DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR REGRESSION AND SYSTEM TEST SUITES

THESIS

submitted in fulfillment of the requirement of the degree of

DOCTOR OF PHILOSOPHY

to

YMCA UNIVERSITY OF SCIENCE & TECHNOLOGY

by

HARISH KUMAR

Registration No: YMCAUST/Ph07/2010

Under the Supervision of

Dr. NARESH CHAUHAN

PROFESSOR

Department of Computer Engineering

Faculty of Engineering and Technology

YMCA University of Science &Technology

Sector-6, Mathura Road, Faridabad, Haryana, INDIA

OCTOBER, 2017

CANDIDATE'S DECLARATION

I hereby declare that this thesis entitled "**DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR REGRESSION AND SYSTEM TEST SUITES"** being submitted in fulfillment of requirement for the award of Degree of Doctor of Philosophy in the Department of Computer Engineering under Faculty of Engineering and Technology of YMCA University of Science and Technology, Faridabad, during the academic year May 2011 to January 2016,is a bonafide record of my original work carried out under the guidance and supervision of **DR. NARESH CHAUHAN, PROFESSOR , DEPARTMENT OF COMPUTER ENGINEERING** and has not been presented elsewhere.

I further declare that the thesis does not contain any part of any work which has been submitted for the award of any degree either in this university or in any other university.

(HARISH KUMAR)

Registration No: YMCAUST/Ph07/2010

CERTIFICATE

This is to certify that the thesis titled **"DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR REGRESSION AND SYSTEM TEST SUITES"** by **HARISH KUMAR** submitted in fulfillment of the requirements for the award of Degree of Doctor of Philosophy in Department of Computer Engineering under Faculty of Engineering & Technology of YMCA University of Science & Technology Faridabad, during the academic year May 2011 to January 2016 is a bonafide record of work carried out under my guidance and supervision.

I further declare that to the best of my knowledge, the thesis does not contain part of any work which has been submitted for the award of any degree either in this university or in any other university.

DR. NARESH CHAUHAN
Professor
Department of Computer Engineering
Faculty of Engineering and Technology
YMCA University of Science & Technology Faridabad

Date:

The Ph.D. viva-voce examination of Research Scholar Harish Kumar (YMCAUST/Ph07/2010) has been held on …….

(Signature of Supervisors) (Signature of Chairman) (Signature of External Examiner)

ACKNOWLEDGEMENTS

First of all, I would like to thank **God**, the Almighty, for providing enough courage and his blessings to me for completion of this thesis.

I would like to express my sincere gratitude to my thesis supervisor, **Dr. Naresh Chauhan**, for his continuous guidance, valuable advice, constructive criticism and helpful discussions. I am very grateful to him for his continual encouragement, motivation and long hours spent throughout the completion of my work. He always offered wisdom, insight and a skilled hand in overcoming the hindrances faced. I greatly value his timely and valuable advices. He gave me the opportunity to learn various new things, and taught me a lot about research, teaching, and life. Without his warm encouragement, I would not have been able to accomplish this thesis.

I am grateful to **Prof. A. K. Sharma**, without his blessings I am not able to reach at this stage of my life. I am also thankful to **Dr. C.K. Nagpal**, from whom I have learnt a lesson of regularity and punctuality in life.

I gratefully acknowledge **Dr. Komal Kumar Bhatia,** Chairman Computer Engineering Department a true well-wisher of mine, who always supports at every stage of completion of this thesis. I express my sincere thanks to **Dr. Atul Mishra** who always support me like elder brother and providing his valuable suggestions for accomplishing this task. I would also like to thanks **Dr. Manjeet Singh** for his kind support.

I am thankful to **Mr. Vedpal** for his continuous support and encouragement for the completion of this thesis. I am thankful to **Mr. Umesh** and **Mr. Sushil Panwar** for their help and support in completing my research work. I extend my thanks to faculty members of Computer Engineering department for their support and cooperation. Although it is not possible to name individual, I cannot forget my well-wishers for their persistent support and cooperation. I am also thankful to all my students who helped me directly or indirectly in completing my research work.

I am also thankful to **Dr. Rashmi Popli , Dr. Preeti Sethi** and **Dr. Anita Arora** for their help and support. I would like to thank **Sh. J.P. Sharma, Sh. Mukesh Gupta, Sh. Mukesh Garg Sh. Sanjay, Sh Neeraj, Sh. Krishan Bhardwaj, Sh.Mohan Singh and Sh. Jitender**, for always providing me their support and necessary resources for the completion of this thesis.

I am grateful to my mother **Smt. Sheela Devi Tanwar** and my father **Sh. Ram Chand Tanwar** for their blessings. I can't forget father's efforts for making me a true human in my life. I am thankful to my mother in law **Smt. Mani Devi** and father in law **Mr. Attar Singh** for their blessings. I am also thankful to my foofa ji **Sh. Girraj Singh,** who always provide his suggestions for living a cheerful life. I am thankful to my elder brother **Mr. Surjeet Singh (Lambu Ji)** for his love and support**.** I would like to thank my younger brother **Niranjan Tanwar** and my sister **Anju** for their love and support. I am also thankful to my friends **Santosh Kumar** and **Yogesh Pandey** who always motivates me and made arrangements for getting me refresh after spending long hours while writing this thesis. I am also thankful for the support received from all members of Tanwar's family, especially I can forget the support from my uncles **Sh. Shobha Ram, Sh. Balkishan, Sh. Jiya Ram and Sh. Hem Chand** and my aunties. I am also thankful to my cousins **Pankaj Tanwar, Nitesh Tanwar, Sumit Tanwar and Parvesh Tanwar** for their respect and regards towards me.

Finally I would like to express my gratitude to my wife **Poonam Tanwar** who made a lot of compromises for the completion of this thesis. I like to thank my dear sons **Divyanshu Tanwar** and **Yatvik Tanwar** for the encouragement they have given to me, probably without knowing it.

Thanks to all of you!

(HARISH KUMAR)

ABSTRACT

Software testing being a critical element of the software development process is an important activity consuming almost 40 -50 % time of the total development process and a large part of resources and effort. However, testing process ensures the quality of the software, which is the major concern of the end customer. To improve the software quality, suitable test cases need to be designed and executed. It is a common problem in software testing that there are large number of test cases which is almost impractical to execute these test cases due to constraints of resources-, manpower and time. If the test cases are executed in sequence or in random order this results in that, test cases with higher risk cannot be detected earlier. Since test cases in the existing test suite can often be used to test a modified program, the test suite is used for retesting. However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite.

Keeping in view these problems of test case execution, there is requirement to prioritize the execution of test cases so as to detect critical bugs early as well as handling of big test suite becomes easy. Test case prioritization is a process of scheduling the test cases in a specific order which results in increasing the chances of early bug detection, thereby improving the software quality. This thesis focuses on the development of test case prioritization techniques at three levels i.e. unit testing, system testing and regression testing.

At unit testing level there may be large number of test cases to be executed by the developer, as the whole control structure of the software needs to be considered during testing. In this work, a test case prioritization technique for unit testing has been proposed based on analysis of structure of the program written by the developer. The proposed technique has been also extended to prioritize the test cases while performing the regression testing of a software component. The proposed technique has been validated and applied on three software cases studies. The results obtained show the efficacy of the proposed technique.

System testing is basically the testing of whole integrated system performed at various grounds such as performance, security and maximum load etc. This results in a large number of test cases. So to prioritize the test cases while performing system testing, a hierarchical system test case prioritization technique is proposed in this thesis, which is based on 12 comprehensive factors. A tool has been also implemented for this purpose. Similarly, to deal with large number of test cases during regression testing a module coupling effect based test case prioritization technique has been presented in this thesis, that helps the tester in finding the badly affected module due to change in a module. A tool has also been implemented for the same.

Data flow testing, a technique for performing white box testing, closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc. More over the problem becomes severe, when regression testing is being performed as there is need to execute all the existing test cases along with new one with even a single change in the code of a module under consideration. To prioritize the test cases while performing regression testing, a novel technique based on data flow testing concepts has been presented in this work. All proposed techniques in this thesis have been validated and the results obtained show the efficacy of the proposed techniques.

Thus this thesis largely focuses on the challenges found in performing regression testing. To overcome these challenges, the work presented in this thesis concentrates on designing and development of test case prioritization techniques that help the software testers in minimizing the testing efforts, cost and schedule of the project. Most of the proposed techniques being developed have been tested and implemented.

TABLE OF CONTENTS

Candidate's Declaration ii

Certificate iii

Acknowledgements iv

Abstract vi

Table of Contents viii

List of Tables xii

List of Figures xvi

List of Abbreviations xix

CHAPTER I: INTRODUCTION 1-8

1.1Test Case Prioritization 1

1.2 Motivation and Research Objectives 2

1.3 Challenges of Test Case Prioritization 5

1.4 Organization of Thesis 7

CHAPTER II: LITERATURE SURVEY 9-53

2.1 Introduction 9

2.2 Regression Testing 11

2.2.1 Test Suite Minimization 12

2.2.2 Test Case Selection 13

2.2.3 Test Case Prioritization 14

2.3 Coupling and its Types 14

2.3.1 Content Coupling 15

2.3.2 Common Coupling 16

2.3.3 Control Coupling 17

2.3.4 Stamp Coupling 18

2.3.5 Data Coupling 19

2.4 Cohesion and its Types 20

2.4.1 Functional Cohesion 20
2.4.2 Sequential Cohesion 21
2.4.3 Communicational Cohesion 22
2.4.4 Procedural Cohesion 24
2.4.5 Temporal Cohesion 31
2.4.6 Logical Cohesion 33
2.4.7 Coincidental Cohesion 35
2.5 Test Case Prioritization Technique 35
2.5.1 Classification of Test Case Prioritization(TCP)Technique 36
2.6 Call Graph 51
2.7 Conclusion 53
CHAPTER III: STRUCTURED PROGRAMMING BASED UNIT TEST CASE PRIORITIZATION(SPUTCP): PROPOSED WORK **55-84**
3.1 Introduction 55
3.2 SPUTCP Technique 56
3.2.1 Proposed SPUTCP Factors 56
3.3 Validation of SPUTCP Approach 59
3.3.1 Case Study of Employee Record Software 60
3.3.2 Analysis of Proposed SPUTCP Approach 64
3.3.3 Case Study of Saving Module of Income Tax Calculator 66
3.3.4 Case Study of Infix to Postfix Conversion 69
3.3.5 Case Study of Restaurant Management System 71
3.3.6 Case Study of Library Management System 74
3.3.7 Case Study of Income Tax Calculator 77
3.4 Structure Programming based Unit Regression Test Case Prioritization Approach 79
3.4.1 Explanation of the Process of SPURTCP 79
3.4.2 Validation of Proposed SPURTCP Approach 80

3.4.3 Analysis of Proposed SPRUTCP Approach 82
3.5 Conclusion 84
CHAPTER IV: HIERARCHICAL SYSTEM TEST CASE PRIORITIZATION(HSTCP) : PROPOSED WORK **85-106**
4.1 Introduction 85
4.2 Proposed HSTCP Approach 86
4.2.1 Prioritization of Requirements 87
4.3 Prioritization of the Module 96
4.4 Test Case Prioritization Process 97
4.5 Analysis of Proposed HSTCP Approach 98
4.5.1 Results obtained for HSTCP Approach 100
4.6 Implementation 104
4.7 Conclusion 106
CHAPTER V: REGRESSION TEST CASE PRIORITIZATION: PROPOSED WORK **107-152**
5.1 Introduction 107
5.2 Module- Coupling- Effect based Test Case Prioritization(MCETCP) Technique 107
5.2.1 Module Dependence Matrix 108
5.2.2 Procedure for Making Module Dependence Matrix 109
5.2.3 Proposed MCETCP Approach 110
5.2.4 Proposed Algorithm for finding highly coupled module 111
5.2.5 Evaluation & results of MCETCP Approach 112
5.3 Test Case Prioritization using Data Flow Testing 114
5.3.1 Analysis of the proposed data flow testing approach 116
5.4 Control-Structure-Weighted Test Case Prioritization(CSWTCP) Technique 141

5.4.1 Proposed CSWTCP Approach 141
5.4.2 Preparing a VDG 142
5.4.3 Calculation of the weight of a node in VDG 143
5.4.4 Calculating the weight of modified statement (WMS) 144
5.4.5 Calculating the weight of du paths(WDU) 146
5.4.6 Evaluation & Analysis of CSWTCP Approach 147
5.5 Conclusion 152
CHAPTER VI: CONCLUSIONS AND FUTURE SCOPE 153-154
6.1Conclusions 153
6.2 Benefits of Proposed Work 153
6.3 Future Scope 154
REFERENCES 155-166
APPENDIX-A 167-171
APPENDIX-B 173-175
APPENDIX-C 177-180
APPENDIX-D 181-189
APPENDIX-E 191-192
BRIEF PROFILE OF RESEARCH SCHOLAR 193
LIST OF PUBLICATIONS OUT OF THESIS 195

LIST OF TABELS

Table 2.1	Statement Coverage	39
Table 2.2	Branch Coverage	39
Table 2.3	Risk Analysis Table	42
Table 3.1	Prioritization factors and their weight for SPUTCP	58
Table 3.2	Case studies for validation of the proposed SPUTCP	60
Table 3.3	Independent paths of employee record case study	62
Table 3.4	Test cases covered and independent paths for employee record case study	62
Table 3.5	Count of proposed STUTCP factors present in the case of employee record	63
Table 3.6	Calculated TCPV for case study of employee record	63
Table 3.7	Faults detected for non prioritized order of test cases	64
Table 3.8	Faults detected for prioritized order for employee record case study	65
Table 3.9	Faults Detected for Random Order of Test Cases	65
Table 3.10	APFD Values for Various Techniques for employee record case study	66
Table 3.11	Independent paths for saving module case study	68
Table 3.12	Factors Covered by the test cases of case study of saving module	68
Table 3.13	Calculated TCPV for case study of saving module	69
Table 3.14	APFD Values for Various Techniques for case study of salary module	69
Table 3.15	Count of factors present in case study of infix to postfix case study	70
Table 3.16	Calculated TCPV for case study of Infix to postfix	71
Table 3.17	APFD Values for Various Techniques for case study of infix to postfix conversion	71
Table 3.18	Count of proposed SPUTCP factors in case study of restaurant management system	72

Table 3.19	Count of proposed SPUTCP factors in case study of restaurant management system	72
Table 3.20	Count of proposed SPUTCP factors in case study of restaurant management system	73
Table 3.21	TCPV for test cases of case study of restaurant management system	73
Table 3.22	Count of proposed SPUTCP factors present in the case study of Library management system	74
Table 3.23	Count of proposed SPUTCP factors present in the case study of Library management system	75
Table 3.24	Count of proposed SPUTCP factors present in the case study of Library management system	75
Table 3.25	Count of proposed SPUTCP factors present in the case study of Library management system	76
Table 3.26	Comparison of APFD Values	76
Table 3.27	Count of proposed STUTCP factors present in the case of gtc ( ) module	77
Table 3.28	Calculated TCPV for case study of gtc ( ) module	78
Table 3.29	Count of factors present in modified Case study of employee record	81
Table 3.30	TCPV of test cases for modified employee record case study	81
Table 3.31	DTCPV for the test cases	82
Table 3.32	Execution of test cases in non prioritized order	82
Table 3.33	Execution of test cases in prioritized order	83
Table 3.34	Execution of test cases in random order	83
Table 3.35	APFD Values for various approaches for modified employee record case study	84
Table 4.1	Factors considered for requirement prioritization	88
Table 4.2	Module Prioritization	96
Table 4.3	Test Case Prioritization	98
Table 4.4	Requirements Prioritization	99

Table 4.5	Module prioritization for income tax calculator case study	99
Table 4.6	Test case prioritization for test cases of tax module	100
Table 4.7	Fault detection in Generate Tax details (GTD) requirement	100
Table 4.8	Fault detection in Income tax deduction (ATD) requirement	101
Table 4.9	Fault detection in Accept Savings and Donation details (ASD)	101
Table 4.10	Fault detection in Income detail module of Accept income detail (AID)	101
Table 4.11	Fault detection in Income detail salaried module of Accept income detail (AID) requirement	101
Table 4.12	Fault detection Accept Personal detail (APD)	101
Table 4.13	Number and type of faults detected by all requirements	102
Table 5.1	Coupling Types and their values	109
Table 5.2	Cohesion types and their values	109
Table 5.3	Coupling Information for case study software	112
Table 5.4	Cohesion Information for case study software	112
Table 5.5	Module Coupling Matrix(C)	113
Table 5.6	Module Cohesion Matrix (S)	113
Table 5.7	Module Dependence Matrix (D)	113
Table 5.8	Test Case Design for income detail from the Independent Paths	120
Table 5.9	Definition and use of variable ‘len’ income detail module case study	121
Table 5.10	Du paths and test cases	121
Table 5.11	Test Cases for saving module case study from the Independent Paths	126
Table 5.12	Definition and use of variable ‘len’ for saving module	127
Table 5.13	Du paths & Test case coverage of variable ‘len’	127
Table 5.14	Test Case Design for income detail case study from independent paths	133
Table 5.15	Definition nodes and Usage nodes of variable or f income detail case study	135

Table 5.16	Du and dc paths with test coverage for income details module	135
Table 5.17	Fault and Test Cases	137
Table 5.18	New definition and uses of variables	138
Table 5.19	New du paths introduced	139
Table 5.20	Set of test cases after applying data flow TCP approach	139
Table 5.21	Data obtained after analyzing the income detail case study	139
Table 5.22	Weight assigned to the different nodes of VDG of Figure 5.17	144
Table 5.23	Proposed control structure weights	145
Table 5.24	Proposed nesting type weight	145
Table 5.25	Directly changed variables (DCV) & affected variables (AV) and their node weights in VDG	148
Table 5.26	List of du paths of DCV & AV	148
Table 5.27	Weights of different factors and WT for du paths	149
Table 5.28	Calculated WMS Values using WT value	149
Table 5.29	WDU Values for various du paths	150
Table 5.30	List of du paths arranged in descending values of WDU	150
Table 5.31	Test cases and their weights	151
Table 5.32	Faults and Test Cases	151

LIST OF FIGURES

Figure 2.1	Different Types of Coupling	14
Figure 2.2	Types of Cohesion	20
Figure 2.3	Classification of Test Case Prioritization Techniques	37
Figure 2.4	Types of Coverage Based TCP	38
Figure 2.5	Call Graph	52
Figure 3.1	Pictorial Representation of SPUTCP Technique	56
Figure 3.2	Algorithm for SPUTCP Approach	59
Figure 3.3	CFG of the case study of employee record software	61
Figure 3.4	Comparison of Proposed SPUTCP, Random and Non Prioritize approach	66
Figure 3.5	CFG for case study of saving module of income tax calculator software	67
Figure 3.6	CFG for case study of infix to postfix conversion	70
Figure 3.7	Comparison of Prioritized and Non prioritized approach	74
Figure 3.8	Comparison of APFD Values	77
Figure 3.9	Comparison of Prioritized and Non prioritized approach for Case Study of gtc () module	78
Figure 3.10	Algorithm for SPURTCP approach	80
Figure 3.11	Comparison of APFD values of different prioritization techniques	84
Figure 4.1	Hierarchical System Test Case Prioritization (HSTCP) Technique	87
Figure 4.2	Implementation Complexity factors	90
Figure 4.3	Expected Faults	93
Figure 4.4	Graph for Proposed HSTCP approach based on requirements	102
Figure 4.5	Graph obtained using PORT approach	102
Figure 4.6	Graph for non – Prioritized test suite	103
Figure 4.7	Comparison between Random, PORT, and HSTCP approach	104

Figure 4.8	Snapshot 1 of HSTCP Tool	105
Figure 4.9	Snapshot 2 of HSTCP Tool	105
Figure 4.10	Snapshot 3 of HSTCP Tool	105
Figure 4.11	Snapshot 4 of HSTCP Tool	105
Figure 4.12	Snapshot 5 of HSTCP Tool	105
Figure 4.13	Snapshot 6 of HSTCP Tool	105
Figure 5.1	Call Graph Example	108
Figure 5.2	Components Showing the Process of MCETCP	111
Figure 5.3	Algorithm for finding highly coupled module	111
Figure 5.4	Call Graph of Case Study Software	112
Figure 5.5	Case Study Software Call Graph with module dependence values	114
Figure 5.6	Algorithm for prioritizing the test cases using data flow testing	115
Figure 5.7	Algorithm for prioritizing the test cases within a set	116
Figure 5.8	CFG for income details module	119
Figure 5.9	APFD values for random and proposed data flow TCP for income detail module	122
Figure 5.10	CFG for saving module of income tax calculator	125
Figure 5.11	APFD values for random and proposed data flow TCP for saving module case study	128
Figure 5.12	CFG for income details module	132
Figure 5.13	Comparison of Random, Previous and Proposed data flow testing approach	140
Figure 5.14	Process of CSWTCP	142
Figure 5.15	Algorithm for making VDG	142
Figure 5.16	Code for Sample program	143
Figure 5.17	VDG for sample program	143
Figure 5.18	Algorithm for assigning the weight of nodes in VDG	144

Figure 5.19	VDG with assigned weights to nodes for the sample program	144
Figure 5.20	VDG for modified program	147
Figure 5.21	VDG for modified program with assigned weights of nodes	148
Figure 5.22	Comparison of APFD values of Random and CSWTCP Approach	152

LIST OF ABBRIVIATIONS

APFD	Average Percentage of Fault Detection
AV	Affected Variable
BCO	Bee Colony Optimization
CBSS	Component Based Software System
CFG	Control Flow Graph
CIG	Component Interaction Graph
CSWTCP	Control-Structure Weighted Test Case Prioritization
DCV	Directly Changed Variable
DTCPV	Difference between Test Case Prioritization Values
DU	Definition Usage
DC	Definition Clear
GA	Genetic Algorithm
HSTCP	Hierarchical System Test Case Prioritization
MCETCP	Module-Coupling-Effect based Test Case Prioritization
SDLC	Software Development Life Cycle
SPUTCP	Structured Programming based Unit Test Case Prioritization
SPURTCP	Structured Programming based Unit Regression Test Case Prioritization
PFV	Prioritization Factor Value
RPFV	Requirement Prioritization Factor Value
TCP	Test Case Prioritization
TCPV	Test Case Prioritization Value
TCWP	Test Case Weight Prioritization
VDG	Variable Dependence Graph
WDU	Weight of du Path
WPU	P-use Statement Weight
WT	Total Weight

Chapter I

INTRODUCTION

1.1 TEST CASE PRIORITIZATION

Software testing is an important and critical part of the software development process, on which quality of software product is strictly dependent. Testing related activities consume almost half of the total time incurred in the software development process and also consume a large part of the effort required for producing software [1, 33, 71, 112, 117, 118].

