing the following features.

i. If the triangle is isosceles print the height of the triangle.

ii. If the triangle is scalene, print the area, circumradius and inradius of the triangle.

The Figure 4.6 shows the modified version of Triangle.c produced by introducing above two features.

```
{
int triang ;
double area;
if( side1 <= 0 || side2 <= 0 || side3 <= 0){
return 4;
}
triang = 0;
if(side1 == side2){
triang = triang + 1;
}
if(side1 == side3){
triang = triang + 2;
}
if(side2 == side3){
triang = triang + 3;
}
if(triang == 0){
if(side1 + side2 <= side3 ||side2 + side3 <= side1 || side1 + side3 <= side2){
return 4;
}
else {
double s=(double)(side1+side2+side3)/2;
double temp=s*(s-side1)*(s-side2)*(s-side3);
area=sqrt(temp);
printf("area of scalene triangle is %lf\n",area);
printf("circumradius of scalene triangle is %lf\n",(side1*side2*side3)/(4*area));
printf("inradius of scalene triangle is %lf\n",(area/s));
return 1;
}
}
if(triang > 3){
return 3;
}
else if ( triang == 1 && side1 + side2 > side3) {
printf("height of isosceles triangle is %lf\n",pow((pow(side1,2)-pow((double)(side3/2),2)),0.5));
return 2;
}
else if (triang == 2 && side1 + side3 > side2){
printf("height of isosceles triangle is %lf\n",pow((pow(side1,2)-pow((double)(side3/2),2)),0.5));
return 2;
}
else if (triang == 3 && side2+side3>side1){
printf("height of isosceles triangle is %lf\n",pow((pow(side1,2)-pow((double)(side3/2),2)),0.5));
return 2;
}
return 4;
}
```

Figure 4.6 Modified Triangle.c code

The fraction of modified lines contained within each block is taken as the block weight. The idea behind doing so is that greater the number of modifications in a block, greater is the error proneness of the block. Doing so, some blocks were found to have weight 0. In order to overcome this ***"zero weight problem",*** '1' was added to each of the values. The table 4.9 gives the weights calculated for the blocks using the modified Triangle.c.

Table 4.9 Weight of Blocks

Block	**Number of Modified Lines Of Code NMLOC**	**Weight NMLOC / ∑NMLOC**
B1	1+1	0.0645
B2	0+1	0.032
B3	0+1	0.032
B4	0+1	0.032
B5	0+1	0.032
B6	0+1	0.032
B7	0+1	0.032
B8	0+1	0.032
B9	0+1	0.032
B10	0+1	0.032
B11	0+1	0.032
B12	6+1	0.226
B13	0+1	0.032
B14	0+1	0.032
B15	0+1	0.032
B16	1+1	0.0645
B17	0+1	0.032
B18	1+1	0.0645
B19	0+1	0.032
B20	1+1	0.0645
B21	0+1	0.032
Total	31	0.996~1

To illustrate the working, two experiments were performed. ***Experiment 1*** was to produce the prioritized test case sequence ***TS1*** using Table 4.8 (after removing redundant test cases)

as input and **APBC** as fitness function in Genetic Algorithm. ***Experiment 2*** was to produce the prioritized test case sequence ***TS2*** using the same Table 4.8 (after removing redundant test cases) as input and **$\mathbf{APBC_m}$** as the fitness function in Genetic Algorithm. Both Experiments were performed using the Test Case Prioritization Tool based on Genetic Algorithm. The convergence criteria used in the tool is the maximum fitness of any individual contained in a population which was set to 0.78. Results of Experiment 1 and Experiment 2 are compared in the Section 4.6 of this chapter.

4.5.VALIDATION OF RESULTS

The results of prioritization as obtained by tool developed are shown below in Figure 4.7 and Figure 4.8.

Figure 4.7 Less AMLOC values per test case in TS1 produced in Experiment 1

Applying APBC means each block is equally likely to have fault and therefore contributes equally to block coverage. A test case covering 6 blocks out of 10 means 60%. In modified APBC metric, blocks that contain greater number of modified lines or is more error prone contributes more to block coverage as shown below. For example, a test case covering 0.453 (sum of weights of blocks covered by test case) to a total of 0.982 (sum of weights of all blocks) means 46% of block coverage. In other words a block having large weight associated with it contributes larger to block coverage. Covering these blocks should therefore be the first concern.

Figure 4.8 Greater convexity(improved coverage rate) graph using $APBC_m$

The values calculated as per ***modified APBC and APBC*** are as follows:

Table 4.10 Results of Experiment 1

Test Case sequence \ Metric	APBC
TS1(T2',T6',T9',T1',T10',T8',T3',T5',T4',T7')	0.79285

Table 4.11 Results of Experiment 2

Test Case sequence \ Metric	Modified APBC
TS2(T10',T8',T5',T9',T3',T7',T2',T1',T6',T4')	0.80285

From the Table 4.10 and Table 4.11 it is clear that using APBC for comparing candidate solutions may finally produce ineffective prioritized Test Case Sequence as output. TS2 is better as it covers the block with greater number of modified lines first (that is blocks with large weights). Hence, Modified APBC is better and is expected to be of great help in version specific test case prioritization. It may therefore help in revealing the faults earlier.

4.6. DRAWBACKS OF THE FRAMEWORK

The framework presented in this chapter is the original framework for test case prioritization proposed by us. We identified two major problems with this framework and enlist them below:

1. A block in the original Average Percentage of Block Coverage (APBC) metric and modified Average Percentage of Block Coverage ($APBC_m$) metric refers to the basic block, that is a block consisting of all sequential statements such that if first statement

in block gets executed then all the consecutive sequential statements in that block get executed. Since it is not practically feasible to calculate the number of changes at the level of blocks, this metric had limited application to small sized programs and not software systems.

2. Most of the software systems developed today use object oriented approach and the framework discussed in [36] was not applicable to the object oriented systems.

In the next chapter we present an enhanced and modified form of this framework which is the main goal of the work presented in this thesis and analyze its effectiveness in the field of test case prioritization problem. The chapter on empirical data collection illustrates the application of the modified framework on two open source projects.