There exist many types of testing and test strategies, however all of them share a common goal, that is, increasing the software engineer's confidence in the proper functioning of the software [1, 80]. Towards this general goal, a piece of software can be tested to achieve various more direct objectives such as exposing potential design flaws or deviations from user's requirements, measuring the operational reliability, evaluating the performance characteristics, and so on. To serve each specific objective, different techniques can be adopted.

Software testing [15, 82, 93] occurs continuously during the software development life cycle to detect errors as early as possible. Since test cases in the existing test suite can often be used to test a modified program, the test suite is used for retesting. However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite. Thus, the size of test suites grows due to following reasons.

1. A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a test set achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed. Coverage criteria are used as a stopping point

to decide when a program is sufficiently tested. In this case, additional tests are added until the test suite has achieved a specified coverage level according to a specific adequacy criterion. For example, to achieve statement coverage adequacy for a program, one would add additional test cases to the test suite until each statement in that program is executed by at least one of the test cases.

2. There may be unnecessary test cases in the test suite including both obsolete and redundant test cases. A change in a program causes a test case to become obsolete by removing the reason for the test case's inclusion in the test suite. A test case is redundant if other test cases in the test suite provide the same coverage of the program. Thus, because of the obsolete and redundant test cases, the size of the test suite continues to grow unnecessarily as software changes are made.

3. As new test cases are added to the test suite to the new or changed requirements or to maintain test-suite adequacy, the size of the test suite grows and the cost of running it on the modified software (i.e., regression testing) increases.

However due to resource constraints, it is almost impossible to execute all the test cases. Therefore there is requirement to prioritize the execution of test cases so as to increase chances of early detection of faults. Test case prioritization is the process of ordering the test cases of the test suite based on certain criteria like code coverage, fault detection capability, risk exposure etc. so that critical faults may be detected earlier. Test case prioritization can be done at Unit testing, Regression Testing and System testing level. This thesis focuses on test case prioritization at these three levels.

2. MOTIVATION AND RESEARCH OBJECTIVES

Although there exist many test case prioritization techniques in the literature, there are certain points where the existing methods can be optimized or there is requirement of

new technique. A critical study of literature available in the area of test case prioritization has been performed and some shortcomings were identified which motivated to pursue this research work.

1. While performing the white box testing for a module, there may be large number of test cases executed by the developer to ensure the correct functionality of their code. This process involves a lot of efforts. But if somehow a developer is able to get the prioritized order of the test cases which he/she is going to execute to ensure the correct functionality during the process of white box testing, makes the task easier.

2. System testing is actually a series of different tests to test the whole system on various grounds where bugs have the probability to occur. The ground can be performance, security, maximum load, etc. The integrated system is passed through various tests based on these grounds and depending on the environment and type of project. This results in large number of test cases, so there is need to prioritize the test cases while performing system testing. In literature [6,36,40] ,the system test cases have been prioritized based on the requirements considering various factors like types of requirements, complexity of requirements, mapping of design and code, fault proneness of mapped code, fault detection rate etc. But there are many factors which have not been considered till now. These factors are show stoppers requirements, frequency of execution of requirement, cost, time, penalty etc. So this research work aims at designing a test case prioritization technique for system test cases considering these new factors.

3. Regression Testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is modification. The following difficulties occur in retesting:

    - Large systems can take a long time to retest.

    - It can be difficult and time-consuming to create the tests

- It can be difficult and time consuming to evaluate the tests. Sometimes, it requires a person in the loop to create and evaluate the results.

- Cost of testing can reduce resources available for software improvements.

Therefore, there is need to prioritize the test cases while performing regression testing. In literature, there exist many techniques for regression test case prioritization. However, these techniques do not consider the effect of changes in one module being propagated in other modules of the software. These techniques are not able to prioritize the modules and their test cases which are badly affected with the changes. Most of the prioritization techniques consider all the modules with their test cases, prioritizes them with some criterion like risk based prioritization, coverage based, fault detection rate, etc and find out the prioritized test cases with the aim of getting high fault detection. But, it becomes cumbersome to analyze the prioritization techniques by finding fault detection rate of all test cases in the test suite. Instead of this, if the approach is to find out the module/modules which are badly affected and then prioritize the test cases, this will provide the high severity bugs very early.

4. While performing the data flow testing the focus is only on the definition and use of the variables within the program under test. A define-usage path (du-path) with respect to a variable v is a path between the definition node and the usage node of that variable. A definition- clear path (dc- path) with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v. Usage node can either be a predicate-usage or a computation-usage node. However, all-du-path criteria is not able to detect critical bugs earlier due to the following reasons:

   - Some of the du-paths may be non-dc. Non-dc paths are the paths wherein the variable is defined more than once. These du-paths which are non-dc may be a problematic area for the testers and affect test case prioritization.

- There may be a large number of test cases corresponding to all-du-paths. So it may not be possible to execute all of these test cases.

The main objective of this research is to design test case prioritization techniques for unit, system and regression test suites. To achieve this objective, the work on following goals has been performed in this thesis:

- To develop and validate a method for unit test case prioritization based on the analysis of source code written by the developers.

- To develop and validate a method for System Test case Prioritization based on types of requirements, complexities included in requirement mapped design and code, fault proneness of mapped code, fault detection rate, etc.

- To develop and validate a method for Regression Test case Prioritization based on module coupling information and data flow testing concepts.

1.2 CHALLENGES OF TEST CASE PRIORITIZATION

Based on the motivation points considered and thereby objectives defined, this section discusses the challenges and their solutions while performing test case prioritization.

Prioritizing the test cases while performing unit testing: The issue is to manage large number of test cases while performing unit testing of a module as the whole control structure of the software needs to be covered during testing. Further, the problem enhances when there is change in an established module due to any reason such as change in user's requirements, appearance of critical bugs, etc.

Solution: *In order to cope up with large number of test cases in unit testing, a test case prioritization technique has been proposed based on analysis of source code*

written by the developers. This analysis provides the importance level of each statement in the code based on certain factors.

Prioritizing the test cases while performing system testing: The issue is to manage large number of test cases while performing system testing as it involves various grounds of testing such as performance, security, maximum load, etc.

Solution: *In order to prioritize system test suite, a requirement based prioritization approach has been proposed based on a list of 12 comprehensive factors. These factors filter the important requirements and thereby these requirements are mapped to their corresponding test cases. Thus a hierarchical test case prioritization technique to obtain prioritized test suite has been proposed.*

Prioritizing the test cases while performing regression testing: The issue is to rerun all the test cases while performing regression testing with even a single line change in code.

Solution: *In order to have a prioritized regression test suite, a module coupling effect based test case prioritization technique has been proposed that uses the coupling information among various modules in the software and thereby identifying the badly affected module due to change in the software and subsequently prioritize the test cases of this affected module.*

Prioritizing the test cases while performing regression testing using data flow testing technique: The data usage for a variable affects the white box testing and thereby the regression testing. There may be large number of test cases corresponding to all du-paths.

Solution: *To resolve this issue a new test case prioritization technique has been proposed that finds the newly introduced non-dc paths in the modified program and also finds the paths which have been changed from dc to non-dc paths. Based on these criteria du-paths are prioritized so that a critical bug is exposed earlier. However, there may a large number of du-paths having equal priority. To resolve this, a control structure weighted test case prioritization technique has been proposed in this thesis.*

This prioritization technique takes into consideration the complexity of the statements, where the variable has been used, and various aspects of structured programming.

1.4 ORGANIZATION OF THESIS

The thesis has been organised in the following chapters:

Chapter 1: Covers the introduction of the thesis.

Chapter 2: The basic concepts of software testing, regression testing and test case prioritization are discussed in this chapter. A detailed review of the available test case prioritization techniques and the problems associated with these techniques are also discussed.

Chapter 3: A test case prioritization technique for unit testing based on analysis of structure of the program called structured programming based unit test case prioritization (SPUTCP) technique is presented in this chapter. The proposed technique is also extended for regression testing, named as structured programming based unit regression test case prioritization (SPURTCP) and is discussed in this chapter. The proposed approach is validated to show the efficacy as compared to the random techniques.

Chapter 4: A hierarchical system test case prioritization (HSTCP) technique for prioritizing the system test cases is proposed in this chapter. To demonstrate the proposed approach a tool is developed and its working is also discussed. The proposed approach is also compared with random as well as previous existing approach.

Chapter 5: This chapter is concerned with prioritization of test cases while performing regression testing and is divided in three sections. In first section, a module-coupling-effect based test case prioritization (MCETCP) technique for regression testing is proposed. The approach helps in finding a badly affected module due to change during regression testing and a tool is implemented for the same. Second section of the chapter discusses a novel test case prioritization technique for

regression testing using data flow testing concepts. The third section discusses a control-structure-weighted test case prioritization (CSWTCP) technique for regression testing and a tool is also developed for the same. All the proposed techniques in this chapter are validated and the results obtained show the efficacy of these techniques.

Chapter 6: It concludes the outcome of the work proposed in this thesis. It also discusses the possibilities of future research work based on the proposed approaches.

Chapter II

LITERATURE SURVEY

2.1 INTRODUCTION

Software testing is the process of analysis so as to find out the difference between the observed and the required conditions and to evaluate its features [82, 39, 42, 43, 28]. Software Testing is the process of verifying a system or its component with the intent to check whether it satisfies the desired requirements as stated by the end customer. This activity is an important and critical part of the software development process, on which quality of software product is strictly dependent [62]. Testing related activities consumes almost half of the total time incurred in the software development process and also consumes a large part of the effort required for producing software. Software testing helps in developing quality software [82, 60, 61, 64, 89, 115, 103]. It is a process which continues all the way through software development.

Software testing basically incorporates Verification and Validation activities [15, 120]. The verification and validation activities are the basis for the any type of testing. It can also be said that the testing process is a combination of verification and validation. The purpose of verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing and will not be allowed to transmit further. The validation process starts replacing the verification in the later stages of SDLC. Validation is a very general term to test the software as a whole in accordance with the end user expectations. Verification and Validation (V&V) are the building blocks of the testing process.
Validation process has following three activities which are also known as the three levels of validation testing.

- **Unit Testing**

  Unit is the smallest possible testable component of the software [80].Unit Testing is a basic level of testing which cannot be overlooked and confirms

the behaviour of a single module according to its functional requirements [1, 12, 25].

- **Integration Testing**

  This validation technique combines all unit tested modules and performs a test on their integration. Unit modules are not independent and are related to each other by interface specifications between them. When one module is combined with another in an integrated environment, interfacing between units must be tested. Therefore ensuring proper communication between the modules integration testing has to be performed.

- **System Testing**

  This particular level of software testing focuses on the testing of entire integrated system. This type of testing incorporates many types of testing, as the full system can have various users in different environments. These are performance testing, load testing, stress testing, compatibility testing etc. The validity of the whole system is checked against the requirement specifications.

Testing can be classified in many ways. One of the most basic classifications is that on the basis of the knowledge testing in which code is known is called white box testing where as the other is called black box testing. The goal of both white box testing and black box testing is to improve the fault finding capacity of the software. Towards this general goal, a piece of software can be tested to achieve various more direct objectives such as exposing potential design flaws or deviations from user's requirements, measuring the operational reliability, evaluating the performance characteristics, and so on. To serve each specific objective, different techniques can be adopted. During the review it was realized that testing forms an integral part of management actives and is even used in medical field and essential in new technologies like cloud [7, 18, 19, 20, 22, 24, 26, 78, 105, 106]. The development of ERP systems has also increased the importance of testing [29]. The security implementations are also highly dependent of good testing [65].

Software requirements are continuously changing. Due to these changing requirements software is modified accordingly to satisfy the needs of the customer. When software is modified there is always need to write new test cases for the modified version. These new test cases are executed to ensure that the modifications do not have any adverse effect on the previously working software. For this purpose regression testing is performed. This review has been conducted as per the guidelines proposed by Kichenham [69].

The remainder of this chapter has been organized as follows. In sub- section 2, the concept of regression testing has been described. Sub-section 3 presents a brief discussion of coupling, sub-section 4 discusses the concept of cohesion, sub-section 5 pertains to the classification of various test case prioritization techniques and finally sub-sections 6 deals with the allied concepts and the last section gives the conclusion.

2.2 REGRESSION TESTING

Regression testing is a kind of software testing that intends to find new software bugs, in existing software system after changes such as modifications, patches or configuration changes, have been made to the system. The main purpose of regression testing is to ensure that changes as mentioned above have not introduced new faults in the software [1, 35, 37, 67]. IEEE software glossary defines regression testing as follows [58].

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

The main reason for regression testing is to check whether a change made in one part of the software affects other parts of the software or not [120]. Regression testing can be performed to test a system by selecting the appropriate minimum set of test cases needed to adequately cover a particular change [73]. Regression testing is a resource and time consuming activity. While performing the process of regression testing a tester has to execute the previous test cases written for ensuring the correct

functionality of the software as well as the new test cases which have been introduced due to the modification. So, there are a large number of test cases required to test the software. However, due to time and cost constraint it may not be possible that all past test cases be executed whenever change is made in software.

The three techniques for accomplishing this task are selection, minimization and prioritization. Minimization techniques describe the elimination of redundant test cases from a test suite. It attempts to select the minimal set of test cases T, a subset of initial test case suite, which yields coverage of the modified or effected portion of the program [80]. Selection technique opts for the test cases that are significant to the recent modifications [11, 15, 96, 120]. Prioritization techniques prioritize the test case so that if the testing is prematurely terminated, even then also the fault detection is maximized. The process increases the plausibility of the test cases being executed in the given order; they will more closely meet the objective of finding maximum faults then otherwise [15, 107, 113].

2.2.1 Test Suite Minimization

This section discusses the concept of test suite minimization and its various approaches that have been put forward in the literature and future directions. The attributes of good test suite minimization techniques have been considered while analyzing various techniques.

Test suite minimization is the technique to reduce the size of the test suite [120]. This can be done by removing the redundant test case. The removal of the redundant test case has the risk that minimization should not lead to a scenario where robustness of the testing process is compromised. There are two problems involved here. The minimization process has been mapped to minimal hitting set formulation. Two approaches have been suggested in the literature. The first one is to decompose a bigger requirement into smaller one, so that each requirement is satisfied by a single test case [1]. The second approach suggests crafting of the test cases in such a way that they cater to a particular requirement.

The minimization problem is an NP Complete problem [120]. Therefore, the technique used to solve NP Complete problem can be used to solve the above problem as well. The literature suggests two ways of dealing with the problem. The first is the application of approximation algorithm and the second is the application of AI based search techniques like genetic algorithms and Ant Colony Optimization [55, 63, 57, 110]. However, it will be not apt to compare the techniques as they have different goals.

2.2.2 Test Case Selection

Regression test case selection is similar to test case minimization, in the sense; both of them reduce the test case suite. However, the key difference in the approach as observed in the literature is that while the test case selection concentrates on the changes between the prior and the subsequent version of the program [94, 95]. One of the earliest studies by Rothermel [94] proposed a technique which reveals the test cases relevant to modification.

It may be noted that if there are any modifications in the program, then the code is bound to change. The change in the code, referred to as textual difference can be a good source of finding out the modifications. This approach was used by Volkolos and Frankl [30]. In the approach they used a Unix tool called diff for identifying the differences. The name of the tool developed was Pythia. The tool was capable of analyzing large software systems written in C [30]. However, it may be noted that Graph walk approach proposed by Rothermel and Harrold [94] was carried forward in different works in the 1993-1997 period. Investigation of these graphs showed that their size may be quadratic. In some of the studies, it was also observed that the relationship between control dependence graph size and program size is linear. An experiment performed implemented tools for constructing the two types of control dependence graphs. This was made to run on about 3000 C functions extracted from a wide range of source programs. The results supported the earlier conclusions. The concept of Control dependency graph was extended to system dependency and finally to System dependency graph. The idea of textual difference explained earlier in the section depends on the graph walk. In the review many other techniques were also studied [8, 91, 97, 119].

2.2.3 Test Case Prioritization

Prioritization techniques promote reusability by implying effective regression testing. It is an important phase in software maintenance activities [56, 75]. The goal is achieved when the software program performs better than the earlier version. Prioritization of Regression Test Cases is an approach that converts the original test suite to one that has priority associated with each test case. A test case that covers large number of potential points of faults may have higher priority.

2.3 COUPLING AND ITS TYPES

One of the major factors in deciding the importance of a module is the type of coupling. Coupling plays an important role in both the type of data transfer and the type of error that may crop in. The work presented in this thesis uses the concept of coupling. Therefore a brief overview of coupling has been discussed in this chapter.

The following section throws some light on the definition and the types of coupling. Coupling can be defined as the degree to which each program module relies on the other module [82]. Coupling can be "low" or "high". Generally, data coupling is considered as the best type of coupling, followed by stamp coupling, control coupling, common coupling, and content coupling. This is true as practically, no coupling is not possible.

The goodness of a type of coupling has been represented in Figure 2.1.

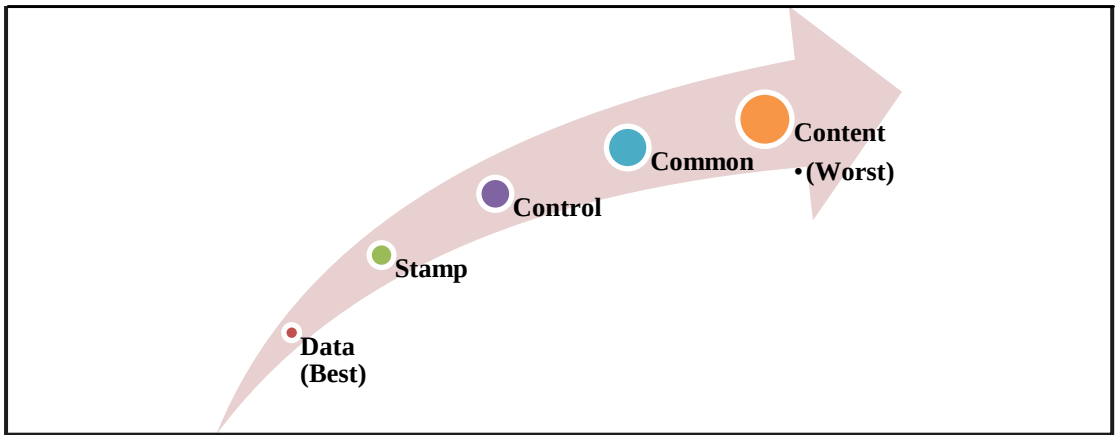


Figure 2.1: Different types of coupling

The following categories place the coupling from the highest to the lowest coupling. The categories are as follows:

2.3.1 Content Coupling

Content coupling is when one module modifies or relies on the internal working of another module. This means when a module accesses the local data of another module. This implies that changing the way the second module produces data may results in the changing of the dependent module.

For example result variable in module1 ( ) calculates result as (a*b).

```
int module1()
{
int result;
…
return result;
}
```

and in the module2 (int result) , result is effected as result = result+d.

```
int module2(int result)
{
int d;
…
result=result+d;
return(result);
}
```

The above is an illustration of content coupling as the value of a variable is changed in another variable.

Example of Content Coupling:

```
//Second Function uses the internal working of First Function
int FirstFunction (int a)
{
```

```
printf ("Inside First Function\n");
  a += 1;                                  //changing value of int a
goto label1;                                        //program control shifts to label1

return a;
}
void SecondFunction ()
{
printf("Inside Second Function\n");
label1:
printf("At Label1\n");
}
```

2.3.2 Common Coupling

Common coupling between two modules occurs when two modules share the same global data. For example if the two modules use a global variable, then they are bound by common coupling. In such cases changing the shared resource implies changing all the modules using it. One of the illustrations of common coupling is as follows.

```
int a;
void module1()
{
a=5;
}
void module2()
{
//Uses a;
}
```

Example of Common Coupling:

```
int  i;
//Here i is the global variable used by First Function and Second Function
int FirstFunction (int a)
```

```
{
if (a > 0)
  {
i++;
    a = 0;
  }

return a;
}
void SecondFunction()
{
if(i> 0)
  {
i = 1;
  }
else
  {
i = -1;
  }
}
```

2.3.3 Control coupling

Control coupling is one in which a module controls the flow of another module. This can be done by passing it information on what is to be done. One of the example of such kind of coupling can be passing a what-to-do flag to another module.
In Operating System Implementations, the semaphores present an excellent example of such kind of coupling.

Example of Control Coupling:

```
//main controls the flow of abc() Function

int flag;
```

```
void  abc()
{
Flag=-1
}
int main()
{
abc()
flag=0;
}
```

2.3.4 Stamp Coupling

Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part. One of the examples of such kind of coupling can be passing a whole record to a function that only needs some part of it. This may result in changing the way a module reads a record because a field, which the module doesn't need, has been modified.

```
void module1()
{
int list[20];
//Input
Module2(list);
}
void module2(int * list)
{
Printf(“%d”,list[5]);
}
```

The second module needed only the first element of the array but was provided with the whole array. Therefore, the above is an example of Stamp coupling.

Example of Stamp Coupling

```
Struct myStruct {
int myint;
char mychar;
longmylong;
}

//Part of structure is used by First Function and Second Function
void First Function ()
 {
My Struct structA;
int x;
   ...
   x = Second Function(structA);
}
int SecondFunction (myStructstructB)
{
return (structB.myint+1);
}
```

2.3.5 Data Coupling

Data coupling occurs when modules share data through parameters. Each datum is an elementary piece, and these are the only data shared. Example of the above can be passing an integer to a function that computes its square root.

For example the following function of a class called Math computes the square root of the value that is passed inside the function.

```
x=5;
Math.sqrt(x);
```

Example of Data Coupling:

```
intFirstFunction (inti)      //i is shared via a parameter
{
i = i+2;
```

```
return i;
}
intSecondFunction (int k)
{
  k = k+i;
return k;
}
```

2.4 COHESION AND ITS TYPES

Coupling depicts the interrelation between the modules and cohesion represents the intra relation. Cohesion is like glue that holds the module together [80]. If cohesion is high, it signifies how good a module handles its various components. Cohesion can also be classified as follows (Figure 2.2).

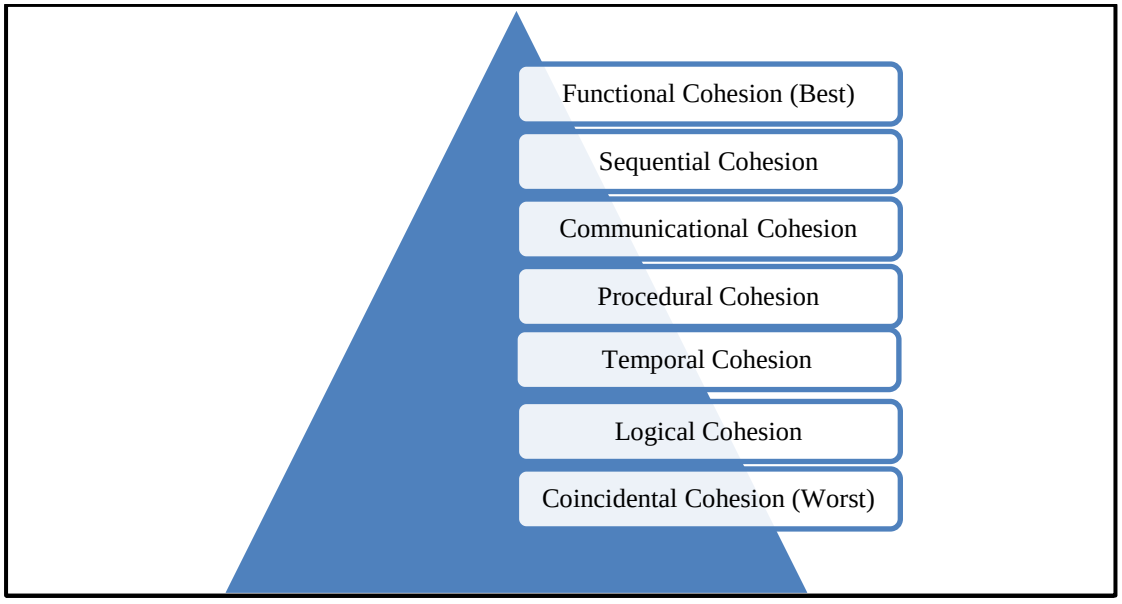


Figure 2.2: Types of Cohesion

2.4.1 Functional Cohesion

There can be many motivations of putting two parts in the same module. One of the best reasons can be their same functionality. Functional cohesion represents the scenario wherein two parts were put in the same module as their function is same.

Examples of functionally cohesive modules are

1. Computing cosine of angle
2. Calculate net salary of employee

Notice that each of these modules has a strong, single-minded purpose. When its boss calls it, it carries out just one job to completion without getting involved in any extracurricular activity.

```
main( )
{
float taxrate = .15;
float hourly= 10.00;
int hoursperweek = 40;
grosspay = hourly * hoursperweek;
taxes = grosspay * taxrate;
netpay = grosspay - taxes;
printf("\nGross pay"%f,grosspay);
printf("\nTaxes:f",taxes);
printf("\nNet pay:%f",netpay);
getch();
}
```

2.4.2 Sequential Cohesion

The output of one part is an input to another part then there is a strong reason to put then in the same module. This is referred to as sequential cohesion. Example of this type of cohesion is given below.

```
1. /* to insert a new record into the file*/
2. void insert (char *a)
3. {
4. FILE *fp1;
5. emp *temp1=(emp*)malloc(size of(emp));
6. temp1->name=(char*)malloc(200*size of (char));
```

```
7. fp1=fopen(a,”a+”);
8. if(fp1==NULL);
9. perror(“”);
10. else
11. {
12. printf(“enter the emplyeee id\n”);
13. scanf(“%d”,&temp1->empid);
14. fwrite(“&temp1->empid,sizeof(int),1,fp1”);
15. printf(“enter  employee name\n”);
16. scanf(“%[^\n]s”,temp1->name);
17. fwrite(temp1->name,200,1,fp1);
18. count++;
19. }
20. fclose(fp1);
21. free(temp1);
22. free(temp1->name);
23. }
```

2.4.3 Communicational Cohesion

If two parts of the program are able to communicate, then they are generally put in the same module. This is referred to as communicational cohesion.

Example of Communication Cohesion:

A communicational cohesive module is one whose elements contribute to activities that use the same input or output data.

1. Find the title of book
2. Find the price of book
3. Find publisher of book
4. Find author of the book

These four activities are related because they all work on the same input data, the book, which makes the “module” communication ally cohesive.

```
#include<stdio.h>
#include<conio.h>
struct lib_books
{
char title[20];
char author[15];
int pages;
float price;
};
struct lib_books, book1, book2, book3;
main()
{
int no.;
printf(“Enter book number”);
scanf(“%d”,&no);
switch(no)
{
  Case 1: scanf(“%s %s %d”,&book1.title,&book1.author,&book1.pages);
          printf(“%s %s %d”,book1.title,book1.author,book1.pages);
          break;
Case 2:   scanf(“%s %s %d”,&book2.title,&book2.author,&book2.pages);
          printf(“%s %s %d”,book2.title,book2.author,book2.pages);
          break;
Case 3:   scanf(“%s %s %d”,&book3.title,&book3.author,&book3.pages);
          printf(“%s %s %d”,book3.title,book3.author,book3.pages);
          break;
default:   printf(“WRONG NO”);
}
}
```

2.4.4 Procedural Cohesion

If two parts have been structured in the same manner, then there is a strong reason to put the two parts in the same module. This is referred to as procedural cohesion.

Example of Procedural Cohesion

A procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next. (Remember that in a sequentially cohesive module data, not control, flows from one activity to the next.) Here is a list of steps in an imaginary procedurally cohesive module.

1. Clean utensils from previous meal
2. Make phone call
3. Take shower
4. Chop vegetables
5. Set table

```
#include <stdio.h>
void main()
{
  int total = 45;
 int divider = 7;
  int a = 0;
  int b = 0;
  a = total/divider;
  printf(" total is %d  and  divider is %d", total, divider);
  printf("\n a is  %d.", a);
  b = total%divider;
  printf("\nThere are %d  left over.\n", b);
}
```

Another example is given below

```
int b_sort(int*,int);
 void f_write();
 void avg();
 void fprint();
 void f_sort();
 void roll();
int b_sort(int x[],int n)
{
 int hold,j,pass,i,switched = 1;
 for(pass = 0; pass < n-1 && switched == 1;pass++)
 {
  switched=0;
  for (j=0;j<n-pass-1;j++)
   if (x[j]>x[j+1])
   {
    switched=1;
    hold = x[j];
    x[j] = x[j+1];
    x[j+1]=hold;
    }
   }
return(0);
}
void f_write()
{
  int roll,ch,mark;
  char nam[50];
   FILE *fp;
```

```
  clrscr();
  fp = fopen("student.txt","a");
   printf("ENTER ROLL NUMBER, NAME , MARKS \n");
  ch =1;
  while(ch)
  {
  scanf("%d%s%d",&roll,&nam,&mark);
  fprintf(fp,"%d %s %d\n",roll,nam,mark);
  printf("\n\n press 1 to continue,0 to stop");
  scanf("%d",&ch);
  }
   fclose(fp) ;
}
void fprint()
{
  int marks[100],rollno[100],x[100],i;
  char name[100][50];
  FILE *fp;
  clrscr();
  fp = fopen("student.txt","r");
   i=0;
   printf("ROLLNO       NAME        MARK\n");
   while(!feof(fp))
  {
     fscanf(fp,"%d %s %d\n",&rollno[i],&name[i],&marks[i]);
     printf(" %d %s %d\n",rollno[i],name[i],marks[i]);
     i=i+1;
   }
   fclose(fp);
```

```
   printf("\n\n\nPRESS ANY KEY");
   getch();
  }
void f_sort()
  { int marks[100],rollno[100],x[100],n,i,j;
    char name[100][50];
    FILE *fp,*fm;
    fp = fopen("student.txt","r");
    fm = fopen("marks.txt","w");
    i=0;
   while(! feof(fp))
    {
     fscanf(fp,"%d %s %d\n",&rollno[i],&name[i],&marks[i]);
     x[i]= marks[i];
     i=i+1;
      }
       n=i;
       b_sort(x,n);
    for(i=0;i<n;i++)
    {
    printf(" %d\t",x[i]);
    }
 for(i=0;i<n;i++)
 {
   for (j=0;j<n;j++)
   {
   if(x[i]==marks[j])
   {
      fprintf(fm,"%d %s %d\n",rollno[j],name[j],marks[j]);
```

```
    }
   }
 }
  fclose(fm);
  fclose(fp);
  printf("\n\n\nPRESS ANY KEY");
  getch();
}
void roll()
{   int i,roll,ch,mark,roll1;
    char name[50];
    FILE *fm;
    ch=1;
  while(ch)
  { clrscr();
    fm = fopen("marks.txt","r");
    printf(" \n ENTER ROLL NUMBER - ");
    scanf("%d",&roll1);
      i=0;
   while(! feof(fm))
    {
     fscanf(fm,"%d %s %d\n",&roll,&nam,&mark);
     if(roll1==roll)
    {printf("\nROLLNO.     NAME        MARKS\n ");
     printf(" %d %s %d\n",roll,nam,mark);
     break;
     }
     else
     i=i+1;
```

```
    }
  printf("\n\npress 1 to see student info, 0 to return to main menu\n");
  scanf("%d",&ch);
  fclose(fm);
  }
 }
void avg()
 {
    int marks[100],rollno[100],n,i;
    float avg,x;
    char name[100][50];
    FILE *fm;
    fm = fopen("marks.txt","r");
    i=0;
   while(! feof(fm))
    {
     fscanf(fm,"%d %s %d\n",&rollno[i],&name[i],&marks[i]);
     x = x + marks[i];
     i=i+1;
      }
     n = i;
   avg = x/n;
  printf("AVERAGE MARKS OF %d STUDENTS ARE -  %f ",n,avg);
  fclose(fm);
  printf("\n\n\nPRESS ANY KEY");
   getch();
 }
void main()
{
```

```
int marks[100],rollno[100],x[100],n,i,j,roll,c,mark,roll1;
char name[100][10],nam[50];

while(c!=6)
 {
  clrscr();
  printf("GIVE CHOICE--\n");
  printf("   1 TO ENTER STUDENT INFO.\n");
  printf("   2 TO SEE STUDENT.TXT FILE\n");
  printf("   3 TO SORT FILE ON BASIS OF MARKS\n");
  printf("   4 TO PRINT STUDENT INFO. USING ROLL NO\n");
  printf("   5 TO FIND AVERAGE OF MARKS\n");
  printf("   6 TO EXIT\n\n--");
  scanf("%d",&c);
  clrscr();
  switch(c)
  {
  case 1:
       f_write();
       break;
  case 2:
       fprint();
      break;
  case 3:
      f_sort();
      break;
  case 4:  roll();
       break;
  case 5:  avg();
```

```
        break;
    case 6:
        break;
    default:
        break;
    }
   }
}
```

2.4.5 Temporal Cohesion

If two parts are to run at the same time, then they can be put in the same module. This is called temporal cohesion.

Example of Temporal Cohesion

A temporally cohesive module is one whose elements are involved in activities that are related in time. Picture this late-evening scene:

1. Put out milk bottles
2. Put out cat
3. Turn of tv
4. Brush teeth

These activities are unrelated to one another except that they're carried out at a particular time. They are all part of an end-of-day routine. A temporally cohesive module also has some of the same difficulties as a procedurally cohesive one. The programmer is tempted to share code among activities related only by time, and the module is difficult to reuse, either in this system or in others.

```
main( )
{
float time;
printf(“enter time”);
scanf(“%f”,&time);
if (time<12)
{
        printf(“do 1 task”);
        printf(“do 2 task”);
        printf(“do 3 task”);
}
else
{
        printf(“do 4 task”);
        printf(“do 5 task”);
        printf(“do 6task”);
}
}
```

The following example also demonstrates temporal cohesion.

```
class employee:
    def getdata(self):
        self.name=input('Enter name\t:')
        self.age=input('Enter age\t:')
    def putdata(self):
        print('Name\t:',self.name)
        print('Age\t:',self.age)
    def __init__(self):
        self.name='ABC'
        self.age=20
e1= employee()
e1.getdata()
```

e1.putdata()

e2=employee()

e2.putdata()

2.4.6 Logical Cohesion

Another reason of putting the two parts in the same group can be their logical cohesion. This is the ability of the two parts to perform the same logical operation.

Example of Logical Cohesion

Someone contemplating a journey might compile the following list:

1. Go by car
2. Go by train
3. Go by boat
4. Go by plane

What relates these activities? They're all means of transport, of course. But a crucial point is that for any journey, a person must choose a specific subset of these modes of transport. It's unlikely anyone would use them all on any particular journey.
A logically cohesive module contains a number of activities of the same general kind. Thus, a logically cohesive module is a grab bag of activities. The activities, although different, are forced to share the one and only interface to the module. The meaning of each parameter depends on which activity is being used; for certain activities, some of the parameters will even be left blank (although the calling module still needs to use them and to know their specific types).

```
main()
{
 int no.;
printf("Enter1 number for transport");
printf("Enter 2 number for food");
```

```
printf(“Enter 3 number for schools”);
printf(“Enter any number”);
scanf(“%d”,&no);
switch (no)
{
        Case 1: printf(“GO By TRAIN”);
                printf(“GO By CAR”);
                printf(“GO By PLANE”);
                break;
        Case 2:  printf(“SELF MADE”);
                printf(“HALF MADE”);
                printf(“READYMADE”);
                break;
        Case 3: printf(“DAY SCHOLAR”);
                printf(“BOARDING”);
                break;
        Default:  printf(“WRONG NO”);
}
}
```

The following example also demonstrates logical cohesion.

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        int x;
        FILE *fp;
        fp=fopen("data.txt","r");
        printf("\nEnter number\t:");
        scanf("%d",&x);
        printf("\ndata %d",x);
        // from file read number
        getch();
```

2.4.7 Coincidental Cohesion

This is a cohesion which occurs by chance. There is no logical reason for this. However, it may be noted that there cannot be a case where in practical software a module has no cohesion. The following example demonstrates coincidental cohesion.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
        {
        char str[20];
        clrscr();
        printf("\nEnter string\t:");
        scanf("%s",str);
        printf("\nth string is\t:%s",str);
        strrev(str);
        printf("\n%c",str[7]+str[5]);
        printf("\n%f",float(str[4]));
        getch();
```

2.5 TEST CASE PRIORITIZATION TECHNIQUES

Test case prioritization techniques schedule test cases for execution in an order that attempt to increase their effectiveness at meeting some performance goal [120, 84]. Therefore, given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of fault detection of test suites, prioritization of test cases has been done in terms of the extent to which they execute modules that, measured historically, have tended to fail. Alternatively, test cases have to be prioritized in terms of their

increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirement specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than would an adhoc or random ordering of test cases.

The test case prioritization can be done at two levels:

(i) Prioritization for Regression Test Suite

This category prioritizes the test suite of regression testing to be performed. Since regression testing is performed whenever there is a change in the software, so there is need to identify the test cases corresponding to the modified modules and the affected modules with change.

(ii) Prioritization for System test Suite

This category prioritizes the test suite while performing the system testing. Here, the consideration is not the change in the modules. The test cases for the system testing are prioritized based on several criteria: risk analysis, user feedback, fault detection rate, etc.

Based on the above mentioned two levels of test case prioritization, there are various prioritization techniques proposed in the literature. These are discussed in detail in the next sections.

2.5.1 Classification of Test Case Prioritization (TCP) Techniques

A Study was carried out on the TCP techniques that are being used in literature. These techniques were then classified in twelve major types as shown in Figure 2.3. All these TCP techniques are being discussed in the subsequent sections [124,125,126].

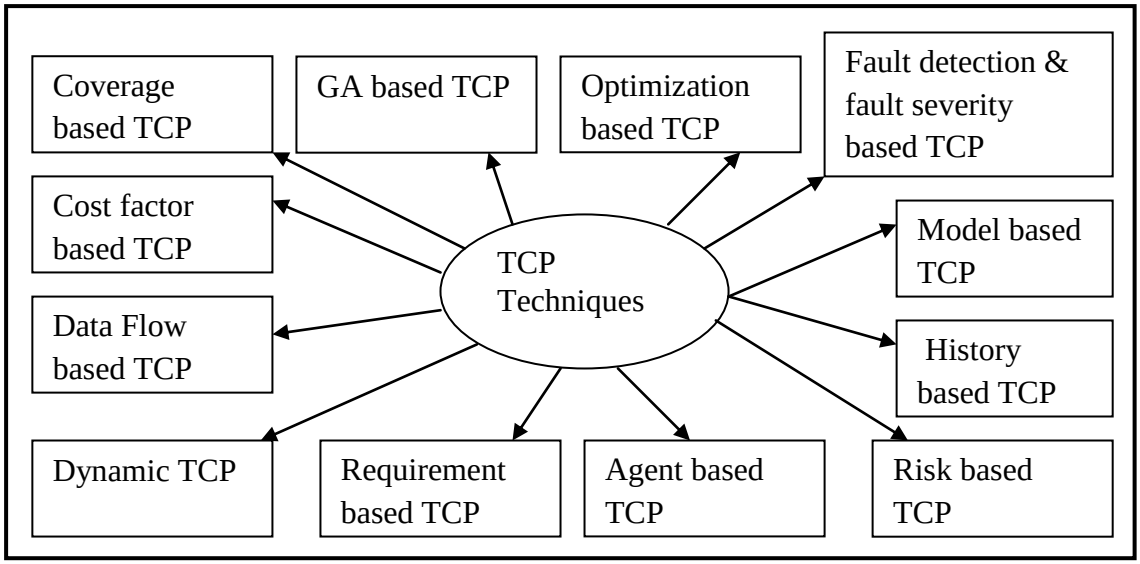


Figure.2.3: Classification of Test Case Prioritization Techniques

1. Coverage based TCP

This type of prioritization is based on the code coverage such as statement coverage, branch coverage, etc. of the test cases. The techniques of prioritization used by various researchers have been formally documented in various primary and secondary studies [5, 120]. Test cases are ordered based on the higher coverage based on the criteria mentioned above. For example, count the number of statements covered by the test cases. The test case with high number of statement covered will be executed first.

Researchers have used Bayesian network to prioritize the test cases of a test suite using coverage based approach [99, 104]. One of the latest work uses requirement weight along with the concept of coverage to accomplish the above task [41]. Other researchers have also used the above concept [14].