CHAPTER 5
EMPIRICAL STUDY

In the previous chapter we observed certain shortcomings of the test case prioritization framework. We now move ahead to resolve those problems and present a test case prioritization framework which is indeed an enhancement over the framework presented in chapter 4. This framework can be applied to both structured and object oriented systems. Also, the framework produces a prioritized test case sequence that has an increased rate of fault detection. In the study that follows, we give the details of the modified test case prioritization framework which is our object of analysis and aim to address two research questions stated in the next section.

5.1.RESEARCH QUESTIONS

Like any research work the work presented in this thesis aims to address the following research questions:

RQ1. Is $APBC_m$ a better comparator metric than APBC?

RQ2. Does the proposed framework improve fault detection rate?

5.2.EFFICACY MEASURE

In order to address our research questions, we need certain measure with which we can analyze the effectiveness of using the $APBC_m$ metric in the GA based tool, in the modified framework. We have used Average Percentage of Fault Detection as the criterion for analysis. It is defined as follows:

$$APFD = 1 - \frac{\mathbf{TF_1}) + (\mathbf{TF_2}) + \cdots \ldots \ldots + (\mathbf{TF_m})}{n * m} + \frac{1}{2n} \qquad (5.1)$$

where 'TB_i' represents the location of the test case, in the test case sequence, that first exposes i_{th} fault, 'm' represents the number of faults and 'n' represents the number of test cases. It is the most widely used evaluator metric or comparator metric. It measures the rate at which faults are discovered or revealed by a test case sequence. That is, higher the value of the APFD metric, better is the prioritized sequence in terms of fault detection.

5.3.TEST CASE PRIORITIZATION TECHNIQUE

In this section, we present the modified test case prioritization framework which is our object of study. That is, it is this framework that we will analyze for effectiveness in the test case prioritization problem.

5.3.1. MODIFIED FRAMEWORK FOR TEST CASE PRIORITIZATION: OBJECT OF STUDY

As discussed earlier, two major problems were identified with the framework presented in previous chapter. A block in the original APBC metric and modified APBC metric refers to the basic block, that is, a block consisting of all sequential statements such that if first

statement in the block gets executed then all the consecutive sequential statements in that block get executed. Since it is not practically feasible to calculate the number of changes at the level of blocks, this metric has limited application to small sized programs and not software systems. In order to resolve the two problems, we modified the framework presented in chapter 4 by redefining the interpretation of the metric $APBC_m$ and APBC. We have, therefore, considered a class as a block unit in the modified framework thereby extending its utility to object oriented systems. In this work we aim to analyze the effectiveness of this GA based test case prioritization framework using two open source projects.

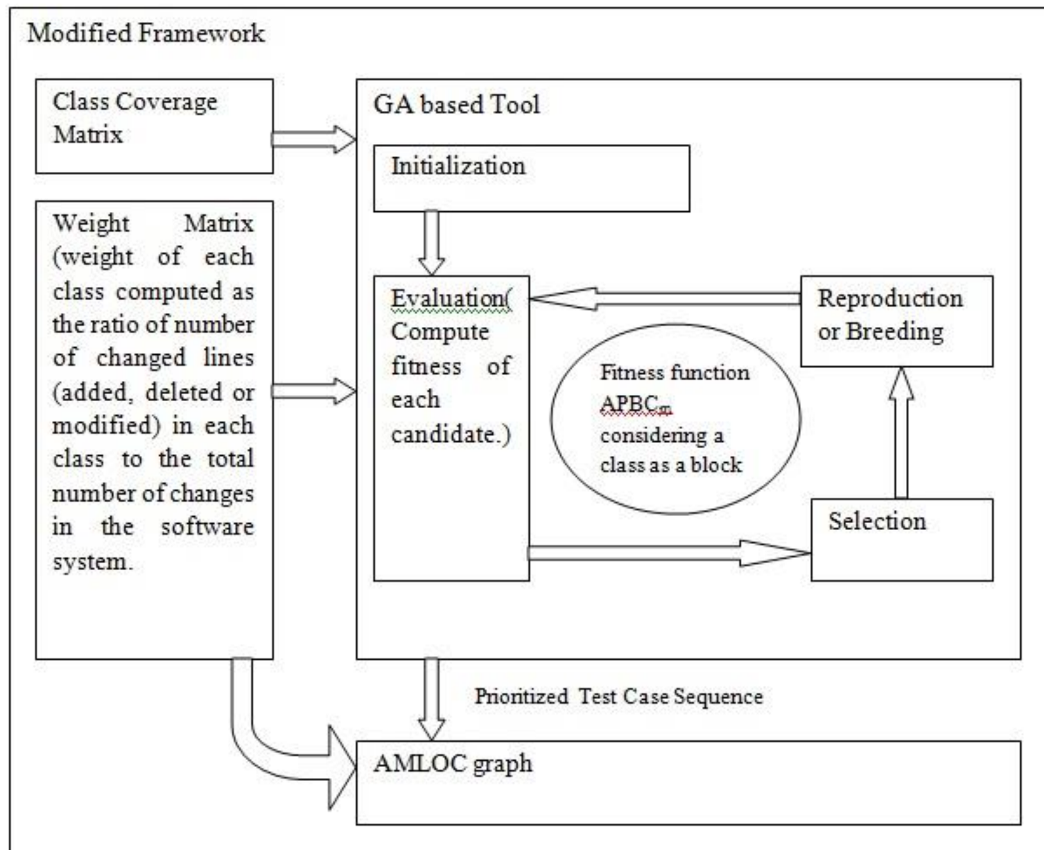

Figure 5.1. Modified Framework for Test Case Prioritization

The modified framework has three major components: the modified $APBC_m$ metric which considers a class as a block, the Additional Modified Lines of Code Coverage graph (AMLOC) and the GA based tool. The GA based tool is the central main component of the framework. Initially the tool takes the weight matrix and the class coverage matrix as the input and produces a prioritized test case sequence and the corresponding AMLOC graph. Figure 5.1 gives an overview of the modified framework.