Some of the techniques for coverage based TCP (see Figure 2 .4) are discussed below.

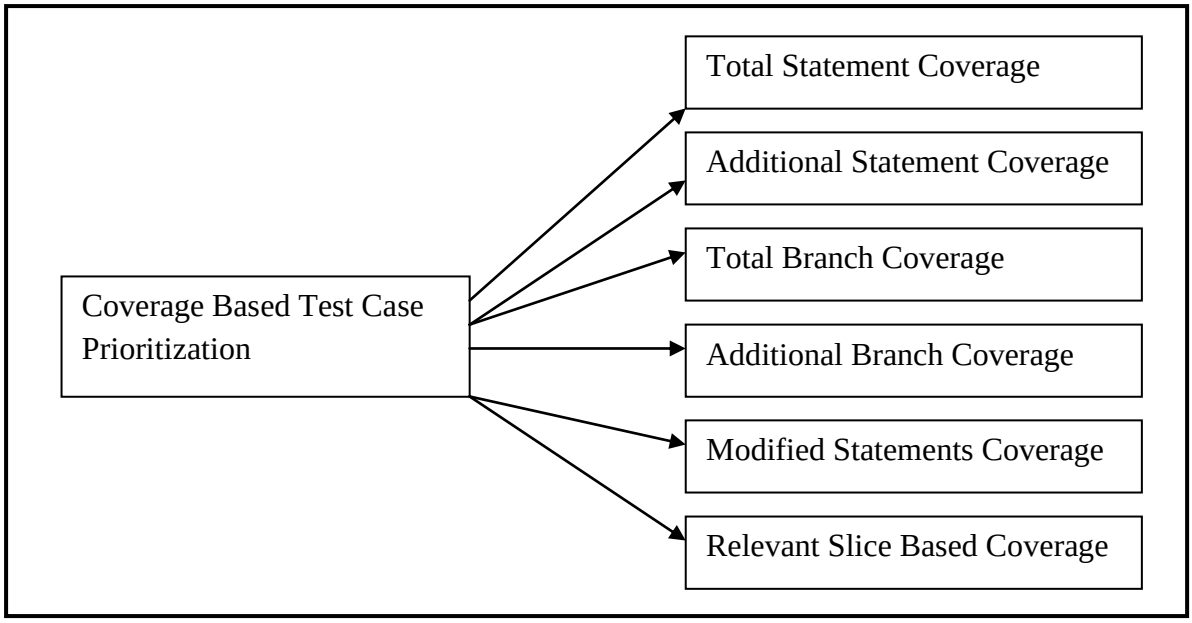


Figure.2.4: Types of Coverage Based TCP

a) Total Statement Coverage Prioritization

This prioritization orders the test cases based on the total number of statements covered by them. Count the number of statements covered by the test cases and order them in descending order of this number. If multiple test cases cover the same number of statements, then a random order may be used. For example, if T1 test case covers 5 statements, T2 covers 3 and T3 covers 12 statements. Then according to this prioritization the order will be T3, T1, T2.

b) Additional Statement Coverage Prioritization

Total statement coverage prioritization schedules test cases in the order of total statements coverage achieved. However, it will be useful if statements are executed that have not yet been covered. Additional statement coverage prioritization iteratively selects a test case T1, that yields the greatest statement coverage, and then selects a test case which covers a statement uncovered by the T1. Repeat this process until all statements covered by at least one test case have been covered.

Table 2.1: Statement Coverage

Statement	**Statement Coverage**		
	Test case 1	Test case2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	X	X	
8	X	X	
9	X	X	

For example, consider Table 2.1, according to total statement coverage criteria, the order is (2, 1, 3). But additional statement coverage select test case 2 first and next it selects test case 3 as it covers statement 4 which has not been covered by test case 2. Thus, order according to addition coverage criteria is (2, 3, 1).

c) Total Branch Coverage Prioritization

In this prioritization, the criterion to order is to consider condition branches in a program instead of statements. Thus, it is the coverage of each possible outcome of a condition in a predicate. The test case which will cover maximum branch outcomes will be ordered first. For example, see Table 2.2. Here the order will be (1, 2, 3).

Table 2.2: Branch Coverage

Branch - statements	**Branch Coverage**		
	Test Case 1	Test case 2	Test Case 3
Entry to while	X	X	X
2-true	X	X	X
2-false	X		
3-true		X	
3-false	X		

d) Additional Branch Coverage Prioritization

Here, the idea is same as in additional statement coverage that first selects the test case with maximum coverage of branch outcomes and then selects the test case which covers the branch outcome not covered by the previous one.

e) Modified Statements based Prioritization

This type of prioritization is based on some priority value assigned to the modified lines of a program covered by a test case.

Amrita Jyoti, Yogesh Kumar Sharma has proposed a model that achieves 100% code coverage optimally during version specific regression testing [9]. The prioritization of test cases was done on the basis of priority value of the modified lines covered by the test case.

f) Relevant Slice based Prioritization

During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original program, except where change is expected. But re-running the test suite for every change in the software makes regression testing a time consuming process. If the portion of the software which has been affected with the change in software can be found out, then prioritization the test cases has been done based on this information. This has been called as a slicing technique [23]. The various definitions related to this technique are given below:

1. Execution Slice

The set of statements executed under a test case is called Execution Slice of the program.

2. Dynamic Slice

The set of statements executed under a test case and have an effect on the program output under that test case is called Dynamic Slice of the program with respect to the output variables.

3. Relevant Slice

The set of statements that were executed under a test case and did not affect the output, but have potential to affect the output produced by a test case is known as Relevant Slice of the program. It contains the dynamic slice and in addition includes those statements which, if corrected, may cause the modification of the value of the variables at which the program failure has been manifested.

If there is change in any statement in the relevant slice, there is a need to rerun the modified software on only those test cases whose relevant slices contain a modified statement. Thus, on the basis of relevant slices, prioritization of the test cases has been done. This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.

Jeffrey and Gupta [23] enhanced the approach of relevant slicing and stated:"If a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case". Thus, they applied the heuristic for prioritizing test cases such that the test case with larger number of statements must get higher weight and will get priority for the execution. A mapping study of the SBSE community in Brazil has used the concept of the relevant slice for the purpose of prioritization. The model structure has been used by some researchers for prioritizing test cases for regression testing [38].

2. Risk based TCP

The Risk based test case prioritization is a well defined process that prioritizes modules for testing [27]. It uses the risk analysis which highlights the potential problem areas, whose failures have more serious adverse consequences. The testers use this risk analysis to select most crucial tests. Thus, risk based technique is to prioritize the test cases based on some potential problems which may occur during the project.

Risk contains two components:

- Probability of occurrence / Fault Likelihood

It tells about how much probability is there of occurrence of the problem.

- Severity of Impact / Failure Impact
  If the problem has occurred, how much the impact is there on the software.

Risk analysis uses these two components by first listing the potential problems and then assigning a probability and severity value for each identified problem as shown in Risk analysis Table (See Table 2.3). By ranking the results in this table in the form of risk exposure, the tester can identify the potential problems against which the software needs to be tested and executed first. For example, the problems in the table given can be prioritized in the order of P5, P4, P2, P3, P1.

Table 2.3: Risk Analysis Table

Problem Id	**Potential Problem**	**Uncertainty Factor**	**Risk Impact**	**Risk Exposure**
P1	Specification Ambiguity	2	3	6
P2	Interface problems	5	6	30
P3	File corruption	6	4	24
P4	Databases not synchronized	8	7	56
P5	Unavailability of modules for integration	9	10	90

3. Fault Detection & Fault Severity based TCP

This type of test case prioritization is based on finding the faults as early as possible and also the impact of these faults. The severity of faults has been also taken into account while prioritizing the test cases of the test suite.

Researchers have proposed numerous test case prioritization techniques to compute average faults discovered per minute [83]. Using APFD (Average Percentage of Fault Detection) metric researchers have demonstrated the effectiveness of their proposed approaches.

Many authors have argued that more effective fault identification at earlier stages of the testing process could be obtained by the using the algorithms for prioritized test cases as compared to non prioritized test cases [87, 88]. The techniques have been successfully applied to the cloud also.

Md. Imrul kayes [77] proposed a new metric for accessing rate of fault dependency detection and an algorithm for prioritizing the test cases. The proposed technique prioritized the test cases with the goal of maximizing the number of faults dependency detection that are likely to be found during the execution of the prioritized test suite.

The techniques which considered the severity of faults early in the testing process, improve the quality of the software [116]. They considered TCP at fault severities in order to have early detection of severe faults in the regression testing process.

4. Requirement based TCP

This technique is used for prioritizing the test cases for system test cases. The system test cases become too large in number as this testing is performed on so many grounds. Since system test cases are largely dependent on the requirements, the requirements can be analyzed to prioritize the test cases. This technique does not consider all the requirements on the same level. Some requirements are important as compared to others [41]. Thus, the test cases corresponding to important and critical requirements are given more weight as compared to others and these test cases having more weight are executed earlier.

The requirements imposed at the beginning of the software development life cycle also helps one to accomplish the task of test case prioritization. During the literature review it was found that many researchers have clubbed together this along with other factors to accomplish the task [36]

Hema Srikanth et al. [36] applied requirement engineering approach for prioritizing the system test cases. It is known as PORT (Prioritization of Requirements for Test). They have considered the following four factors for analyzing and measuring the criticality of requirements:

a) Customer-Assigned priority of requirements: The customer assigns a weight (on a scale of 1 to 10) to each requirement according to the priority which he feels is more important. The higher number is considered as of the highest priority.

b) Requirement Volatility: This is a rating based on the frequency of change of a requirement. The higher change frequency of a requirement is assigned higher weight compared to the stable requirements.

c) Developer-perceived implementation complexity: All the requirements are not equal on the implementation level. The developer having the more difficulty in implementing a requirement is given more weight.

d) Fault proneness of requirements: This factor is identified based on the previous versions of the system. If a requirement in an earlier version of system is having more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for new software.

Based on these four factor values, a Prioritization factor value (PFV) is computed as given below. PFV is then used to produce a prioritized list of system test cases.

$$PFVi = \sum (FVij * FWj) \text{ ----------} \quad (2.1)$$

where FV = Factor value is the value of factor j corresponding to requirement i.

FW = Factor weight is the weight given to factor j.

R. Kavitha & N.Suresh Kumar [92] proposed a method to prioritize the regression testing test cases considering the following factors: (1) customer assigned priority of requirements, (2) Developer-perceived code implementation complexity, (3) Changes in requirements, (4) Fault impact of requirements, (5) Completeness ,(6) Traceability (7) Execution time etc. Based on these factors, a weightage was assigned to each test case in the software. According to the weightage assigned, the test cases were prioritized.

Many authors have [109] proposed approaches for prioritizing the test cases, which ware based on the requirements of the system. The techniques were quite useful in black box environment. The proposed techniques could be of use when source code or

binary code was not available. The main idea was to find the most severe faults early in the testing process and hence to improve the quality of the system according to the customer point of view. A genetic algorithm was proposed for test case prioritization to improve the regression testing. The analysis was done for prioritized and non prioritized tests to prove the effectiveness of the proposed algorithm.

Patric Berander and Anneliese Anfrews [14] considered an approach that provides means to find an optimal subset of requirement resulting in trade of desired project scope against sometime conflicting constraint such as

- Schedule
- Budget
- Resources
- Time to market and quality

They also considered requirement prioritization as the basis of the product strategy.

Maya Daneva and Andera Herrman [6] proposed a conceptual model of requirements prioritization based on benefit and cost prediction. Other researchers have also used the concept for achieving the goal [70].

Siripong Roongruangsuwan and Jirapun Daengdej [108] proposed a new classification of test case prioritization techniques considering a new test case prioritization method along with practical weight factors like test case complexity, dependency and test impact etc.

Thillaikarasi Muthusamy et. al. [114] proposed a technique which prioritizes the test cases based on four groups of practical weight factor such as:

- Customer allotted priority,
- Developer observed code execution complexity,
- Changes in requirements,
- Fault impact,

- Completeness and
- Traceability.

5. Data flow based TCP

Data-flow testing is a white box testing technique. The technique has been used by many researchers to detect improper use of data values due to coding errors [2, 77, 72, 74, 76, 100, 102]. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. The data usage for a variable affects the white box testing and thereby the regression testing. If the prioritization of regression test suite is based on this concept, the rate of detection of faults will be high and critical bugs can be discovered earlier.

J. Rummel et al proposed an approach to regression test prioritization that leverages the all-DUs(definition-use) test adequacy criterion that focuses on the definition and use of variables within the program under test. DU-paths which are variable usage paths are taken for the test cases prioritization [98].

Yogesh Kumar, Arvinder Kaur & Bharat Suri proposed an approach for test case prioritization using DU path as well as DC (definition clear) paths [119, 85]. The idea was that the DU paths which may not be DC may be very problematic as non DC paths may be subtle source of errors.

6. Genetic Algorithm (GA) based TCP

Over several years, organisms are evolving on the basis of fundamental principle "survival of fittest" to accomplish noteworthy results. In 1975, Holland employed principle of natural evolution to optimization problems and built first GA [34, 53, 54]. In GA, a population P = (c1… cm) is formed from a set of chromosomes and each chromosome is composed of genes. The GA populates the population of chromosomes by successively replacing one population with another based on fitness function assigned to each chromosome. The strong individual is included in next

population and individuals with low-fitness are eliminated from each generation [34]. There are two main concepts: crossover and mutation.

- **Crossover**: The crossing over (key operator) is process of yielding recombination of alleles via exchange of segments between pairs of chromosomes. Crossover is applied on an individual by switching one of its allele with another allele from another individual in the population.

- **Mutation:** The mutation is a process wherein one allele of gene is randomly replaced by (or modified to) another to yield new structure .It alters an individual in the population. It can regenerate all or a single allele in the selected individual.

In literature many algorithms based on GA have been proposed that automates the process to prioritize the test suites as per the criteria given to genetic algorithm [9, 90, 109].

Arvinder Kaur, Yogesh Singh et.al proposed a model for prioritizing the test suite on the basis of the complete code coverage [9]. The proposed model achieves 100 % code coverage optimally during version specific regression testing.

Yu-Chi Huang & Chin-Yu Huang [40] proposed a cost cognizant test case prioritization technique which was based on the previous historical records and genetic algorithm. The test costs, fault severities, and detected faults of each test case were gathered from the latest regression testing and then used a GA to find an order with the greatest rate of "units of fault severity detected per unit test cost." The cost-cognizant metric, Average Percentage of Faults Detected per Cost (APFDc), was proposed to evaluate the effectiveness of the cost-cognizant test case prioritization techniques. Others have also used the above concept [88, 92].

7. Optimization based TCP

In this type of test case prioritization the prioritization of test case in a test suite is based on certain optimization algorithm.

Some researchers [10] used the Bee Colony Optimization (BCO) algorithm for the regression test suite prioritization based on the code coverage of the program. The proposed algorithm made effective use of the path construction (exploration) and path structuring (exploitation) phenomenon of scout bees and forager bees for the prioritization of the test suite of the modified code.

Camila Loiola Brito Maia, Thiago do Nascimento Ferreira1 [17] proposed an ant colony optimization based algorithm for prioritizing the test cases considering the precedence of the test cases. Each ant builds a solution, and when it is necessary to choose a new vertex (test case), only allowed test cases are seen by the ant, implementing the precedence constraint of the problem.

8. Agent based TCP

Software agents are autonomous software units which, within their decision space, act independently in order to pursue their predefined goals. Software agents can flexibly interact with the environment and with each other and cooperate through negotiations in order to achieve their goals. Agent-based software systems can reflect the distribution of information, activities, resources or decision processes, as well as different viewpoints or conflicting interests of the concrete problem definition.
In this approach software agents interact and cooperate with each other in order to determine the priority of each test case using information out of the architecture model, out of the available databases and also the information exchanged between each other. The agents have prioritization knowledge, which they use to evaluate the information for prioritizing the test cases.

Cristopz Malz & Peter Gohner [21] presented an Adaptive Test Management System (ATMS) based on software agents which prioritized test cases considering available information from test teams and developments teams about the software system and the test cases. The goal of Adaptive Test Management System was to increase the number of faults found in the available test time with the determined prioritization order.

9. Dynamic TCP

Nilam Kaushik et. al [81] addressed the challenges posed by in-situ changes during the testing process. They introduced the idea of dynamic prioritization in regression testing which uses in-process events to re-order test cases. Dynamic Prioritization uses the most up-to-date pool of test cases and generates a new test case order based on in-process events.

10. Model Based TCP

Component based software often consists of a set of self-contained and loosely coupled components allowing plug-and-play. The components may be implemented by using different programming languages, executed in various operational platforms distributed across geographic distances; some components may be developed in-house, while others may be the third party off-the-shelf components of which the source code may not be available to the developers. So the cost of maintaining the component based software is comparatively more than the maintenance of conventional software system. So when modifying or adding a component and applying the regression testing, incurs more cost and time. So to reduce these two factors, a test case prioritization technique is used which is based on two criteria like maximum state changes and maximum data base access occurred by a test case during component interaction scenario. The test case having maximum state changes and database access given higher priority and executed first so that the debugger will not sit idle as a result, fault will be detected early.

Many researchers have used the concept of Model Based Software testing [13, 59, 100]. Sujata Mohanty, Arup Abhinna Acharya, Durga Prasad Mohapatra [13] proposed a new prioritization technique to prioritize the test cases to perform regression testing for Component Based Software System (CBSS). The components and the state changes for a component based software systems were being represented by UML state chart diagrams which were then converted into Component Interaction Graph (CIG) to describe the interrelation among components. The proposed prioritization algorithm took this CIG as input along with the old test cases and generated a prioritized test suit taking into account total number of state changes and

total number of database access, both direct and indirect, encountered due to each test case. The algorithm was found to be very effective in maximizing the objective function and minimizing the cost of system retesting when applied to few JAVA projects.

11. History based TCP

In this type of test case prioritization the prioritization of test cases is generally based on the test case execution history. During the extensive review carried out, it was found that many researchers have also used the history of a test case as the deciding criteria for the purpose of prioritization. One of the works uses the concept of billattice theory and hence ignores the negative information. This idea of considering the positive information only simplifies the thing and hence leads us to a better, efficient and effective solutions.

It may also be noted that many automata theories have also been used to accomplish the above tasks. One such work [68], uses a specialized automata for History based testing. As per the work this method improves the fault finding capacity of the existing systems.

The development of newer technologies has also helped the cause a lot. Many papers studied during this review also brought forth the fact that many tools and specialized languages have also been developed.

Kim & Porter [68] proposed to use information about each test case's prior performance to increase or decrease the probability that it will be used in the current testing session. This prioritization technique was based on historical data execution. They conducted a series of experiments to assess the effectiveness of the proposed technique on the long run performance of resource constrained regression testing.
Qu et.al [86] proposed a prioritization technique which was applied in the black box testing environment. In this technique the idea was to initialize a test suite using test history and then adjust the order of test cases based on run time information.

12. Cost Factor based TCP

Cost effective based test case prioritization techniques prioritized the test cases based on costs, such as cost of analysis and cost of prioritization. Many researchers have considered the cost of a test case as the deciding criteria for its prioritization. This is referred to as cost factor based test case prioritization. The researchers have gone beyond model based testing to explore the concept in regression testing. Four latest works were studied concerning the topic [70, 4, 38, 39].

Leung and White [70] proposed a cost model for regression test selection. It incorporated various cost of the regression testing. These costs include: the cost of executing the test case, validating the test cases, the cost of performing the analysis to support test selection etc. These costs provided a way to compare the test cases for their effectiveness.

Alexy G. Malishevsky, Gregg Rothermal and Sebastian Elbum [4] has presented a cost model for prioritizing the test cases which takes into account the cost of overlooking faults due to discard tests. They defined the following variables for prioritization the test cases: Cost of analysis, Ca (T) and cost of the prioritization algorithm, Cp (T) .They calculated the weight prioritization value of each test case by the following Formula:

$$WP = Ca(T) + Cp(T) \quad \text{---------- (2.2)}$$

where, WP-is the weight prioritization value for each test case, Ca (T) - is the cost of source code analysis, analysis of changes between old and new versions and, collection of execution traces, Cp (T) – is the actual cost of running a prioritization tool and depending on the prioritization algorithm used.

2.6 CALL GRAPH

Graphs are based on the connections among the software components. Connections are dependency relations also called coupling. A software system can be described by a call graph. It is the most common graph for structural design Testing. A call graph is a directed graph (as shown in Figure 2.5) [120], where vertices represent programs,

classes, or similar program units, and where an arc (v, w), i.e v->w means that a program v calls program w. A call graph is a directed graph D= (V, A), where the vertex set V is the set of programs of the software system and the arc set A= {(u, v) € VXV | u calls v}. A graph has nodes and edges.