5.3.2. MODIFIED APBC METRIC ($APBC_m$)

The $APBC_m$ metric defined by Malhotra et al. is given below [1].

$$APBC_m = 1 - \frac{(w_1 * TB_1) + (w_2 * TB_2) + \dots\dots\dots + (w_m * TB_m)}{n * (\sum_{i=0}^{m} w_i)} + \frac{1}{2n} \qquad (5.2)$$

The interpretation of the above equation has been redefined in the modified framework as follows: 'w_i' represents the weight of the 'i_{th}' class, 'TB_i' represents the location of the test case, in the test case sequence, that first covers i_{th} class, 'm' represents the number of classes and 'n' represents the number of test cases. It should be noted that “block” refers is a generic term and can be used to dente a basic block, a function, a class, a module. The GA based tool uses the weight matrix and the class coverage matrix to compute the value of $APBC_m$ for each of the candidate present in a population. Higher value of $APBC_m$ denotes higher fitness level of a candidate in the population. In this work we shall analyze how the concept of weighing the different regions of code differently, employed in this metric, affects the working of GA based tool and rate of fault detection of the prioritized test case sequence produced by the tool.

5.3.3. GENETIC ALGORITHM BASED TOOL

We have developed a GA based tool in java using Eclipse IDE which uses the modified interpretation of the $APBC_m$ metric as discussed in the section 3.3.2. The tool works as follows:

Algorithm

Inputs: Weight matrix, Class Coverage Matrix

Output: A prioritized Test Case Sequence

Begin

a. **Encoding:** The tool uses permutation encoding to represent the candidate solutions to test case prioritization problem. A candidate encoded using this strategy is represented as a sequence of numbers which denotes a permutation. In our problem of test case prioritization it denotes a permutation of test cases. That is, in a candidate sequence, 'i_{th}' value indicates a test case whose rank is 'i'. For instance, consider a sample sequence (permutation of 5 test cases) "3, 4, 5, 1, 2". In this sample sequence test case '3' has rank '1', test case '4' has rank '2', test case '5' has rank '3' and so on.
b. **Initialization:** The tool takes as input a set of test cases and computes permutations of this set. An initial population of 50 candidates ordered lexicographically is generated as the initial pool of candidates.
c. **Evaluation:** The tool uses the $APBC_m$ metric (defined in section 3.3.2) to compute the fitness of each of the candidate.
d. **Selection:** The tool uses the Roulette Wheel strategy [] to select pairs of candidates and for each selected pair performs step e and step f.

e. **Crossover:** The tool performs the crossover operation based on crossover probability, P_c and generates two children from the parents.

f. **Mutation:** The tool performs the mutation of the two children produced in step 4 on the basis of mutation probability, P_m , and generates two children. These two children become a part of the new population which is used as input to the next iteration.

g. Repeat steps c to step f till the algorithm converges.

End.

CHAPTER 6
DATA COLLECTION AND EXPERIMENT DESIGN

In this chapter we describe the meaning of the different data that will be used throughout the experimentation and also highlight the computational details of each data. Apart from this, we will provide the details of various tools, employed for successful conduction of the experiments, as and when required.

6.1.DATA COLLECTION

In order to assess the test case prioritization framework, we designed and performed two experiments. In Experiment 1, we used the APBC metric as fitness function in GA based tool and obtained a prioritized test case sequence TS1.While in Experiment 2, we used the $APBC_m$ metric as the fitness function in GA based tool and obtained another prioritized test case sequence TS2. To assess the effectiveness and quality of the two test case sequences, we used the additional fault coverage strategy and developed a graph. This section provides a deep insight into how the data, to be experimented, was prepared.

The two open source projects were taken from Souceforge [43] and Software Artifact Infrastructure Repository (SIR) [44]. Sourceforge is a Free Open Source Software host that allows users to build open source software systems. It also provides a common repository of source code and documentation related to software projects. SIR is a repository, specialized and dedicated to research in the field of Software Testing.

6.1.1.SUBJECT PROGRAMS

The data, to be experimented, was obtained from two open source projects JTopas [45] and Xml-Security [46]. The JTopas project provides a small and easy to use java library for tokenizing and parsing arbitrary text data like Html files, Xml files, RTF files, etc. It comprises of 50 classes and around 5400 LOC. XML-security is a component library implementing XML signature and encryption standards. It is supplied by the XML subproject of the open source Apache project and is available at [47]. Currently, it provides a mature implementation of Digital Signatures for XML, along with implementation of encryption standards in progress. It comprises of 143 classes and around 16800 LOC. For the purpose of our validation, we used version v0 and v1 (SIR versions) and considered a class as a block for both the projects. we used CLOC tool [48] and scanned for all the classes present in the version v0 and v1 and counted the number of lines added, deleted and modified in each of the classes. CLOC is freely available, command based software that counts the number of changes (additions, deletions, modifications) made in the source code and generates a report. It supports a variety of report formats like csv, sql, etc. The weight of each class is calculated using the following formula:

$$\mathbf{w_i} = \frac{\mathbf{c_i}}{\sum_{i=1}^{n} \mathbf{c_i}} \qquad (6.1)$$

where ‘n’ denotes the number of classes, ‘c_i’ denotes the number of changes in the ‘i_{th}’ class and ‘w_i’ denotes the weight of the ‘i_{th}’ class. While preparing the weight matrix it was found that some of the classes no changes, that is, number of modified lines of code is ‘0’. In order to overcome this “zero error problem” a minimal count of ‘1’ was added to each of the classes. Table 6.1 gives a short description of the subject programs and table 6.2 lists the weight matrix of the JTopas project.

Table 6.1. Subject Programs

Project	Size (LOC)	Number of Classes	Number of Sequential Versions	Number of Test Cases	Number of Faults
JTopas	5400	50	4	10	10
Xml-Security	16800	143	9	15	20

Table 6.2.Weight Matrix for JTopas Project

Class	NMLOC(Number of Modified LOC)	Weights
C1	33	0.194
C2	27	0.158
C3	23	0.135
C4	23	0.135
C5	23	0.135
C6	1	0.005
C7	1	0.005
C8	1	0.005
C9	15	0.088
C10	1	0.005
C11	1	0.005
C12	1	0.005
C13	1	0.005
C14	18	0.1
C15	1	0.005
	170 (total)	~1

6.1.2. TEST SUITE

In order to conduct our experiments, we required the entire test suite of the JTopas and Xml Security project, version v1. The Software Artifact Infrastructure Repository (SIR) provides most of the information necessary for research in the field of testing like, test cases, regression faults, change logs, etc. The JTopas and Xml-Security project have 10 and 15 JUnit test cases respectively. Apart from this we conducted several activities like obtaining coverage information, creating faulty versions of the original project (version v1), preparation of fault matrix and preparation of class coverage matrix. we used Eclipse Galileo, an IDE to support the development of java projects and EclEmma [49], an eclipse plugin to provide coverage based information. Each of the test cases was executed for both the projects. For each test case execution EclEmma provides a detailed report at different levels of coverage like class, methods, complexity, lines, etc. All the reports were merged manually which required effort in activities like adding the classes which were completely missed by each of the test cases, sorting the entire file and eliminating classes which were not covered by any of the available test cases. The final common class coverage was reported in an excel sheet. Table 6.3 displays the class coverage matrix for JTopas Project.

Table 6.3. Class Coverage Matrix for JTopas Project.

Class	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**	**TC9**	**TC10**
C1	0	0	1	1	1	1	1	1	1	1
C2	1	1	0	0	0	0	0	0	0	0
C3	1	1	0	0	0	0	0	0	0	0
C4	1	0	0	0	0	0	0	0	0	0
C5	1	1	0	0	0	0	0	0	0	0
C6	0	0	0	0	0	0	0	1	1	1

C7	0	0	1	1	1	1	1	0	0	1
C8	0	0	0	0	0	1	0	0	0	0
C9	0	0	0	0	0	0	0	1	1	1
C10	0	0	1	1	1	1	1	0	0	1
C11	0	0	0	0	0	1	0	0	0	0
C12	0	0	1	1	1	1	1	0	0	1
C13	0	0	1	1	1	1	1	1	1	1
C14	1	0	0	0	0	0	0	0	0	0
C15	0	0	1	1	1	1	1	1	1	1

6.1.3. FAULTS

The objective of this work was to analyze the prioritized test case sequences produced by the GA based tool in the two experiments. In order to assess the test case sequences, we used the percentage of regression faults discovered by them. For this we needed regression faults data. Regression faults are the faults introduced in the software system as a result of changes made to the system. SIR provides such faults for the JTopas and Xml-security project. The fault seeding procedure, followed by SIR, is similar to that defined and used in several previous studies in the field of testing techniques [37],[38],[39]. The following types of faults were considered by the SIR seeders [35]:

a. Faults associated with variables: definition of variable, redefinition of variable, deletion of variable, change value of variable in existing assign statement.

b. Faults associated with control flow: addition of new block of code, deletion of path, redefinition of execution condition, removal of block, change order of execution, new

call to external function, removal of call to external function, adding function, removing function.

c. Faults associated with specific Java language constructs or facilities (such as constructors or inheritance).

Given the potential faults, we seeded the faults one at a time and created faulty versions of the existing original version v1 for both the projects. We then executed the entire test suite on each of the faulty versions to visualize which test cases succeeded in revealing which faults. In doing so it was found that there were 3 faults (F4, F7, F8) in JTopas project which could not be revealed by any of the existing test cases in the test suite. While following similar procedure with the Xml-security project we found that there were 13 faults which could not be revealed by any of the available test cases. Apart from this there were two 2 faults which were revealed by more than 25% of the test cases. Since, we were not interested in assessing the effectiveness of the test suite, we simply discarded these faults. The final fault matrices for the JTopas and Xml-Security project are given in Table 6.4 and Table 6.5.

Table 6.4. Fault Matrix of JTopas

Fault	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10
F1	1	0	0	0	0	0	0	0	0	0
F2	1	0	0	0	0	0	0	0	0	0
F3	1	0	0	0	0	0	0	0	0	0
F5	0	0	0	1	1	0	0	0	0	0
F6	0	0	0	1	0	0	0	0	0	0
F9	1	1	0	0	0	0	0	0	0	0
F10	1	1	0	0	0	0	0	0	0	0

Table 6.5. Fault Matrix of Xml-Security

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12	TC13	TC14	TC15
F1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
F2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
F3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
F4	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
F5	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0

6.2.EXPERIMENT DESIGN

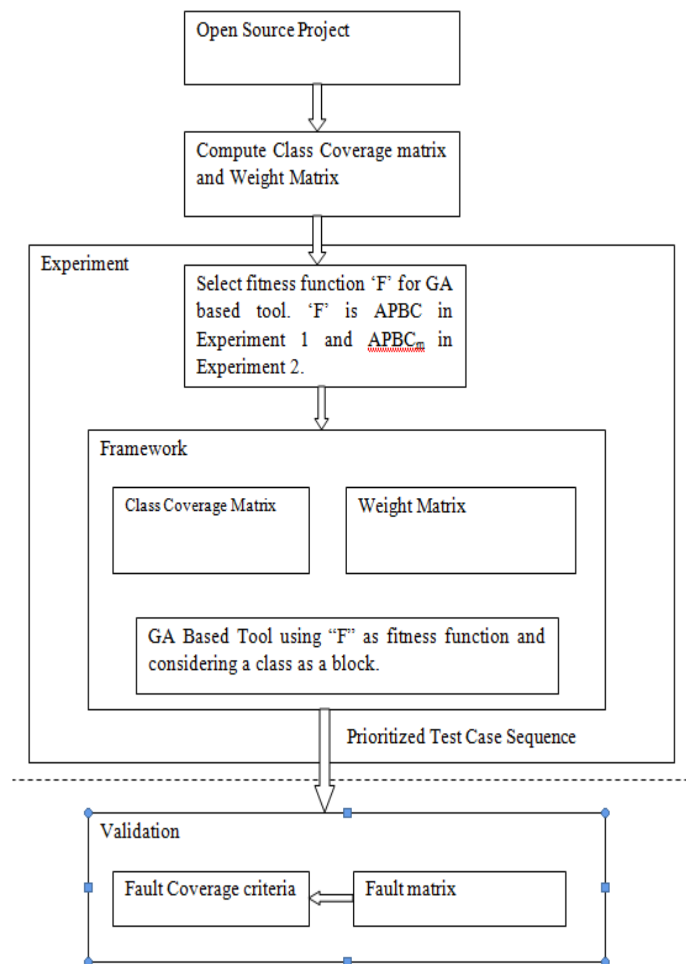

Figure 6.1. Experiment Design or Process

In order to be systematic in our validation process, we designed a strategy for conducting the experiments. Figure 6.1 gives an overall description of the complete experimental process including the modified framework presented in figure 5.1, various inputs to the framework, outcome of applying the framework and the basis or criteria for validation. The experiments are designed in such a way that our three research questions can be addressed using these two experiments.