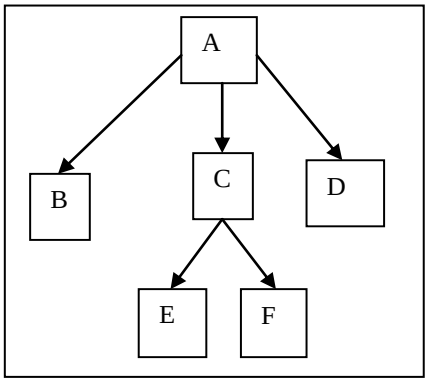


Figure 2.5: Call Graph

Each vertex may have a weight, such as the number of lines of code of the corresponding program. A module contains a subset of the vertices, representing a subset of the programs. The size of a module equals the sum of the vertex weights in the corresponding subset; the size of its interface is the number of vertices which have an incoming arc from a different module. Thus the s/w splitting problem can be formulated as a partitioning problem of a call graph.

NODE COVERAGE……..call every unit at least once (method coverage)
EDGE COVERAGE…….. execute every call at least once(call coverage)

A call graph (also known as a call multigraph) is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge *(f,g)* indicates that procedure *f* calls procedure *g*. Thus, a cycle in the graph indicates recursive procedure calls.

Call graphs are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses, such as an analysis that

tracks the flow of values between procedures. One simple application of call graphs is finding procedures that are never called. All diagrams follow the notation

calling function -> called function

2.7 CONCLUSION

Regression testing is needed when a change is made in the software. It is not possible to rerun all the test cases when some change is made. Therefore it is important to select some test cases out of all the test cases so that the testing time can be reduced and at the same time the fault finding capacity of the test case suite remains the same. There are three ways of doing this. These are prioritization, selection and minimization. These have been discussed in the chapter. Since this work focuses on Test Case Prioritization, this chapter discusses the TCP Techniques in detail along with some other miscellaneous topics required to understand this work.

Chapter III

STRUCURED PROGRAMMING BASED UNIT TEST CASE PRIORITIZATION (SPUTCP): PROPOSED WORK

3.1 INTRODUCTION

White-box testing or structural testing is typically focused on the internal structure of the program. In white box testing, structure means the logic of the program which has been implemented in the language code. Basis path testing is an important part of white box testing. It monitors the whole control structure of the program. Based on the control structure a flow graph is prepared and all the independent paths are covered and executed during testing [109]. An independent path [79] is any path through the graph that introduces at least one new set of processing statements or new conditions. It is considered as a general criterion for detecting more errors as all statements and all branches are covered while testing.

While performing the white box testing there may be large number of test cases executed by the developer to ensure the correct functionality of their code. This process involves a lot of efforts. But if somehow a developer is able to get the prioritized order of the test cases which he/she is going to execute to ensure the correct functionality during the process of white box testing makes the task easier. Keeping this idea in mind a test case prioritization technique for unit testing is proposed in this chapter. The proposed technique is based on the analysis of the source code written by the developers. To show the effectiveness of the proposed approach it is compared with non prioritized and random approach. The APFD value obtained by proposed approach is more, showing the efficacy of the proposed approach.

3.2 SPUTCP TECHNIQUE

To accomplish the above task a new methodology has been proposed [48]. The process starts with the crafting of the Control flow diagram of the program under test. From the control flow diagram, the independent paths are selected. This is followed by the characterization of each node of the graph based on its criticality. Criticality has been decided by various factors of structured programming which have a great potential of introducing the errors in the program. These factors have been decided by conducting a research survey among a group of researchers from both academics and industry having a vast experience of computer programming (See Appendix D). The steps of this process have been depicted in the Figure 3.1.

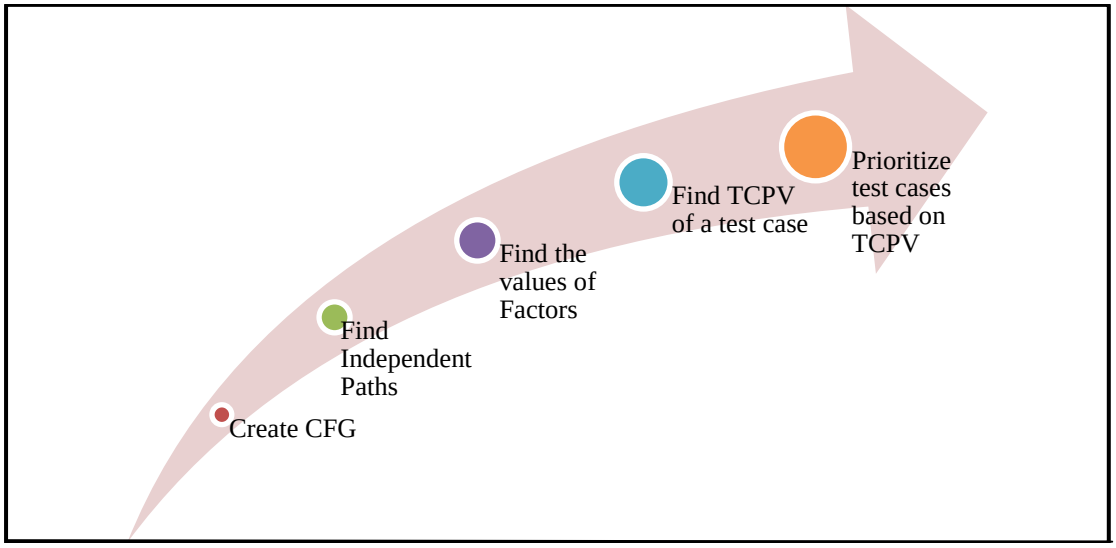


Figure 3.1: Pictorial Representation of SPUTCP Technique

As is evident, the factors form the most important part of the process. Further these factors have been assigned the weights accordingly. These factors are discussed in the next section.

3.2.1 Proposed SPUTCP Factors

The total eight factors have been considered for the purpose of prioritizing the test cases of a test suite. These factors are discussed in next section.

1. The Lines of Code: The lines of code is a metric for software evolution. The line of code is an important evaluation metric. The extensive literature review carried out,

proved the importance of this metric. It has been shown in the evaluation that the path having more lines of code should be assigned more priority in order to have a better APFD value. In order to prove the point some programming examples have been taken for illustration. In the following discussion the example has been elaborated.

The weight of this factor was given as per the suggestions by the experts having experience in development and those who have worked on large software systems.

2. Type Casting: Type casting is required in the program to get the correct results. It may be stated here that this is an important metric, as type casting becomes the source of error in many cases. For example in a modular system if a float is sent to another module and the receiving module takes it as an integer then the final results can remarkably vary from the expected ones. However, it is not always the case that we encounter type casting in each path of the given program.

3. Predicate Statements: The predicate statements diversify a path. Thus, they create more paths and therefore are an important source of inducing an error in the program. During the extensive literature review it was found that this factor has been considered by many researchers to give importance to a path and hence the corresponding test case.

4. File Access: File access is an important factor while running a program. A tester should be highly vigilant of any path that uses file. The failure to check these paths may lead to fatal errors and hence jeopardize the integrity of software. This work, therefore, gives importance to this metric as well. The weight of this factor has been assigned in accordance with the suggestions made by the experts, as stated earlier.

5. Dynamic Memory Allocation: The memory allocation to a pointer, of any type, may lead to severe faults. As it is well known that the memory allocation is of two types: static and dynamic. The dynamic memory allocation leads to more errors and hence should be dealt with care.

6. Number of Input Variables: The input to a procedure also determines the importance of a module and hence the paths generated therein. The inputs can be primitive, complex or even user defined. The different types of input and their number would therefore be treated differently. Moreover, in a procedural program having most of the task in the main module, the input is a greater source of contention as compared to the output.

7. The number of Output Variables: The importance of variables in designing a test case is well known. This work considers these factors important, if not immensely important. The paths selected after the formation of the control flow graph would deal with output variables. These variables and their types increase the importance of path. The concept has been exemplified in the example taken in the following part. The weights of these factors have been assigned by consulting the experts.

8. Assignment Statements: The assignment statements change the values of a variable. These statements are hence important as they can be a source of an incorrect value as well. This work assigns importance to the assignment statements and gives them weight in accordance with the suggestions made by the experts.

All the factors discussed above cannot be at same level. So each factor has been given a weight accordingly. The considered factors and their corresponding weights are shown in Table 3.1. The factors weight shows the criticality of the factor in term of the probability of errors introduced by the factors.

Table 3.1: Prioritization factors and their weight for SPUTCP

S. No.	Prioritization Factors	Factor weight
1	Line of Code	.05
2	Type Casting	.15
3	Predicate Statement	.175
4	File Access	.15
5	Dynamic memory Allocation	.225
6	Number of Input Variable	.1
7	Number of Output Variable	.05
8	Assignment Statement	.1

Test cases are prioritized on the basis of a Test Case Prioritization Value (TCPV) which is determined by using the Formula 3.1. Higher the TCPV of the test case higher is the priority of the test case for execution.

$$TCPV_i = \sum_{j=1}^{8} (vfactor_{ij} * wfactor_j) \ldots\ldots\ldots\ldots\ldots\ldots (3.1)$$

where, the vfactor is value of the factor covered by the ith test case , wfactor is the weight of the jth factor.

The algorithm for the proposed work has been depicted in Figure 3.2.

Begin
Source code (S)
1. Create the control follow graph (CFG) of S.

2. Identify all the independent path of S.

3. Determine the test case covering the independent paths and create the non prioritized list of test cases T.

4. Let T' be the prioritized order of test cases

5. While (T not empty)
Begin

6. Identify the factors covered by the test case corresponding to independent path of test case.

7. Calculate the TCPV of the test case by applying the formula 3.1.

8. Order the test cases in the decreasing order on the basis of the TCPV and let it be T'.

9. T' is the new prioritized order of test cases.

End

10. Execute the test cases in prioritize order

End

Figure 3.2: Algorithm for SPUTCP Approach

3.3 VALIDATION OF SPUTCP APPROACH

For validation of the proposed approach it has been applied on three case studies implemented in C language. The details of the case studies has been presented in Table 3.2

Table 3.2: Case studies for validation of the proposed SPUTCP

S. No	Program Name	Appendix
1	Employee Record	A
2	Saving Module of Income Tax calculator software	B
3	Postfix to Infix Conversion	C

3.3.1 Case Study of Employee Record Software

The first case study is of Employee record software. It is software implemented in C programming language. The considered case study has 154 lines of code and performs various operations such as add a new record, display the records and update records. Before applying the proposed approach some errors have been intentionally introduced in the program. The Control Flow Graph for the case study considered for analysis purpose is shown in Figure 3.3.

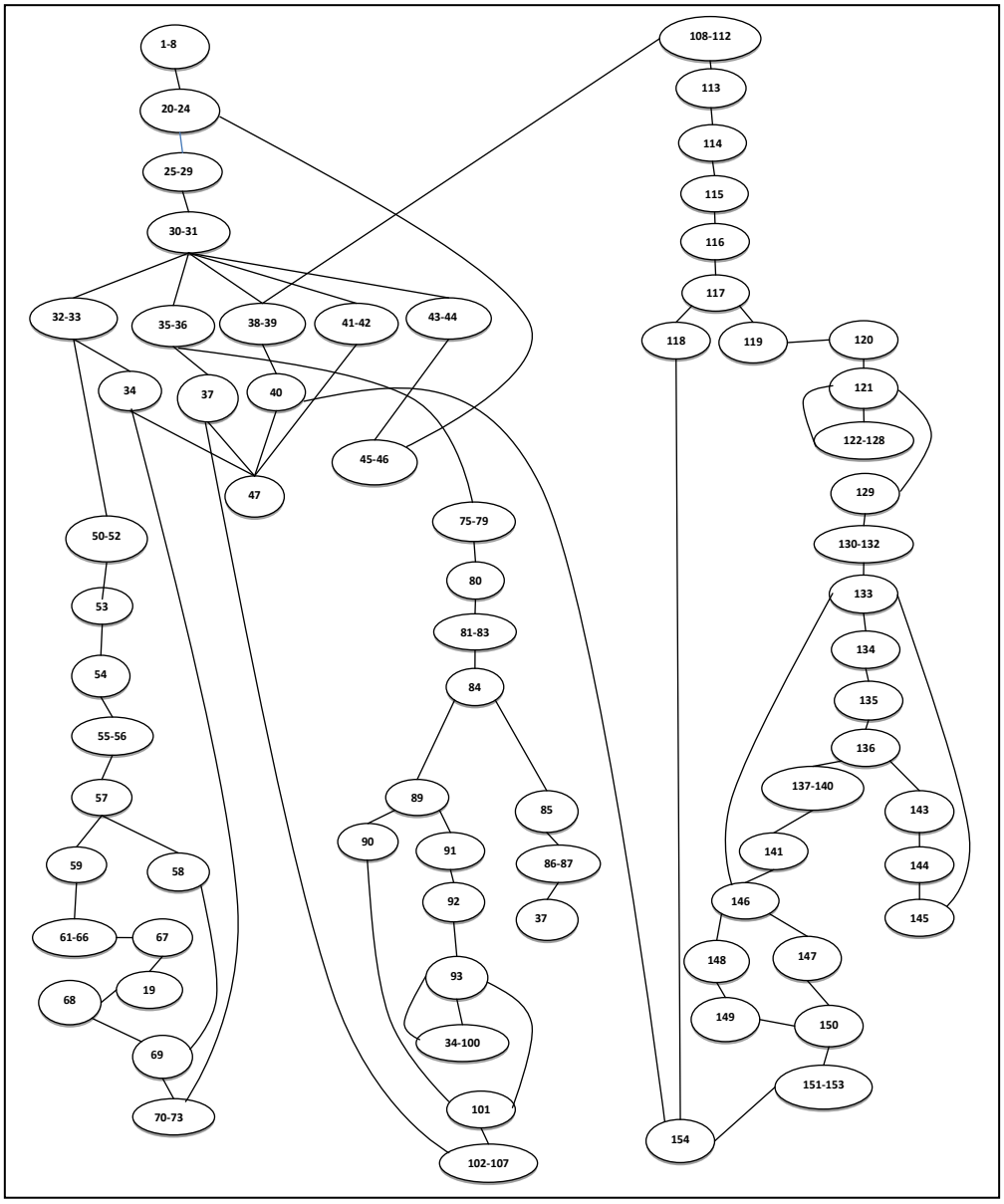


Figure 3.3: CFG of the case study of employee record software

After analysis of the CFG (See Figure 3.3) all independent paths are determined. Table 3.3 shows all independent paths of the case study.

Table 3.3: Independent paths of employee record case study

S.No.	Path No.	Independent path
1	Path1	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-32-33-50-51-52-53-54-55-56-57-58-69-70-71-72-73-34-47
2	Path2	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-32-33-50-51-52-53-54-55-56-57-59-61-62-63-64-65-66-67-19-68-69-70-71-72-73-34-47
3	Path3	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-35-36-75-76-77-78-79-80-81-82-83-84-85-86-87-37-47
4	Path4	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-35-36-75-76-77-78-79-80-81-82-83-84-89-90-101-102-103-104-105-106-107-37-47
5	Path5	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-35-36-75-76-77-78-79-80-14-81-82-83-84-89-91-92-93-94-95-96-97-98-99-100-93-101-102-103-104-105-106-107-37-47
6	Path6	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-108-109-110-111-112-113-114-115-116-117-118-154-40-47
7	Path7	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-108-109-110-111-112-113-114-115-116-117-119-120-121-122-123-124-125-126-127-128-121-129-130-131-132-133-134-135-136-137-138-139-140-141-146-147-150-151-152-153-154-40-47
8	Path8	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-108-109-110-111-112-113-114-115-116-117-119-120-121-122-123-124-125-126-127-128-121-129-130-131-132-133-134-135-136-143-144-145-133-146-148-149-150-151-152-153-154-40-47
9	Path9	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-41-42-47
10	Path10	1-2-3-4-5-6-7-8-20-21-22-23-24-25-26-27-28-29-30-31-43-44-45-46-24-25-26-27-28-29-30-31-41-42-47

Table 3.4 shows the test cases which cover all feasible independent paths for the case study of employee record software.

Table 3.4: Test cases covered and independent paths for employee record case study

Test case id	Value1	Value 2	Result expected	Path followed
TC1	-	-	ERROR	P1
TC2	100	ABC	INSERTED	P2
TC3	-	-	NO RECORD TO DISPLAY	P3
TC4	-	--	ERROR	P4
TC5	-	-	100 ABC 200 XYZ	P5
TC6	-	-	ERROR	P6
TC7	100	MNW	UPDATE SUCCESSFUL	P7
TC8	250	-	ENTER CORRECT ID	P8
TC9	4	-	Exit	P9
TC10	5	-	ENTER CORRECT CHOICE	P10

After selecting the test cases corresponding to the all feasible independent paths the factors are determined covered by the test cases. Out of these test cases TC9 and TC10 are not consider because they do not cover any factors discussed in the proposed approach.

After counting the value of various proposed factors (See Table 3.5), TCPV for each case is calculated by using the Formula 3.1. Table 3.6 shows the TCPV for each test case.

Table 3.5: Count of proposed STUTCP factors present in the case of employee record

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
1	Line of Code	10	17	14	12	21	11	39	32
2	Type Casting	2	2	2	2	2	2	2	2
3	Conditional Statement	1	1	1	2	3	1	8	7
4	File Access	2	3	2	2	4	2	8	7
5	Dynamic memory Allocation	2	2	2	2	2	2	2	2
6	Number of Input Variable	0	2	0	0	0	0	2	2
7	Number of Output Variable	0	0	0	0	2	0	2	2
8	Assignment Statement	3	4	4	4	5	4	8	7

Table.3.6: Calculated TCPV for case study of employee record

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
1	Line of Code	5	8.5	7.0	6.0	10.5	5.5	19.5	16
2	Type Casting	.3	.3	.3	.3	.3	.3	.3	.3
3	Conditional Statement	.175	.175	.175	.35	.525	.175	.875	.875
4	File Access	.3	.45	.3	.3	.6	.3	1.20	1.05
5	Dynamic memory Allocation	.45	.45	.45	.45	.45	.45	.45	.45
6	Number of Input Variable	0	.2	0	0	0	0	.2	.2
7	Number of Output Variable	0	0	0	0	.1	0	.1	.1
8	Assignment Statement	.3	.4	.4	.4	.5	.4	.8	.7
	TCPV	6.525	10.475	8.625	7.8	12.975	7.125	23.425	19.625

On the basis of the TCPV (See Table 3.6), the prioritized order of the test cases is TC7, TC8, TC5, TC2, TC3, TC4, TC6, TC1.

3.3.2 Analysis of the Proposed SPUTCP Approach

For the purpose of analysing the effectiveness of the approach the Average Percentage of Faults Detected (APFD) metric is used [32]. The formula for calculating the APFD is given below.

APFD = 1 – (TF1 + TF2 +TF3 +TF4+.....+ TFm) / (n*m) + 1/ (2*n) -------- (3.2)

where, m is the number of faults and n is the number of test cases.

Faults have been introduced in the program which were exposed by the test cases. APFD values for the random, non prioritized and prioritised order of test cases has been determined and results obtained are encouraging.

Table 3.7 shows the faults detected by the test cases when test cases are executed in non prioritized order.

Table 3.7: Faults detected for non prioritized order of test cases

	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**
F1	*	*						
F2		*						
F3		*						
F4			*		*			
F5					*			
F6						*	*	*
F7							*	*
F8							*	*
F9							*	
F10						*	*	*

Table 3.8 shows the faults detected by the test cases when the test cases are executed in the prioritized order obtain by applying the proposed approach

Table3.8: Faults detected for prioritized order of test cases

	TC7	**TC8**	**TC5**	**TC2**	**TC3**	**TC4**	**TC6**	**TC1**
F1				*				*
F2				*				
F3				*				
F4			*		*			
F5			*					
F6	*	*					*	
F7	*	*						
F8	*	*						
F9	*							
F10	*	*					*	

Table 3.9 shows the faults detected by the test cases when they executed in the random order

Table 3.9: Faults Detected for Random Order of Test Cases

	TC3	**TC5**	**TC8**	**TC4**	**TC1**	**TC7**	**TC2**	**TC6**
F1					*		*	
F2							*	
F3							*	
F4	*	*						
F5		*						
F6			*			*		*
F7			*			*		
F8			*			*		
F9						*		
F10			*			*		*

A. APFD for the non Prioritized approach

APFD = 1- (1+2+2+3+5+6+7+7+7+6) / 80 + 1/(2*8)

= 1-(46/80) +1/(16)

= 49 %

B. APFD for the Random approach

APFD = 1- (5+7+7+1+2+3+3+3+6+3) / 80 + 1/ (2*8)

= 1 – (40/80) +1/(16)

= 56 %

C. APFD for the Proposed approach

APFD = 1- (4+4+4+3+3+1+1+1+1+1) / 80 + 1/(2*8)

= 1-(23/80) +1/(16)

= 78 %

Table 3.10 shows the APFD values obtained when the test cases are executed in non prioritize order, in random order and in prioritized order resulting from applying the proposed approach.

Table 3.10: APFD Values for Various Techniques for employee record case study

S. No	Applied Technique	APFD
1	Non Prioritized	49%
2	Random approach	56%
3	Proposed SPUTCP approach	78%

The comparison of APFD graph of proposed SPUTCP approach, random approach, and non prioritized approach as shown in graph (See Figure 3.4) shows the effectiveness of the proposed approach.

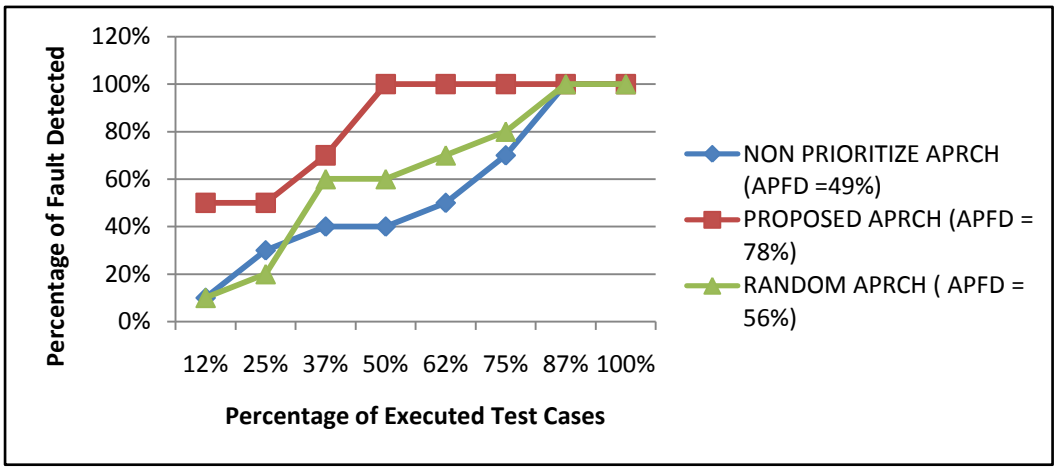


Figure 3.4: Comparison of Proposed SPUTCP, Random and Non Prioritize approach

3.3.3 Case Study of Saving Module of Income Tax Calculator

Another case study is also taken to validate the proposed approach [48]. The source code of this example has been given in Appendix B [80]. The control flow graph of the same is depicted in Figure 3.5.

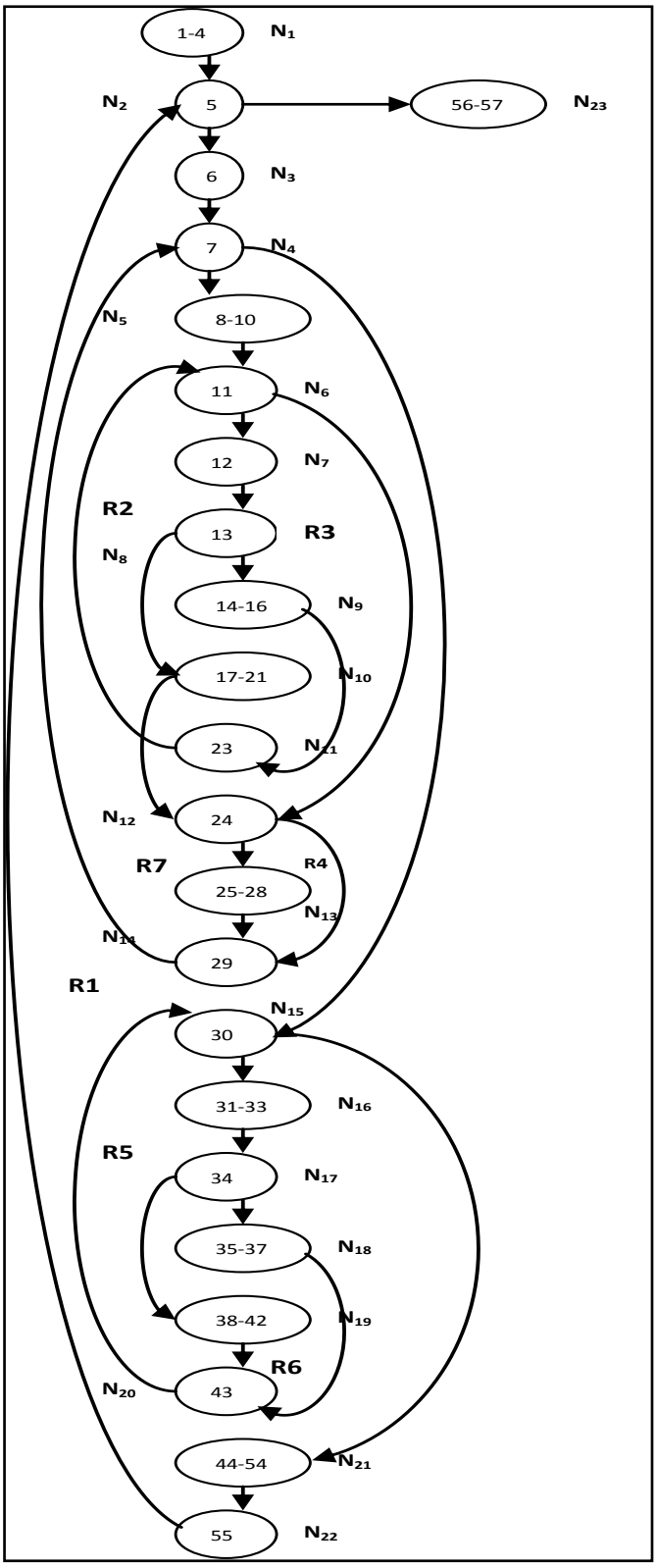


Figure 3.5: CFG for case study of saving module of income tax calculator software

The independent paths obtained from the Control flow graph (See Figure 3.5) are shown in Table 3.11.

Table 3.11: Independent paths for saving module case study

S .No.	Path id	Independent path
1	P1	1-2-3-4-5-56-57
2	P2	1-2-3-4-5-6-7-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
3	P3	1-2-3-4-5-6-7-8-9-10-11-24-29-7-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
4	P4	1-2-3-4-5-6-7-8-9-10-11-13-17-18-19-20-21-24-29-7-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
5	P5	1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-23-11-24-29-7-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
6	P6	1-2-3-4-5-6-7-8-9-10-11-24-25-26-27-28-29-7-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
7	P7	1-2-3-4-5-6-7-30-31-32-33-34-38-39-40-1-42-43-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57
8	P8	1-2-3-4-5-6-7-31-32-33-34-35-36-37-43-30-44-45-46-47-48-49-50-51-52-53-54-55-5-56-57

The various factors covered by the test cases of saving module case study have been shown in Table 3.12 and TCPV for each test case has been shown in Table 3.13.

Table 3.12: Factors Covered by the test cases of case study of saving module

S. No.	Factors covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
1	Line of code	7	22	29	45	46	32	31	29
2	Type casting	0	0	0	0	0	0	0	0
3	Conditional statement	2	4	9	11	11	7	8	7
4	File access	0	0	0	0	0	0	0	0
5	Dynamic memory allocation	0	0	0	0	0	0	0	0
6	No of input variable	0	0	1	1	1	1	0	0
7	No of output variable	1	6	7	10	12	7	6	6
8	Assignment statement	5	8	8	10	10	8	8	6

Table 3.13: Calculated TCPV for case study of saving module

S.No.	Factors covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
1	Line of code	0.35	1.1	1.45	2.25	2.3	1.6	1.55	1.45
2	Type casting	0	0	0	0	0	0	0	0
3	Conditional statement	0.35	0.7	1.575	1.325	1.925	1.225	1.4	1.225
4	File access	0	0	0	0	0	0	0	0
5	Dynamic memory allocation	0	0	0	0	0	0	0	0
6	No of input variable	0	0	0.1	0.1	0.1	0.1	0	0
7	No of output variable	0.05	0.3	0.35	0.5	0.6	0.35	0.3	0.3
8	Assignment statement	0.5	0.8	0.8	1	1	0.8	0.8	0.6
	TCPV	1.25	2.9	4.275	5.775	5.925	4.075	4.05	3.575

From test case prioritization values shown in Table 3.13, prioritized order of test cases is TC5, TC4, TC3, TC6, TC7, TC8, TC2, TC1.

Table 3.14 shows the APFD values obtained when the test cases are executed in non prioritize order, in random order and in prioritized order resulting from applying the proposed approach.

Table 3.14: APFD Values for Various Techniques for case study of salary module

S. No	Applied Technique	APFD
1	Non Prioritized	60%
2	Random approach	52%
3	Proposed SPUTCP approach	77.5%

3.3.4 Case study of Infix to Postfix Conversion

The case study analyzes a program that converts infix expression to that in postfix. The code has been included in the Appendix C of this thesis. The CFG for the same is depicted in Figure 3.6. Total nine test cases have been designed for this case study. Table 3.15 shows the count of factors present in the source code covered by the various test cases. Table 3.16 shows the test case prioritization value for these nine test cases. The prioritization order for these test cases based on TCPV is TC5,TC6,TC9,TC2,TC7,TC3,TC4,TC1,TC8.

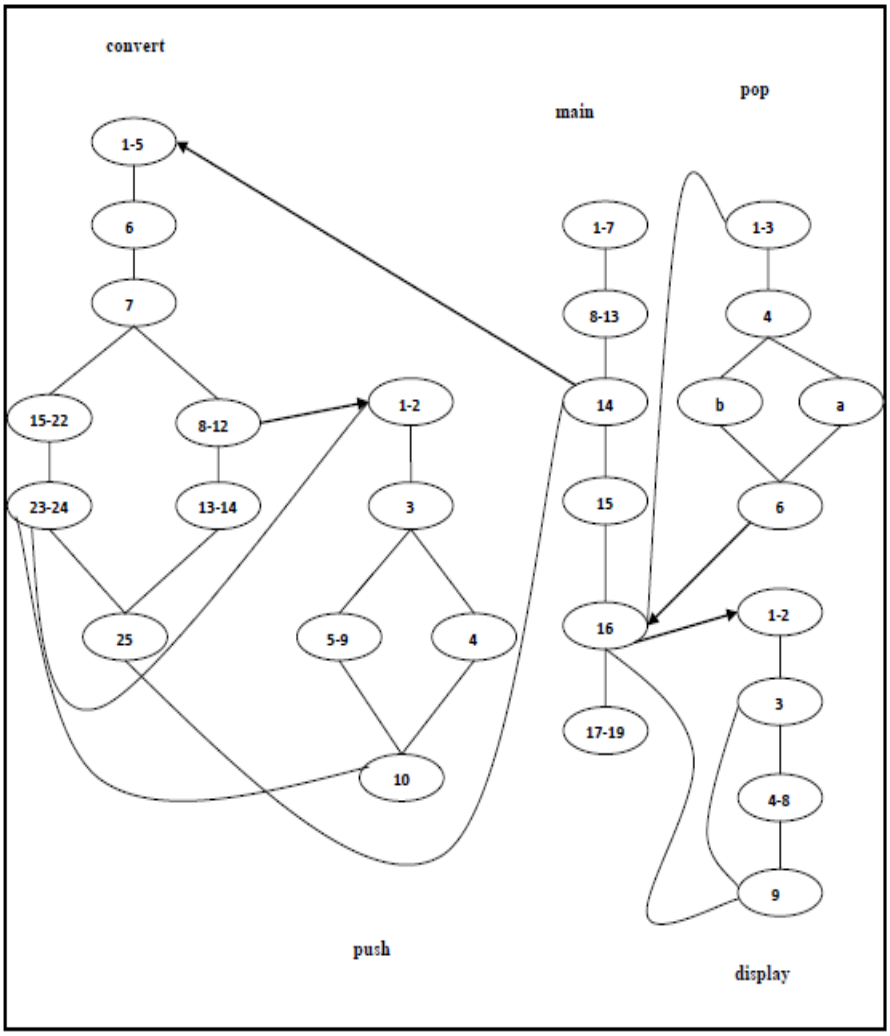


Figure 3.6: CFG for case study of infix to postfix conversion

Table 3.15: Count of factors present in case study of infix to postfix case study

S. No.	Factors covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
1	Line of code	6	8	8	8	14	17	8	3	11
2	Type casting	0	0	0	0	0	0	0	0	0
3	Conditional statement	1	1	1	1	2	1	1	1	0
4	File access	0	0	0	0	0	0	0	0	0
5	Dynamic memory allocation	0	0	0	0	0	0	0	0	0
6	No of input variable	0	0	0	0	0	0	0	0	1
7	No of output variable	1	0	0	0	0	0	1	0	4
8	Assignment statement	0	2	1	0	4	2	0	0	0

Table 3.16: Calculated TCPV for case study of Infix to postfix

S.No.	Factors covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
1	Line of code	0.3	0.4	0.4	0.4	0.7	0.85	0.4	0.15	0.55
2	Type casting	0	0	0	0	0	0	0	0	0
3	Conditional statement	0.175	0.175	0.175	0.175	0.35	0.175	0.175	0.175	0
4	File access	0	0	0	0	0	0	0	0	0
5	Dynamic memory allocation	0	0	0	0	0	0	0	0	0
6	No of input variable	0	0	0	0	0	0	0	0	0.05
7	No of output variable	0.1	0	0	0	0	0	0.1	0	0.4
8	Assignment statement	0	0.2	0.1	0	0.4	0.2	0	0	0
	TCPV	0.575	0.775	0.675	0.575	1.45	1.225	0.675	0.325	1

Table 3.17 shows the APFD values obtained when the test cases are executed in non prioritize order, in random order and in prioritized order resulting from applying the proposed approach.

Table 3.17: APFD Values for Various Techniques for case study of infix to postfix conversion

S. No	Applied Technique	APFD
1	Non Prioritized	52.69%
2	Random approach	48.20%
3	Proposed SPUTCP Approach	77.11%

3.3.5 Case study of Restaurant Management System

The case study is of restaurant management system software taken from the book C projects by Reeta Sahoo[123]. The LOC of the software is 1325.Total 24 test cases have been designed in this case study. After counting the value of various proposed factors, TCPV for each test case is calculated by using the Formula 3.1. Table 3.18, 3.19 and Table 3.20 shows count of proposed factors and Table 3.21, shows the TCPV for each test case.

Table 3.18 Count of proposed SPUTCP factors in case study of restaurant management system

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
1	Line of Code	135	55	56	74	118	42	114	57
2	Type Casting	0	0	0	0	0	0	0	0
3	Conditional Statement	29	3	3	4	17	3	17	5
4	File Access	1	0	0	0	2	1	2	1
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0
6	Number of Input Variable	0	5	5	6	3	3	3	3
7	Number of Output Variable	0	1	1	0	1	1	1	1
8	Assignment Statement	26	9	9	5	12	3	11	5

Table 3.19: Count of proposed SPUTCP factors in case study of restaurant management system

S. No.	Factors	TC9	TC10	TC11	TC12	TC13	TC14	TC15	TC16
1	Line of Code	73	70	28	54	58	187	15	19
2	Type Casting	0	0	0	0	0	0	0	0
3	Conditional Statement	10	12	7	9	9	26	2	3
4	File Access	1	2	1	1	1	8	0	0
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0
6	Number of Input Variable	3	3	3	3	3	3	3	3
7	Number of Output Variable	1	1	1	1	1	1	1	1
8	Assignment Statement	8	12	6	8	8	17	1	1

Table 3.20: Count of proposed SPUTCP factors in the case study of restaurant management system

S. No.	Factors	TC17	TC18	TC19	TC20	TC21	TC22	TC23	TC24
1	Line of Code	82	154	69	13	98	42	86	21
2	Type Casting	0	0	0	0	0	0	0	0
3	Conditional Statement	12	20	11	0	5	2	5	4
4	File Access	2	8	11	0	5	2	5	2
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0
6	Number of Input Variable	3	3	3	0	0	0	0	4
7	Number of Output Variable	1	1	1	0	3	3	3	0
8	Assignment Statement	6	16	4	0	12	8	10	9

Table 3.21: TCPV for test cases of case study of restaurant management system

Sr. No.	Test Case	TCPV
1	TC1	15
2	TC2	4.7
3	TC3	4.8
4	TC4	5.5
5	TC5	11
6	TC6	3.4
7	TC7	10
8	TC8	4.7
9	TC9	6.7
10	T10	7.5
11	TC11	3.7
12	TC12	5.6
13	TC13	5.8
14	TC14	17
15	TC15	1.6
16	TC16	1.9
17	TC17	7.5
18	TC18	14
19	TC19	7.8
20	TC20	0.7
21	TC21	7.9
22	TC22	3.7
23	TC23	7.1
24	TC24	3.4

On the basis of TCPV the prioritized order of test cases is TC14,TC1,TC18,TC5,TC7,TC21,TC19,TC17,TC10,TC23,TC9,TC13,TC12,TC4,TC3,TC2,TC8,TC11,TC22,TC24,TC6,TC16,TC15,TC20.

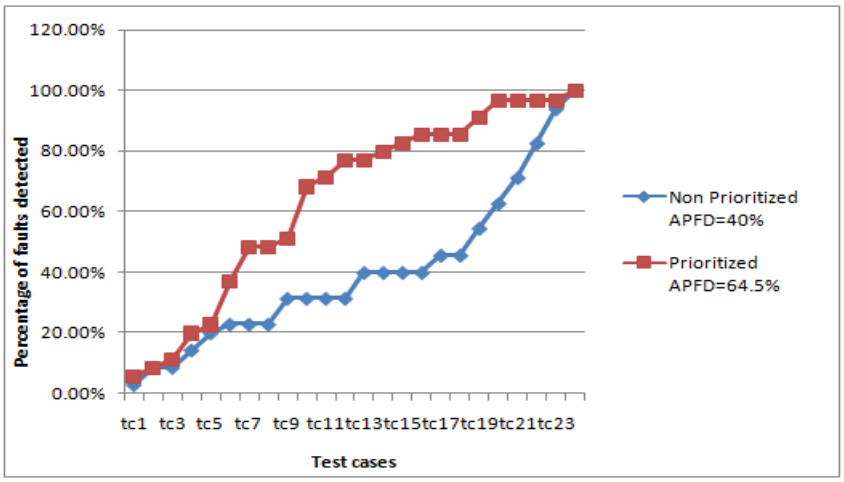


Figure. 3.7 Comparison of Prioritized and Non prioritized approach

Here results show (See Figure. 3.7) that the proposed prioritized approach provide the better APFD.

3.3.6 Case study of Library Management System

Another case study of Library Management System has been taken from the book C projects by Reeta Sahoo[123]. The LOC of the software is 2060. Total 36 test cases have been designed. After counting the value of various proposed factors, TCPV for each test case is calculated by using the Formula 3.1. Table 3.22, 3.23, 3.24 and Table 3.25 show count of proposed factors. TCPV for each test case has been calculated by using the proposed formula.

Table 3.22: Count of proposed SPUTCP factors present in the case study of Library management system

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
1	Line of Code	26	26	51	17	232	42	83	103	147
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	1	1	8	0	10	3	5	6	7
4	File Access	0	0	3	0	2	1	2	1	1
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	0	0	0	3	3	3	3	3	3
7	Number of Output Variable	0	0	3	0	1	1	1	1	1
8	Assignment Statement	15	15	1	8	14	4	8	9	10

Table 3.23:Count of proposed SPUTCP factors present in the case study of Library management system

S. No.	Factors	TC10	TC11	TC12	TC13	TC14	TC15	TC16	TC17	TC18
1	Line of Code	196	270	89	231	133	172	10	144	120
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	8	17	5	12	8	10	2	12	9
4	File Access	2	1	1	1	8	0	0	2	1
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	3	3	3	3	3	3	3	3	3
7	Number of Output Variable	1	1	1	1	1	1	1	1	1
8	Assignment Statement	12	14	3	11	7	9	1	10	7

Table 3.24: Count of proposed SPUTCP factors present in case study of Library Management system

S. No.	Factors	TC19	TC20	TC21	TC22	TC23	TC24	TC25	TC26	TC27
1	Line of Code	176	54	64	84	145	162	61	77	138
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	14	3	5	3	11	13	5	6	11
4	File Access	1	0	2	2	2	0	0	3	0
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	3	0	0	0	0	0	0	0	3
7	Number of Output Variable	1	0	0	1	1	0	0	0	0
8	Assignment Statement	12	2	4	6	10	12	5	6	10

Table 3.25 Count of proposed STUTCP factors present in the case study Library management system

	Factors	TC28	TC29	TC30	TC31	TC32	TC33	TC34	TC35	TC36
1	Line of Code	153	76	66	41	235	92	37	119	107
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	4	3	2	2	8	2	8	8	1
4	File Access	2	1	2	1	1	2	1	1	1
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	3	3	3	3	3	3	3	3	3
7	Number of Output Variable	1	1	1	1	1	1	1	1	1
8	Assignment Statement	6	4	4	1	10	2	1	13	10

On the basis of the TCPV the test cases are prioritized. The prioritized order of the test cases is as follows:

TC11,TC5,TC13,TC32,TC10,TC19,TC15,TC24,TC17,TC23,TC9,TC14,TC27,TC28, TC35,TC18,TC8,TC36,TC7,TC12,TC26,TC33,TC22,TC29,TC21,TC3,T30,TC25,TC 34,TC6,TC20,TC1,TC2,TC31,TC4 and TC 16. The APFD values obtained by applying the proposed approach show the efficacy of proposed approach. Table 3.26 given shows the comparison of APFD values obtained by applying the different approaches of test case prioritization.