We selected two open source projects to validate the modified framework. For each project we computed the weight matrix and the class coverage matrix and performed two experiments as shown in figure 6.1 using the weight matrix and the class coverage matrix of the selected project. In the first experiment APBC was chosen as the fitness function in the GA based tool and in the second experiment $APBC_m$ was chosen as the fitness function. Apart from this, as discussed earlier, for both the experiments, a class is considered as a block. For both the experiments the convergence criteria for GA based tool is chosen to be the maximum fitness of any candidate in the population. It was set to be 0.96 for Xml-Security project and 0.913 for JTopas project. The reason behind choosing different values in different projects is that the two projects differ significantly in project size and other characteristics. we also computed the fault matrix using the faults data available with the project and analyzed the results of the experiments (prioritized test case sequence) using the APFD metric. For this an additional fault coverage graph for the prioritized sequence was plotted using the fault matrix The APFD value gives the area under the curve plotted. For details on APFD metric refer to chapter 5.

CHAPTER 7
RESULT ANALYSIS

In this chapter we present an analysis of the results produced by the modified and enhanced test case prioritization framework, presented in chapter 6, when applied on the two open source projects Jtopas and Xml-Security and provide answers to the research questions stated in chapter 5.

7.1. (RQ1) Is $APBC_m$ a better comparator metric than APBC?

To address this research question, it is important to understand the role played (role of fitness function) by the $APBC_m$ metric (fitness function in experiment2) or APBC metric (fitness function in experiment 1) inside the GA based tool. The fitness function in the GA based tool is used for comparison of various candidate solutions (test case sequences) and selection of parents for reproduction. A good and healthy comparison of candidates from the population, during selection phase in GA, results in good parents being selected. This ensures that the future generation has fitter candidates than are present in the current generation.

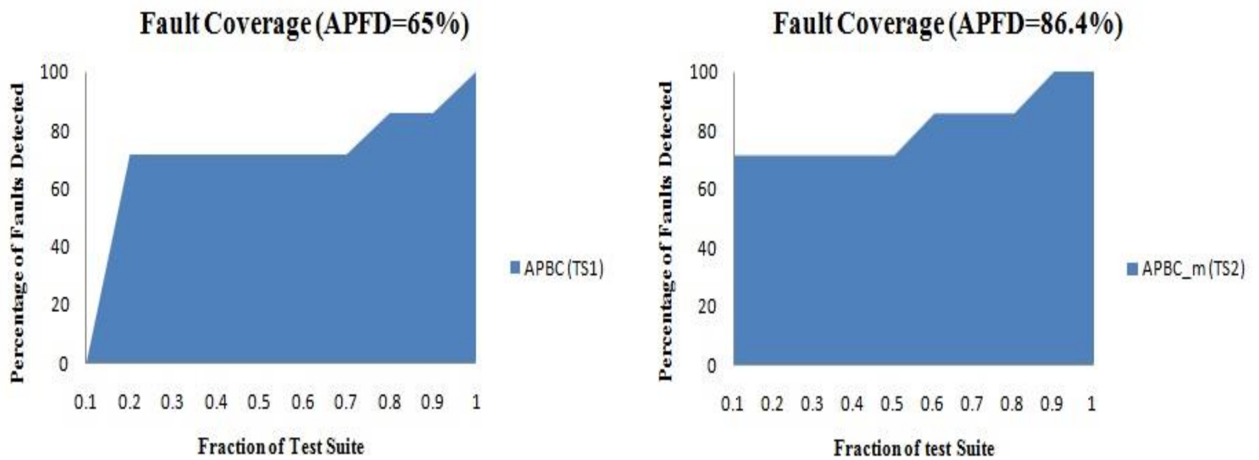

Figure 7.1. Results of JTopas Project

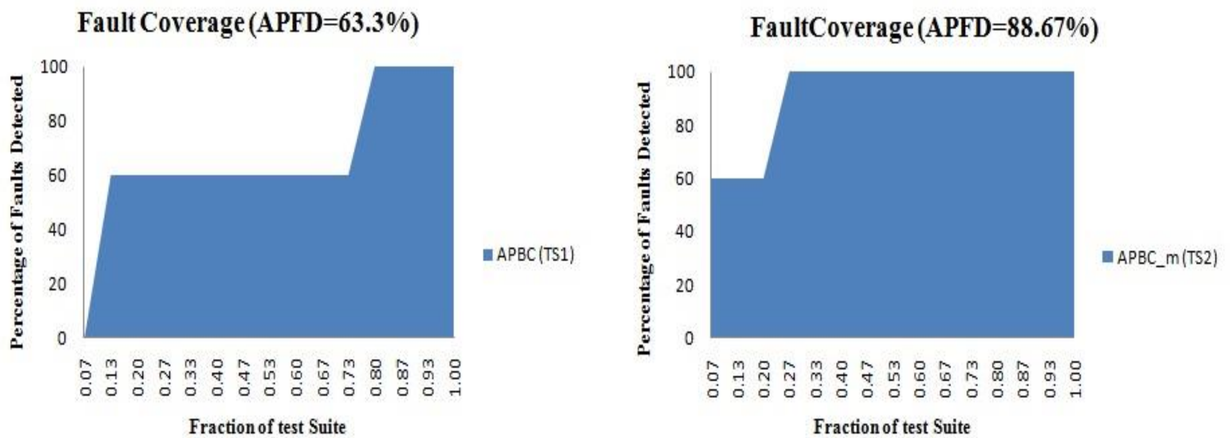

Figure 7.2. Results of Xml-Security Project

Table 7.1. Results of JTopas project

	Test Sequence	**APFD Value**
Experiment 1 (Using APBC)	TC6, TC1, TC9, TC2, TC8, TC3, TC7, TC5, TC10, TC4	65%
Experiment 2 (Using $APBC_m$)	TC1, TC9, TC6, TC4, TC2, TC10, TC5, TC8, TC3, TC7	86.4%