Table 3.26: Comparison of APFD Values

Sr. No.	Techniques	APFD
1.	Random	56.80%
2.	Non-Prioritized	65.55%
3.	Proposed SPUTCP Approach	95.41%

The APFD values are shown graphically as Figure 3.8.

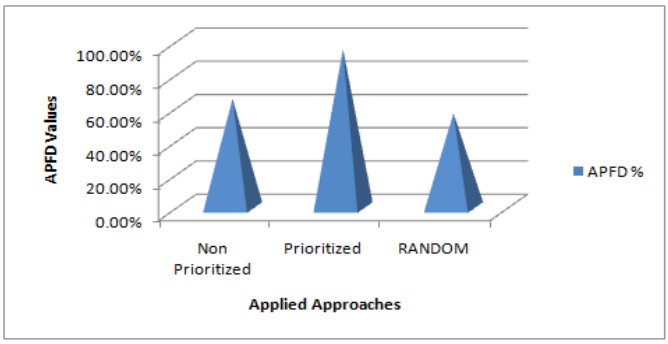


Figure. 3.8: Comparison of APFD Values

3.3.7 Case Study Income Tax Calculator

A case study of gtc () module of the income tax calculator software has been presented for analysis purpose. The LOC of this software is 1161.The two other modules of this software have already been taken for analysis purpose in the thesis.

After counting the value of various proposed factors, TCPV for each test case is calculated by using the Formula 3.1. Table 3.27 shows the count of factors and Table 3.28 shows TCPV for each test case.

Table 3.27: Count of proposed STUTCP factors present in the case of gtc ( ) module

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
1	Line of Code	22	23	22	22	23	22	22	23	22
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	0	0	0	0	0	0	0	0	0
4	File Access	0	0	0	0	0	0	0	0	0
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	1	1	1	1	1	1	1	1	1
7	Number of Output Variable	1	1	1	1	1	1	1	1	1
8	Assignment Statement	1	1	1	1	1	1	1	1	1

Table.3.28: Calculated TCPV for case study of gtc ( ) module

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
1	Line of Code	1.1	1.15	1.1	1.1	1.15	1.1	1.1	1.15	1.1
2	Type Casting	0	0	0	0	0	0	0	0	0
3	Conditional Statement	0	0	0	0	0	0	0	0	0
4	File Access	0	0	0	0	0	0	0	0	0
5	Dynamic memory Allocation	0	0	0	0	0	0	0	0	0
6	Number of Input Variable	.1	.1	.1	.1	.1	.1	.1	.1	.1
7	Number of Output Variable	.05	.05	.05	.05	.05	.05	.05	.05	.05
8	Assignment Statement	.1	.1	.1	.1	.1	.1	.1	.1	.1
	TCPV	1.35	1.40	1.35	1.35	1.40	1.35	1.35	1.40	1.35

On the basis of the TCPV (See Table 3.28), the prioritized order of the test cases is TC2, TC5, TC8, TC1, TC3, TC4, TC6, TC7, and TC9.

For the analysis purpose, some faults have been intentionally introduced in the software which are exposed by the test cases and the APFD's values are calculated. The APFD values obtained by applying the proposed technique for prioritizing the test cases comes out to be 85.5% where as non prioritized order of test cases gives 66.66 APFD. The graph shows (See Figure. 3.9) the comparison of both prioritized and non prioritized approaches.

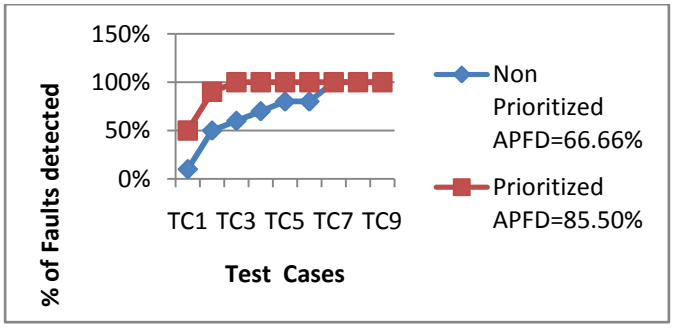


Figure.3.9: Comparison of Prioritized and Non prioritized approach for Case Study of gtc () module

3.4 STRUCURED PROGRAMMING BASED UNIT REGRESSION TEST CASE PRIORITIZATION (SPURTCP) APPROACH

While performing unit testing, it may be the case that the unit has been tested but there may be changes in the tested unit. Therefore, there is a need to perform the regression testing. To reduce the number of test cases during the regression testing, the same factors have been taken into account for the purpose of prioritizing the test cases.

3.4.1 Explanation of the Process of SPURTCP

The proposed approach [52] starts with finding the TCPV of the test case before modifying the program and after the modification is calculated using the Formula 3.1.After that difference between the test case prioritization value (DTCPV) of the test cases before the modifications and after modification is determined. The DTCPV can be calculated by using the Formula 3.3 which is given below.

$$DTCPV_i = \forall_{i=1}^{n}(TCPV'_i - TCPV_i) \quad \text{.....(3.3)}$$

where n is number of test cases, TCPV' is the test case prioritization value of ith test case after modification and TCPV is the fitness value of the ith test cases before modification.

The DTCPV of the test cases before and after the modifications shows that the path has been changed. There may be some considered factors which have been either added or deleted while modifying the code. The DTCPV of test cases may be positive or negative value. Both the value shows that the code covered by the test cases has been changed. The test cases are prioritized on the bases of DTCPV. Higher the value of DTCPV of a test case means that more changes have been taken place in the code covered by that test case, so higher chances of the errors in the corresponding path covered by test case. If the DTCPV for two test cases comes out to be zero, it shows that no changes have been occurred in the code. So there is no need to execute the corresponding test case during the regression testing process, therefore reducing the number of test cases. The algorithm for this approach is shown below in the Figure 3.10.

Let T be the list of test cases ,P is original program, P' is modified program, TCPV is the prioritization value of test case before the modification, DTCPV is after the Modification and T' is the list of prioritized test cases .
Begin
1. If P
While (T not empty)
Begin
2. Determine the prioritization value TCPV of the test cases by using the formula 3.1.
End
3. If P'
While (T not empty)
Begin
4. Determine the prioritization value of the test cases using the formula 3.1.
End
5. Determined the difference between the prioritizations values (DTCPV) of the test cases before the modification and after modification

$$DTCPV = TCPV' - TCPV$$

6. Orders the test cases in the decreasing order on the basis of the DTCPV.
7. Create a list T' of the prioritized test cases.
Execute the test cases in the prioritizing order.
End

Figure 3.10: Algorithm for SPURTCP approach

3.4.2 Validation of the Proposed SPURTCP Approach

For experimental evaluation and analysis, the proposed approach has been applied on case study of employee record considered in section 3.3 of this chapter. The case study is software which performs various operations like add records, display record and update records of employees. This case study has 154 lines of code. For applying the approach the case study has been modified. The case study is modified by adding some new variable or some considered factors proposed in previous section. Modified case study has 184 lines of code.

After the modification in the case study of employee record, the count of factors considered for the purpose of test case prioritization is determined and are shown in Table 3.29.

Table 3.29: Count of factors present in modified Case study of employee record

Factors	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**
Line of Code	12	24	17	15	27	14	52	39
Dynamic memory allocation	03	03	03	03	03	03	03	03
Type Casting	03	03	03	03	03	03	03	03
Assignment	04	05	04	04	05	05	11	08
File Access	02	06	02	02	06	02	12	09
Conditional Statement	1	1	1	02	03	01	05	05
Number of input variable	0	4	0	0	0	0	4	4
No. of output variable	0	0	0	0	4	0	4	4

The prioritization values of each test case are calculated using the formula 3.1. Table 3.30 shows the TCPV after modification in the employee record case study.

Table 3.30: TCPV of test cases for modified employee record case study

Factors	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**
Line of Code	6	12	8.5	7.5	13.5	7	26	19.5
Dynamic memory allocation	.675	.675	.675	.675	.675	.675	.675	.675
Type Casting	.45	.45	.45	.45	.45	.45	.45	.45
Assignment	.4	.5	.4	.4	.5	.5	1.1	.8
File Access	.3	.9	.3	.3	.9	.3	1.8	1.35
Conditional Statement	.175	.175	.175	.35	.525	.175	.875	.875
Number of input variable	0	.4	0	0	0	0	.4	.4
No. of output variable	0	0	0	0	.2	0	.2	.2
TCPV	8.0	15.1	11.2	9.675	16.75	9.1	31.5	24.25

A) Determining DTCPV and Prioritizing Test Cases

DTCPV of test cases is calculated by using the formula 3.3 and prioritized order of the test cases is obtained on the basis of the DTCPV. Higher the DTCPV of the test case highest is the priority of the test case. Table 3.31 shows the DTCPV for all test cases.

Table 3.31: DTCPV for the test cases

S. No	Test Case ID	Finding of DTCPV
1	TC1	DTCPV = 8.0-6.525 =1.475
2	TC2	DTCPV-15.1- 10.475 =4.525
3	TC3	DTCPV=11.2- 8.625 =2.575
4	TC4	DTCPV=9.675 – 7.8 =1.875
5	TC5	DTCPV=16.75 – 12.975 =3.78
6	TC6	DTCPV=9.1-7.125 =1.975
7	TC7	DTCPV=31.5-23.425 =8.075
8	TC8	DTCPV=24.25-19.825 =4.425

From the Table 3.31 the prioritized order of the test cases is TC7, TC2, TC8, TC5, TC3, TC6, TC4, TC1.

3.4.3 Analysis of the Proposed SPURTCP Approach

To analyse the effectiveness of the proposed work, it is applied on the case study of the program. The faults exposed by the test cases when executed in non prioritized order are given in Table 3.32.

Table 3.32: Execution of test cases in non prioritized order

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
F1	*	*						
F2		*						
F3		*						
F4			*	*	*			
F5					*			
F6			*	*	*			
F7						*	*	*
F8							*	*
F9							*	
F10							*	
F11							*	*

Table 3.33 shows the faults detected when the test case are executed in the prioritized order obtained by the proposed SPURTCP approach.

Table 3.33: Execution of test cases in prioritized order

	TC7	TC2	TC8	TC5	TC3	TC6	TC4	TC1
F1		*						*
F2		*						
F3		*						
F4				*	*		*	
F5				*				
F6				*	*		*	
F7	*		*			*		
F8	*		*					
F9	*							
F10	*							
F11	*		*					

Table 3.34 shows the faults exposed by the test cases when these are executed in random order.

Table 3.34: Execution of test cases in random order

	TC7	**TC2**	**TC8**	**TC5**	**TC3**	**TC6**	**TC4**	**TC1**
F1				*				*
F2				*				
F3				*				
F4			*		*	*		
F5			*					
F6			*		*	*		
F7	*	*					*	
F8	*	*						
F9	*							
F10	*							
F11	*	*						

Table 3.35 shows the APFD value of the non prioritized approach, random approach and the prioritized approach.

Table 3.35: APFD Values for various approaches for modified employee record case study

S. No.	Applied Technique	APFD
1	Non Prioritized Approach	59%
2	Random Approach	73%
3	Proposed SPURTCP Approach	80%

Figure 3.11 shows the comparisons of the non prioritized approach, random approach and the proposed approach.

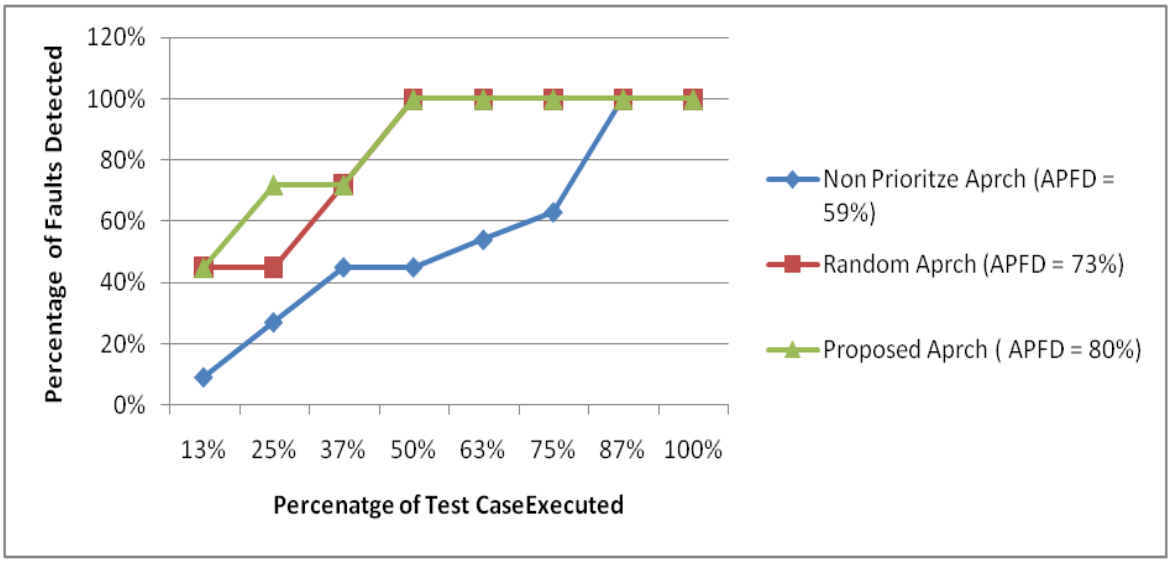


Figure 3.11: Comparison of APFD values of different prioritization techniques

3.5 CONCLUSION

This chapter focuses on the discussion of proposed test case prioritization technique for unit testing based on analysis of the structure of program considering certain factors which have a great potential of inducing an error in the code of software. The proposed technique has also been extended to prioritize the test cases for regression testing. The proposed approach has been applied to certain case studies. The results obtained are encouraging.

Chapter IV

HIERARCHICAL SYSTEM TEST CASE PRIORITIZATION (HSTCP): PROPOSED WORK

4.1 INTRODUCTION

Test case prioritization techniques organize the test cases in a test suite by ordering in such a manner that the most critical test cases are executed first thereby increasing the effectiveness of testing. The prioritization techniques [50] provide a way to find out more bugs under resource constrained environment and thus improve the reliability of the system quickly. Moreover, as faults are revealed earlier, software engineers have more time to fix the bugs and adjust the project schedule System Testing encompasses a large number of test cases, which may not be able to get executed due to constrained time, budget and limitation of the resources. Therefore, the test cases must be prioritized in some order such that the critical and most required functionality can be tested early. Researchers have proposed prioritization techniques based on requirements [101]. In this chapter, a hierarchical approach for system test case prioritization based on requirements has been proposed that maps requirements on the system test cases. This approach analyzes and assigns value to each requirement based on a comprehensive set of twelve factors thereby prioritizing the requirements. Further, the prioritized requirement is mapped on the highly relevant module and then prioritized set of test cases.

Many prioritization techniques have been proposed for prioritizing the system test cases on the basis of requirements. However, the requirements only in consideration cannot include critical test cases. The implementation complexity and test case complexity may also effect the test case prioritization. Though Hema Srikanth [36] has included the developer perceived complexity for implementation factor but it is only a scaling assigned by developer explicitly. There may be lot of complexities and issues in design and code of the mapped requirements. All these factors should also be considered while prioritizing the test cases. The researchers have also considered,

fault proneness of requirements only in connection with customer-reported failures. But, there is need to consider fault-proneness for every requirement with every affected factor. Moreover, the fault proneness associated with mapped code should also participate in prioritizing the test cases.

In this chapter, a hierarchical system test case prioritization (HSTCP) approach [45, 49] is proposed wherein the prioritization process is performed at three levels given below:

(1) The requirements are first prioritized on the basis of twelve factors by assigning a priority weight age to each requirement.

(2) The highest priority requirements are then mapped to their corresponding modules to get prioritized modules.

(3) The test cases based on to the highest prioritized module are then put in order for execution.

The chapter has been organized as follows. The second section of the chapter presents the proposed work, the third section presents the methodology and the forth section presents the analysis of the proposed system. The fifth section presents the implementation and the last section presents the conclusion.

4.2 PROPOSED HSTCP APPROACH

The proposed approach starts with analyzing and assigning value to each requirement based on a comprehensive set of twelve factors thereby prioritizing the requirements [45]. After getting the ordered list of requirements, a mapping between the highest priority requirement and its corresponding modules is performed. The modules are then prioritized based on cyclomatic complexity and non dc path. The weighted prioritized module is then selected for testing. It may be possible again that there are several test cases corresponding to this selected module. For this purpose, the third level of prioritization is applied by prioritizing these several test cases based on four factors. In this way HSTCP technique is proposed and discussed in subsequent

sections. In the proposed prioritization process almost every stakeholder viz. the customer, developer, tester, and business analyst participate. The prioritization process is shown in Figure. 4.1.

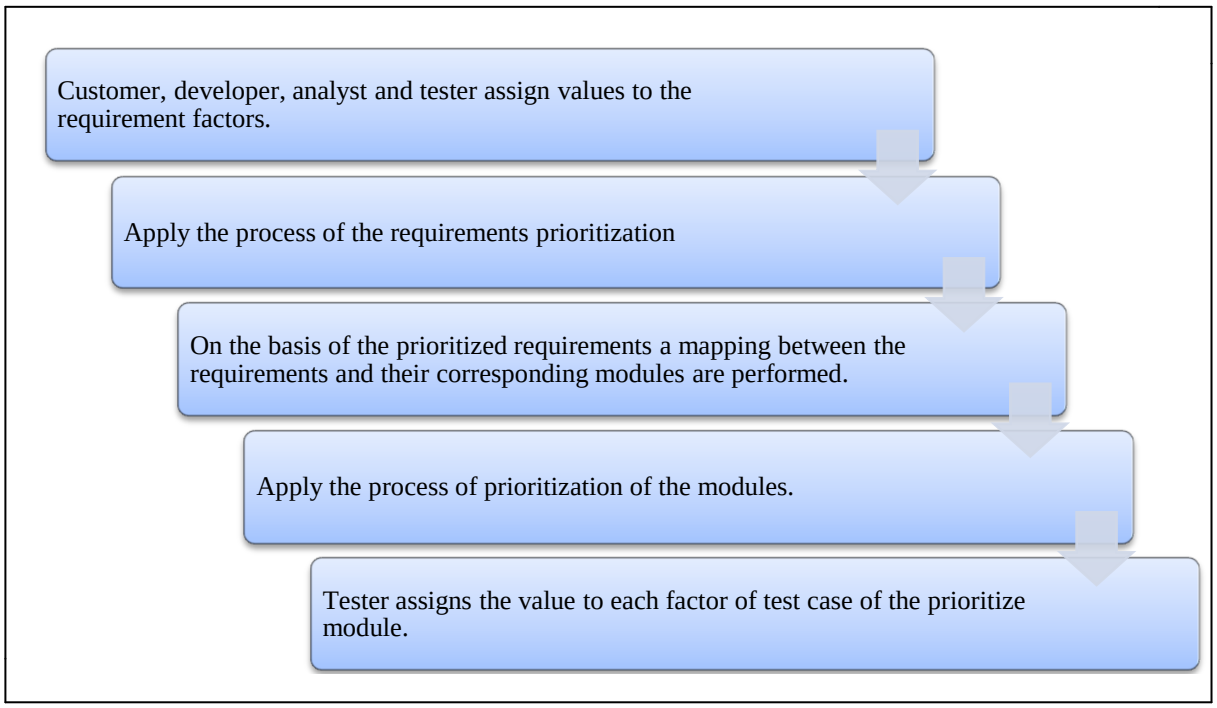


Figure 4.1: Hierarchical System Test Case Prioritization (HSTCP) Technique

4.2.1 Prioritization of Requirements

A critical review of the work done by the researchers discussed in chapter 2, in the direction of system test case prioritization indicates that the following factors have not been considered that may affect the system test case execution:

- **Developer assigned priority:** The developer may assign the priority to every requirement on the basis of its importance.

- **Show Stopper Requirements:** These are the critical requirements in the absence of which the software may not work. The developer may therefore assign the priority to these types of requirements.

- **Frequency of Requirements:** It is the frequency of a requirement how much it is being used in the software.

- **Expected fault:** The developer may analyze the causes which may make the software error prone.

- **Implementation Complexity:** It is the criteria how much each requirement is difficult to implement considering technology dependency, interdependency of the requirements, complexity of requirement itself, etc.

- **Cyclomatic Complexity:** It is the logical complexity of a program [80]. The module with higher complexity may lead to complex test cases.

- **Non DC path:** In data flow graph of a program, the non-dc paths which are the path between the definition node and the usage node of the variable wherein the variable is defined more than once are the problematic areas with respect to the use of a variable [80]. Therefore, this factor may also be considered for module prioritization.

Considering these shortcomings, in this work, a comprehensive list of 12 factors has been identified. There are the various factors on the basis of which process of prioritization of requirements is performed. These factors are in accordance with every phase of SDLC. All these factors have been assigned a priority value between 0 to 10.These priority values are assigned by various stakeholders of the project. Table 4.1 shows these factors.