Table 7.2. Results of Xml-Security Project

	Test Sequence	APFD Value
Experiment 1 (Using APBC)	TC2, TC8, TC10, TC7, TC5, TC9, TC15, TC1, TC14, TC4,TC3,TC6,TC11,TC13,TC12	63.3%
Experiment 2 (Using $APBC_m$)	TC8, TC2, TC10, TC6, TC11, TC7, TC1, TC4, TC14, TC3,TC5,TC12,TC15,TC13,TC9	88.67%

The two sequences TS1 and TS2 produced by the GA based tool, in the two experiments, are given in table 7.1 and table 7.2 for the JTopas and Xml-Security projects respectively. Figure 7.1 and figure 7.2 show the fault coverage graph for the two projects. Since the APFD value of the sequence TS2, in both the projects, is greater than APFD value of the sequence TS1, it is clear that the tool performs better with the $APBC_m$ metric. It can therefore be deduced that the $APBC_m$ metric is a better comparator metric than APBC.

7.2. (RQ2) Can the proposed framework improve the rate of fault detection?

In order to address this research question we plotted the fault coverage graph given in figure 7.1 and figure 7.2. The vertical axis shows the percentage of total regression faults revealed where as the 'i_{th}' value on horizontal axis shows the test suite fraction that has been executed. A value (x,y) in the graph denotes that 'y' percent of total regression faults are revealed after 'x' fraction of the test suite, in the order specified by the test case sequence, is executed.

The APFD value represents the area of the graph shown. Looking at the graphs, it is observed that the APFD values show a remarkable increase of 21% and 25% in the second

experiment for the JTopas and Xml-Security project respectively. Figure 7.3 and figure 7.4 shows this increase in fault coverage per test case execution for the two projects. This implies that the modified framework for test case prioritization increases the rate of fault detection of the regression test suite. It can therefore be used to generate a test case sequence with a high potential of exposing the faults earlier during regression testing.

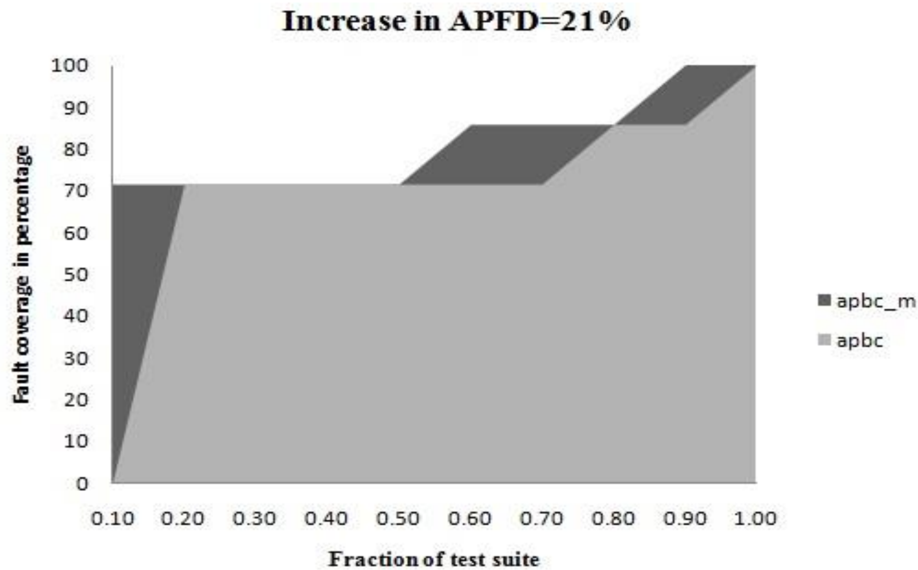

Figure 7.3. Increase in Fault Deatection Rate for Jtopas Project.

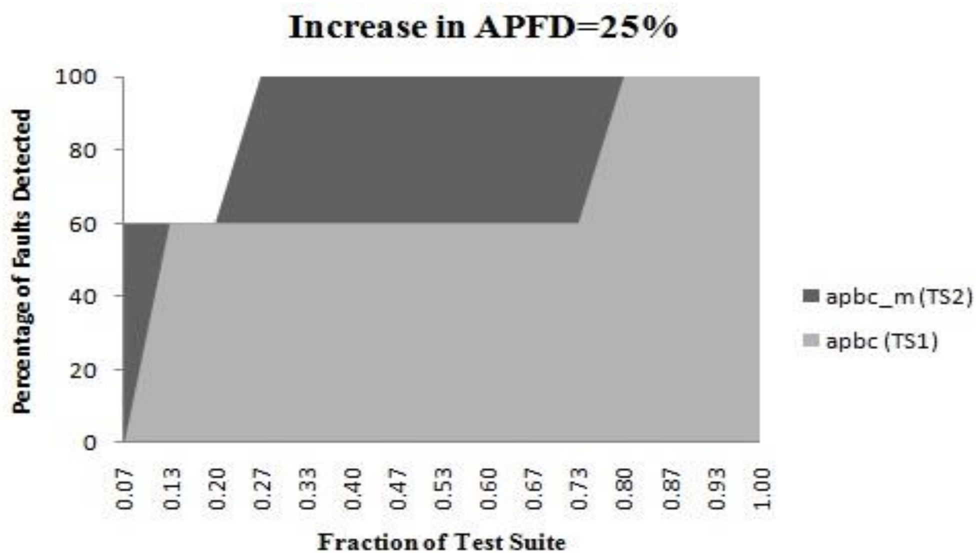

Figure 7.4. Increase in Fault Detection Rate for Xml-Security Project.

7.3. DISCUSSIONS

Software maintenance is a rigorous activity requiring a huge amount of resources, time and effort. Often during maintenance and regression testing, testers are short of time and resources. In such a scenario, smart testing and not hard testing is the key objective of testers. Using the concept of weights to highlight the error prone and significantly modified code from less error prone regions allows testers to divert their time and effort in testing these parts rather than testing the entire software system with equal focus on each and every portion of code. The concept of assigning weights also highlights the significance of the changed code regions. There is no point in extensively testing a change in a feature or functionality that is known to be less than 5% usable. Errors and bugs are less likely to be reported for such a feature as these features are hardly used. The question now arises that “Is it worth testing such a change?” when it is understood that a fault or a bug in such a feature will probably be reported after several years. The new and enhanced test case prioritization framework presented in chapter 5 allows testers to perfectly manage their time, effort and resources by ordering test cases in a smart way so that highly error prone regions and significant changes are tested first and faults are revealed earlier during testing.

The major advantage of the modified framework are as follows:

1. It is applicable to object oriented systems as well as structured systems (considering a function as a block).

2. The concept of weighing also helps testers to identify weak portions and holes in the code. Such regions of code are more likely to contain faults. Testing such regions of the code prior to others exposes the faults in the software system early in regression testing. It helps testrs to understand **“what to test”** and **“how much to test”**.

3. Choice of class as a block unit is apt because it is neither too small like a basic block or a statement, nor it is too large like a complete module or a component. Small size of a block makes the framework difficult to apply and large block size makes it difficult to localize the faults exposed.

CHAPTER 8
CONCLUSIONS

Regression testing is a complex and costly process that may involve multiple objectives and constraints. For example, the cost of executing a test case is usually measured as the time taken to execute the test case. However, there may be a series of different costs involved in executing a test case, such as setting up the environment or preparing a test input, each of which may be subject to a different constraint. Existing techniques also assume that test cases can be executed in any given order without any change to the cost of execution, which seems unrealistic. Test cases may have dependency relations between them. It may also be possible to lower the cost of execution by grouping test cases that share the same test environment, thereby saving set-up time.

Considering the complexity of real-world regression testing, existing representations of problems in regression testing may be over-simplistic. Larger software systems do not simply entail larger problem size; they may denote a different level of complexity. Test Case Prioritization is an essential task that tries to reduce the testing effort in maintenance phase to a considerable extent. In this thesis we have described the original test case prioritization

framework proposed by us in [36]. we have, thereby, highlighted the major drawbacks of this framework and propose a new enhanced and modified framework which is applicable to object oriented systems. The framework uses a tool based on Genetic Algorithm, developed in Java. The framework also highlights the necessity and the benefits of using a new metric $APBC_m$ (considering a class as a block unit in the metric) as fitness evaluation function in GA. Several factors that can be used to embed the knowledge about significance of blocks in the $APBC_m$ metric have also been discussed. However, the exact computation of weights taking into account all the factors discussed, is still an open challenge. Finally, the results have been analyzed and compared, on the basis of fault coverage criteria using APFD metric, with those produced when traditional APBC metric was used as fitness evaluation function in GA based tool. It was then found that $APBC_m$ metric is better and efficient than APBC. The approach, presented here, has its application in the areas of version specific test case prioritization but can also be extended to generalized test case prioritization problem. Considering practical weight factors is a general concept that can help improve cost of regression testing and can also be extended to the problems of test suite minimization and regression test selection.

This work provides a detailed analysis of incorporating weights in test case prioritization problem, with the aim of improving the quality of regression testing. This can be used to reduce the costs incurred and time elapsed in regression testing or software testing in general. The literature survey is an evidence to suggest that the topic of test case prioritization is of increasing importance. The field continues to attract growing attention from the wider research community.

The main contributions of the work are as follows:

a. In this thesis we propose a test case prioritization framework that is applicable to object oriented systems. The framework uses the GA based tool and $APBC_m$ metric as fitness function in the tool.

b. We applied the framework on two open source projects JTopas and Xml-Security.

c. We compared and analyzed the prioritized test case sequences produced by the framework using the APFD metric and show that the framework has indeed improved the rate of fault detection by 21% and 25% for JTopas and Xml-Security projects respectively.

d. We also compared the APBC and $APBC_m$ metric and show that the latter is a better comparator metric and yields better results when used as fitness function in GA based tool.

e. We have presented an empirical validation of the proposed framework which shows that the proposed framework produces a prioritized test case sequence with high potential of exposing faults early during regression testing.

Hence, we conclude that the proposed framework can be used by software practitioners and researchers for obtaining prioritized test case sequence during version specific regression testing.

There are a few areas which still need to be looked upon in future. In this work, we have computed the weights using the knowledge of modifications in the code. There are several other factors also that can be used along with this knowledge to highlight the error prone regions of code such as complexity, coupling between classes, etc. Apart from this, the

framework can be extended to general test case prioritization problem where testers can focus on testing the highly usable features rather than testing each feature with equal time and effort. In such case, portion of code, implementing these features, is significant and should be assigned a higher weight. In future we aim to address above mentioned challenges.

REFERENCES

1. Aggarwal, K.K, Singh, Y., “Software Engineering”, New Age International Publishers, Second ed., 2006.
2. Kaner, C., “Exploratory Testing,” Quality Assurance Institute Worldwide Annual Software Testing Conference Florida Institute of Technology, Orlando, FL, 2006
3. Rothermel, G., Untch, R.J., Chu, C., Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering October 2001,Vol 27 No. 10 ,p 929–948.
4. Elbaum, S., Rothermel, G., Kanduri, S., Malsihevsky, A.G., “Selecting A Cost Effective Test Case Prioritization Technique”.Software Quality Control Journal, 2004, Vol 12 No. 3, p 185-210.
5. Claes, W., Martin, H., Magnus, C. Ohlsson, “Understanding The Sources Of Software Defects: A Filtering Approach”, Proceedings the 8th International Workshop on Program Comprehension, Limerick, Ireland, 2000.
6. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., Test case prioritization: An empirical study. Proceedings of International Conference on Software Maintenance (ICSM 1999), IEEE Computer Society Press, 1999,p 179–188.
7. Elbaum, S.G., Malishevsky, A.G., Rothermel, G., Prioritizing test cases for regression testing. Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2000), ACM Press, 2000; p 102–112.
8. Elbaum, S., Gable, D., Rothermel, G., Understanding and measuring the sources of variation in the prioritization of regression test suites. Proceedings of the Seventh International

Software Metrics Symposium (METRICS 2001), IEEE Computer Society Press, 2001, p 169–179.

9. Elbaum, S.G., Malishevsky, A.G., Rothermel, G., Incorporating varying test costs and fault severities into test case prioritization in *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, ACM Press, 2001,p 329–338.
10. Malishevsky, A., Rothermel, G., Elbaum, S. “Modeling the cost-benefits tradeoffs for regression testing techniques” in *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society Press, 2002, p 230–240.
11. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B. “The impact of test suite granularity on the costeffectiveness of regression testing.” in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, ACM Press, 2002, p 130–140.
12. Jones, J.A., Harrold, M.J. “Test-suite reduction and prioritization for modified condition/decision coverage” in *Proceedings of International Conference on Software Maintenance (ICSM 2001),* IEEE Computer Society Press, 2001, p 92–101.
13. Leon, D., Podgurski, A. “A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases” in Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), IEEE Computer Society Press, 2003, p. 442–456.
14. Carlson, R., Do, H., Denton, A. “A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study” in *27th IEEE International Conference on Software Maintenance (ICSM),* 2011.

15. Tonella, P., Avesani, P., Susi, A. “Using the case-based ranking methodology for test case prioritization” in *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 2006),* IEEE Computer Society, 2006, 123–133.

16. Yoo, S., Harman, M., Tonella, P., Susi, A. “Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge” in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, ACM Press, 2009, p 201–211.

17. Kim, J.M., Porter, A. “A history-based test prioritization technique for regression testing in resource constrained environments” in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002),* ACM Press, 2002,p 119–129.

18. Mirarab, S., Tahvildari, L. “A prioritization approach for software test cases based on bayesian networks” in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering,* Springer–Verlag, 2007, p 276–290.

19. Korel, B., Tahat, L., Harman, M. “Test prioritization using system models” in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005),* 2005, p 559–568.

20. Korel, B., Koutsogiannakis, G., Tahat, L.H. “Model-based test prioritization heuristic methods and their evaluation” in *Proceedings of the 3rd international workshop on Advances in Model-based Testing (A-MOST 2007)*, ACM Press, 2007, p 34–43.

21. Korel, B., Koutsogiannakis, G., Tahat, L.. “Application of system models in regression test suite prioritization” in *Proceedings of IEEE International Conference on Software Maintenance 2008 (ICSM 2008),* IEEE Computer Society Press, 2008, p 247–256.

22. Panigrahi, Chhabi, R., Mall, R. “Model Based Regression Test Case Prioritization”. ACM SIGSOFT Software Engineering Notes,2001, Vol 35 No 6, p 1-7

23. Roongruangsuwan, S., Daengdej, J., “A Test Case Prioritization Maethod with Practical Weight Factors”, Journal Of Software Engineering, 2010, Vol 4 No. 3, p 193-214.
24. Yoo, S., Harman, M. Pareto, “Efficient multi-objective test case selection” in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007),* ACM Press, 2007, p 140–150.
25. Do, H., Mirarab, S.M., Tahvildari, L., Rothermel G., “An empirical study of the effect of time constraints on the cost-benefits of regression testing” in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, 2008, p 71–82.
26. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.. “Time aware test suite prioritization” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, ACM Press, 2006, p 1–12.
27. Zhang, X., Qu, B., “An Improved Metric for Test Case Prioritization” in *Eighth Web Information Systems and Applications Conference,* IEEE, 2011.
28. Johnson, S.M. (1963), "Generation of permutations by adjacent transposition", Mathematics of Computation 17, p 282–285, doi:10.1090/S0025-5718-1963-0159764-2.
29. Trotter, H. F. (August 1962), "Algorithm 115: Perm", Communications of the ACM , Vol 5 No. 8, p 434–435.
30. Levitin, A. “Introduction to The Design & Analysis of Algorithms”, Addison Wesley, 2003.
31. Thengade, A., Dondal, R., “Genetic Algorithm – Survey Paper”**.** MPGI National Multi Conference 2012 (MPGINMC-2012), *Proceedings published by International Journal of Computer Applications.*

32. Ma, Z., Zhao, J. "Test Case Prioritization based on the Analysis Of Program Structure" in *15th Asia-Pacific Software Engineering Conference, 2008. APSEC '08.*,IEEE, p 471-478.

33. Zhao, L., Wang, L., Yin, X. "Context Aware Fault Localization via Control Flow Analysis". Journal Of Software,2011, Vol 6, No. 10.

34. Langdon, W.B., Harman, M., Jia, Y. "Efficient multi-objective higher order mutation testing with genetic programming". Journal Of Systems and Software, 2010, ACM, Vol 83 No. 12, p 2416-2430.

35. Allen, F.E. "Control Flow Analysis" in *Proceedings of Symposium on Compiler Optimization July 1970,* ACM-SIGPLAN, Vol 5 No. 7, p 1-19

36. Malhotra, R., Tiwari, D. "Development of a framework for Test Case Prioritization using Genetic Algorithm"

37. Foster, H., Goradia, T., Hutchins, M., Ostrand, T. "Experiments on the effectiveness of data flow and control flow test adequacy criteria" in *Proceedings of the International Conference on Software Engineering,* (May 1994), 191-200.

38. Horgan, J.R., London, S., Mathur, A.P., Wong, W.E. "Effect of test set size and block coverage on the fault detection effectiveness" in *Proceedings of the Fifth International Symposium on Software Reliability Engineering,* (November 1994), 230-238.

39. Horgan, J.R., London, S., Mathur, A.P., Wong, W.E. "Effect of test set minimization on the fault detection effectiveness" in *Proceedings of the 17th International Conference on Software Engineering,* (April 1995), 41-50.

40. Avritzer, A., Weyuker, E.J. "The automatic generation of load test suites and the assessment of the resulting software". IEEE *Transactions on Software Engineering,* 21(9), September 1995, 705-716.

41. Agrawal, H., Horgan, J., London, S., Wong, W. “A study of effective regression in practice” in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, November 1997, p 230-238.
42. Chu, C., Harrold, M., Rothermel, G., Untch, R. “Test case prioritization: An empirical study” in *Proceedings of the International Conference on Software Maintenance, 1999*, p 179-188.
43. http://www.sourceforge.net
44. http://sir.unl.edu
45. http://jtopas.sourceforge.net/jtopas
46. http://santuario.apache.org
47. http://sir.unl.edu/content/bios/xml-security.php
48. http://cloc.sourceforge.net/
49. http://www.eclemma.org.

.