Table 4.1: Factors considered for requirement prioritization

Sr. No.	**Factors**	**Phase of SDLC**	**Priority value assigned by**
1	Requirement Volatility	Requirement Analysis	Customer
2	Customer Assigned Priority	Requirement Analysis	Customer
3	Implementation Complexity	Design	Developer
4	Fault Proneness of Requirements	Design	Developer
5	Developer assigned priority	Requirement Analysis	Developer
6	Show Stopper requirements	Design	Developer
7	Frequency of execution of requirement	Requirement Analysis	Developer
8	Expected Faults	Coding	Developer
9	Cost	Requirement Analysis	Analyst
10	Time	Requirement Analysis	Analyst
11	Penalty	Requirement Analysis	Customer
12	Traceability	Testing	Tester

1. Requirements Volatility

Requirement volatility is the frequency of changing a requirement during development cycle of the software.

Reasoning: Most of errors are found during the requirements gathering and analysis phase. If the developers implement the particular requirement and that requirement changes then developer has to redesign and re-implement the same. Due to reimplementation of requirement it also increases the fault density in the programs. Studies show that 35 % of the requirements for an average project change before project completion. The requirement with a higher change frequency is assigned a higher priority value as compared to the stable requirements.

2. Customer Assigned Priority

Based on the priority of the requirement, the customer assigns a priority value to each requirement.

Reasoning: Several studies indicate that some requirements of a project are frequently used and some are rarely used. The studies show that approximately half of the software functions are never used. Only 36 % of the software function is always used and most of the faults lie in these functions which are frequently executed. So the customer is involved to know which requirements are very important to him so that these are tested earlier to increase the customer satisfaction. Customer assigns the highest weight to requirement which is very important for him.

3. Implementation Complexity

Each requirement may be analyzed according to how difficult it is to implement. There are various factors considered during requirement implementation. So before assigning a priority value to this factor it is necessary to consider all factors related with that requirement. The priority value for this factor is the sum of the priority values assigned to these factors. There are 3 factors which are taken into consideration as shown in Figure 4.2. These 3 factors are discussed below.

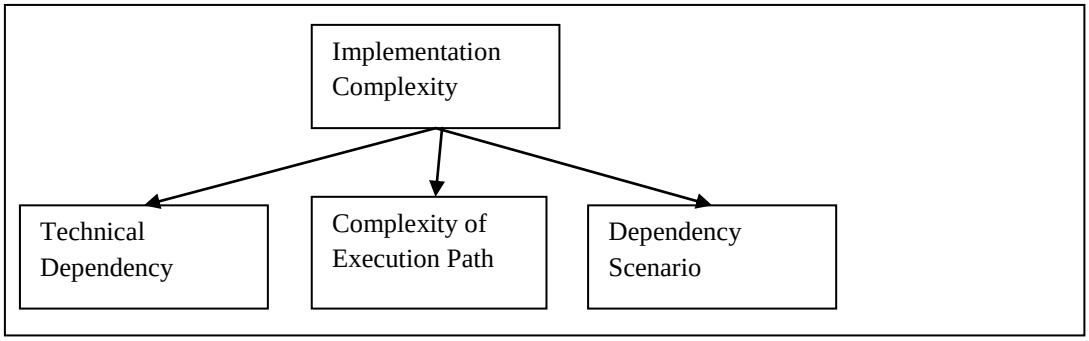


Figure 4.2: Implementation Complexity factors

Reasoning: The studies show that more complex is the requirement, more it tends to have faults. So a priority value is assigned by the developer to this factor.

- **Technical dependency:** Technology plays a very important role in development of any software. Implementation technique of software is varying from technology to technology. With the selection of suitable technology developer can develop less error prone project within target time and budget. Some time the customer bounds developer to a particular technology. Sometimes the proposed requirements are very complex to implement in selected technology whereas the same requirement can be implemented in other technology without the much complexity and less error. So this factor is considered for prioritizing the test cases. For this factor a priority value between 0 and 3 is assigned.

- **Complexity of execution path:** Sometimes in the project a requirement is very simple to implement thereby its complexity is very low. But to execute that requirement user have to follow the complex path of the execution. So the long path of execution also affects the complexity of requirement .This factor assigned a priority value between 0 and 3.

- **Dependency scenario:** The studies show that more the dependency between the modules of a requirement higher is its complexity. It means if a requirement is covered by more than one modules and the dependency among these modules is high then higher is the complexity of that requirement. For this factor a priority value between 0 and 4 is assigned.

4. Fault Proneness of Requirements

Fault proneness signifies those requirements which are associated with faults or which shows failures in the previous releases of the software. If a requirement in an earlier version of the system has more bugs, then this requirement in the current version is given more weight.

Reasoning: Fault proneness factor is important because the requirements which have shown failures in the earliest release are more faults prone. So it is important to give more weight to requirements with high fault proneness so that they can be tested on higher priority. This factor is valid for only those requirements which have been implemented in earlier version of software and not valid for the new requirements. So a priority value is assigned accordingly.

5. Developer – Assigned Priority

Developer assigns the priority to every requirement on the basis of the importance of the requirement. Developer assigns the priority value to each requirement ranging from 0 to 10.

Reasoning: Developer plays an important role for successfully completion of a project within target time and budgeted cost. Studies show that more than 50% project are not completed in the target time and cost. Here the developer analyzes each requirement and assigns the weight to each requirement on the basis of that requirement how much it is important for the project. It may happen that lowest priority given by the customer to a particular requirement is very important for the project. So the developer gives a weight to each requirement on the basis how much it contributes towards the success of the project. Larger value of the weight given to a requirement shows it is very critical to the project.

6. Show Stopper Requirements

Show stopper requirement are those requirements on the basis of which software works. Such requirements are given more importance and assigned the priority value accordingly.

Reasoning: In every project there are some core requirements on the basis of which all modules are working .If these requirements are failed then whole project will stop. For example, consider online ticket booking website. By using website user can inquire about the train, see the available seats in a particular train, cancel out ticket, online payments to tickets and book tickets. These are the requirements which are frequently used. Suppose for a moment the online payment system fails, In this case users are not able to book the ticket until customer has not paid for the tickets. So here the online payment system is critical requirement. There may be more than one requirements on which the whole project works.

7. Frequency of the Execution

In this factor priority value to each requirement is assigned on the basis of its execution frequency. The more priority value is assigned to the requirements which are frequently used.

Reasoning: In every project there are some requirements which are never executed in product and some requirements are frequently executed. The requirement may be executed directly or may be through the other requirements. Therefore a priority value is assigned to them on the basis of their frequency of execution. Consider online ticket booking website. By using website user can inquire about the train, see the available seats in a particular train, and cancel the tickets, make online payments to tickets and book tickets are those requirements which are being frequently used. But update the fair of tickets, update the timings of the trains are those requirements which are not frequently used.

8. Expected Fault

This factor identifies the future implementation faults. In this factor developer analyzes the causes which make the software error prone.

Reasoning: The study shows that it's not possible to implement software without faults. The reason that may be responsible for generating the fault should be considered. As the studies show if developer analyzes the fault in the initial phase then the project will be successfully completed within the time and the budget. The two factors that we are using are shown in Figure 4.3.

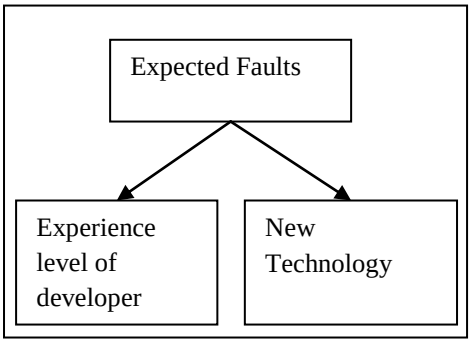


Figure 4.3: Expected Faults

- **Experience level of developer**:

  The study shows that skills and experience of a developer play an important role in success full completion of a project. Lower is the experience of a developer more is the chance of getting a bug in the implementation of a particular requirement. A developer with lower experience may implements a requirement with higher complexity whereas the experience developer implements same requirements with less error. For this type of requirement a weight between 0 and 5 is assigned.

- **New Technology**

  Sometimes customer bounds the developing team to use a particular platform to implement their requirement then if that particular technology or platform is never used by the developer then to work on the new platform is difficult for the developer. So there are more chances of bug in the requirement so a priority value between 0 and 5 is assigned. Higher value is assigned for the

very new technology which is never used and medium value which has been used in previous projects and zero value for our current technology used by the developers.

9. Cost

It corresponds to expenditure done to implement the requirements. Here a cost factor is considered for each requirement and a weight between 1 and 10 is assigned. The higher value being assigned to the cost factor shows that cost to implement the requirement is very high.

Reasoning: The software development cost is difficult to predict. The study shows [16] that 45 % projects complete with overrunning the cost. There are many factors which influence the cost of requirements. These factors are: complexity of a requirement, the ability to reuse of the code, amount of testing and the documentation.

Generally the cost is expressed in the term of the staff effort since for the implementation of a product new persons should be hired, trained them, buy new resources, new tools. The cost in software development is related to the number of hours spent by the staff for the implementation of the product. The implementation cost is usually estimated by developing organization.

10. Time

This factor is the most critical factor in software development cycle. Since in every organization there is pressure to complete the product with in specified time. So the time for every requirement is estimated and assigned the priority value accordingly. A higher value of time factor indicates that it takes higher staff hours to complete the requirements.

Reasoning: In software industry on every product there is constraint to complete a product with in time. Time in software developments is related with number of staff hours. The development time of requirements is influenced by many factors such as degree of parallelism in development, train the staff, need to develop support

infrastructure. Time is directly related with the cost. The more is the time to develop the requirement the more is the cost to implement the requirement.

11. Penalty

It is the punishment imposed on organization if they are not able to deliver the complete product within budget in the specified time. Penalty is critical factor in development of a requirement. This factor shows the penalty associated with each requirement. The higher value of penalty shows that they incur a high penalty if failed to deliver the right product. Here a weight between 1 and 10 is assigned.

Reasoning: In software development process it may be possible that a low priority requirement incurs high penalty if the developer fails to complete the requirement. Penalty factor is associated with each requirements .It also increases the quality of product. If a requirement is not fulfilled then it is possible to evaluate the penalty corresponding to that requirement. High penalty value means high penalty of that requirement.

12. Traceability

Traceability is the factor when a requirement is traceable to its test cases or not.

Reasoning: If there is pre-prepared test cases available then it is very beneficial for the developer organization and if the test case are not available then test cases must be designed for testing the requirements. If there are set of test cases corresponding to the requirements then assign zero priority value to this factor.

For each requirement, based on these 12 factors a Requirement Prioritization factor value (RPFV) is calculated by using Formula 4.1.

$$RPFV = \sum_{j=1}^{n} (pfvalue_{ij} * pfweight_j) \ldots\ldots\ldots (4.1)$$

Here i represent number of requirements and j represents number of factors.

In Formula 4.1 the RPFV represents the prioritization factor value for a requirement which is the summation of the product of priority value of a factor and the project factor weight. pfvlaue represents the value for factor for ith requirements and pfweight represents the factor weight for the jth factor for a particular project .

The value of the RPFV depends on the value of the pfvalue and the pfweight. The value of the RPFV will vary with a change in the factor weights and the factor value. The factor weight is assigned by the developer for the each factor. Total factor weight assigned by the developer to the all factors should not more than one. In this approach the developer can analyze the complexity of a requirement based on the factor weight assigned to that requirement.

4.3 PRIORITIZATION OF THE MODULE

In the process of prioritization of module mapping between the chosen prioritized requirement and its corresponding modules are performed. If there is more than one module the modules are prioritized. The criteria for module prioritization is based on the cyclomatic complexity and non dc path. Higher the cyclomatic complexity and non dc path of the module, higher is the priority of that module. The test cases of the higher priority module are prioritized first and executed. For each module a module prioritization value (MPV) is calculated by adding the cyclomatic complexity and the number of non-dc paths. A module having higher MPV is prioritized first.

Table 4.2 Module Prioritization

Factors	**M1**	**M2**	**M3**	**M4**
Cyclomatic complexity	8	4	4	5
Non Dc path	7	5	6	3
MPV	15	9	10	8

Table 4.2 shows the prioritization of four sample modules on the basis of MPV for each module. The order of prioritization of modules on the basis of MPV is M1, M3, M2 and M4.

4.4 TEST CASE PRIORITIZATION PROCESS

The test case prioritization process is used to prioritize and schedule the test cases corresponding to prioritized modules. In this test case prioritization process, there are some practical weight factors. On the basis of these practical weight factors process of the test case prioritization is performed. These factors are test Impact, test case complexity, requirements coverage and the dependency of the test cases as discussed below.

- **Test Case Complexity:** Complexity of test case shows that how difficult is a test case to execute. It shows how much efforts are required to execute the test case. After analyzing the complexity of test case the value of this factor is assigned between the value 1 and 10.

- **Requirement Coverage:** This factor shows that how many requirements are covered by executing the test case. This factor is scaled between the values from 1 to 10. The higher value shows the maximum requirements being covered by the test case. Higher the number of requirements coverage higher the priority of the test case to be executed first.

- **Dependency:** This factor shows the dependency of test case on some pre-requisites. It shows how many pre-requisites are required for each test case before the execution of the test case. The value of dependency factor is assigned between the values from 1 to 10.

- **Test Impact:** Test impact is the most critical factor in test case prioritization. It shows the impact of test case on a system if it is not executed. So this factor assesses the importance of the test cases. Here a value between the 1 and 10 is assigned.

After assigning the prioritize factor value to each factor as discussed above TCWP (Test case weight prioritization) is computed using Formula 4.2.

$$TCWP = \sum_{j=1}^{n} (fvalue_{ij} * fweight_{j}).....(4.2)$$

where TCWP is weight Prioritization for each test case calculated from the four factors, fvalue is value of factors assigned to each test case, fweight is a weight assigned to each factors.

The test cases are ordered on the basis of value of TCWP. A test case having maximum value is given highest priority and executed first.

Consider a set of four sample test cases TC1, TC2, TC3, and TC4 which are to be prioritized.

For these test cases TCWP is calculated by Formula 4.2 and test cases are prioritized on the basis of the value of TCWP (See Table 4.3).

Table 4.3 Test Case Prioritization

S. No.	Factors	TC1	TC2	TC3	TC4	Weight
1	Test Impact	4	8	7	9	0.4
2	Test case Complexity	8	7	5	9	0.3
3	Requirement coverage	6	2	4	4	0.2
4	Dependency	7	6	6	8	0.1
	TCWP	5.90	6.30	5.70	7.90	1.0

Now the order of the test case for the execution is TC4, TC2, TC1, and TC3. If the TCWP of the two test cases are same then a random order for execution of test cases is followed.

4.5 ANALYSIS OF PROPOSED HSTCP APPROACH

To analyze the effectiveness of proposed HSTCP approach, it was applied to the income tax calculator software which is used to calculate the tax on the income [80]. The software consists of 1160 lines of code and has nine modules named; Income details non salaried, income details salaried, savings, tax deductions, male Tax, female tax, senior tax and generates tax. All types of bugs like critical, major and medium and minor bugs were introduced intentionally so that testing can be performed on the software using proposed approach. Income tax software is based on following requirements.

- Accept Personal detail (APD)
- Accept income detail (AID)

- Accept tax deduction (ATD)
- Accept Savings and Donation details (ASD)
- Generate tax detail (GTD)

Now considering the twelve factors for requirements prioritization discussed in Section 3, the corresponding weight values for each requirement was calculated as shown in Table 4.4.

Table 4.4 Requirements Prioritization

Requirements → **Factors ↓**	**APD**	**AID**	**ATD**	**ASD**	**GTD**	**Weight factor**
Customer assigned priority	8	10	9	9	10	0.02
Developer assigned priority	8	9	9	8	10	.08
Requirements volatility	3	0	3	2	8	0.1
Fault Proneness	0	0	0	0	0	0.15
Expected faults	2	3	4	2	3	.10
Implementation Complexity	3	4	5	3	6	.10
Execution frequency	5	10	9	6	10	.05
Traceability	0	0	0	0	0	.05
Show stopper requirements	0	9	8	0	10	.2
Penalty	1	4	6	3	8	.05
Time	3	6	7	4	6	.05
Cost	4	7	8	6	7	.05
RPFV	2.25	4.77	5.20	2.47	6.25	1.0

Based on computation of RPFV the requirements prioritized list of the requirements is GTD, ATD, AID, ASD and APD Now the requirements were mapped to their corresponding modules. The cyclomatic complexity, number of non dc paths and the number of test cases of the modules are shown in Table 4.5.

Table 4.5: Module prioritization for income tax calculator case study

Requirements	**Module**	**C complexity**	**Non dc path**	**No. of test cases**	**MPV**
APD	Main module			8	
AID	NON salary Salary	8 12	7 10	4 6	15 22
ATD	Deduction	16	17	10	33
ASD	Saving	8	5	4	13
GTD	Male Tax Female Tax Senior Tax **Tax module**	4 4 4 6	0 0 0 0	4 4 4 6	4 4 4 6

In the Table 4.5, cyclomatic complexity, non DC paths and the number of test cases for testing of each module are shown. Here GTD requirement has the highest priority. There are four modules corresponding to this requirement. On the basis of the values of cyclomatic complexity and non dc paths, the MPV value for Tax module is more as compared to other three modules. So the test cases of the tax module have to be prioritized. Table 4.6 shows the values for different factors for six test cases and the weight assigned.

Table 4.6: Test case prioritization for test cases of tax module

S. No.	Factors	TC1	TC2	TC3	TC4	TC5	TC6	Weight
1	Test Impact	4	7	7	9	8	7	0.4
2	Test case Complexity	8	7	8	9	8	9	0.3
3	Requirement coverage	0	0	0	0	0	0	0.2
4	Dependency	2	2	2	2	2	2	0.1
	TCWP	4.2	5.1	5.4	6.5	4.8	5.7	1.0

Now by using Formula 4.2 the value of TCWP is calculated for these six test cases of tax module. The prioritized order of the test cases is TC4, TC6, TC3, TC2, TC5, and TC1.

4.5.1 Results obtained for HSTCP approach

The Tables (Table 4.7 to Table 4.12) show the number of the faults detected by the test cases of all prioritized requirements.

Table 4.7 Fault detection in Generate Tax details (GTD) requirement

Test ID	Critical Fault	Major fault	Medium fault	Minor fault
1	1	1		1
2		1		1
3		1		1
4		3		1
5		2		
6		2		1

Table 4.8 Fault detection in Income tax deduction (ATD) requirement

Test ID	Critical	Major	Medium	Minor
1		1		1
2				1
3				0
4			1	
5			1	
6			1	
7			1	1
8		1	1	
9	0	0	0	0
10		2	4	3

Table 4.9 Fault detection in Accept Savings and Donation details (ASD)

Test ID	Critical	Major	medium	Minor
1		1		
2		1		
3			1	
4			1	1

Table 4.10 Fault detection in Income detail module of Accept income detail (AID)

Test id	Critical	Major	Medium	Minor
1		1		3
2				1
3				1
4		1		1

Table 4.11: Fault detection in Income detail salaried module of Accept income detail (AID) requirement

Test id	Critical	Major	Medium	Minor
1		1		
2				
3				
4				
5		1		
6		2		

Table 4.12: Fault detection Accept Personal detail (APD)

Test id	Critical	Major	Medium	Minor
1		1	1	
2		1		
3				
4				
5				1
6				
7				1
8				

Table 4.13 shows the total faults severity of each requirement. Faults severity is calculated using the Formula 4.3.

Fault severity = 4* no. of critical bugs+ 3* no of major bugs+2* no of medium bugs+1* no of minor bugs ---------- (4.3)

Table 4.13: Number and type of faults detected by all requirements

Requirement	Critical	Major	Medium	minor	Total Faults severity
GTD	1	10	0	5	39
ATD	0	4	9	6	36
AID	0	6	0	6	24
ASD	0	2	2	1	11
APD	0	2	1	2	11

The fault severity corresponding to various requirements is shown below in Figure 4.4.

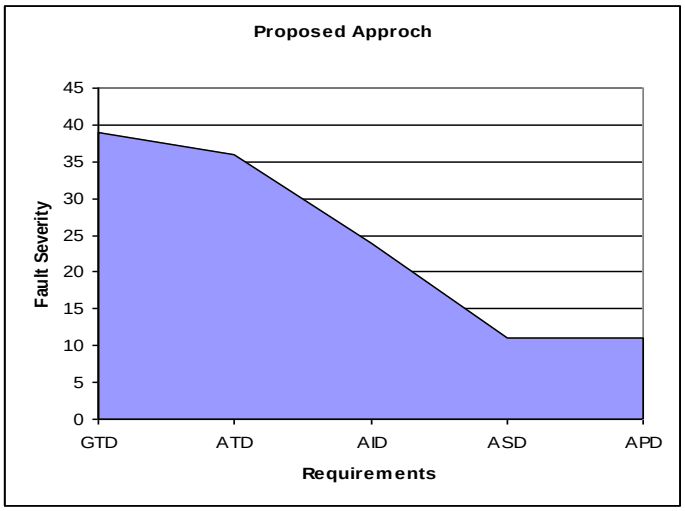


Figure 4.4: Graph for Proposed HSTCP approach based on requirements

A comparison of the proposed HSTCP approach has also performed with random as well as PORT [36] approach as shown in Figure 4.5 and Figure 4.6.

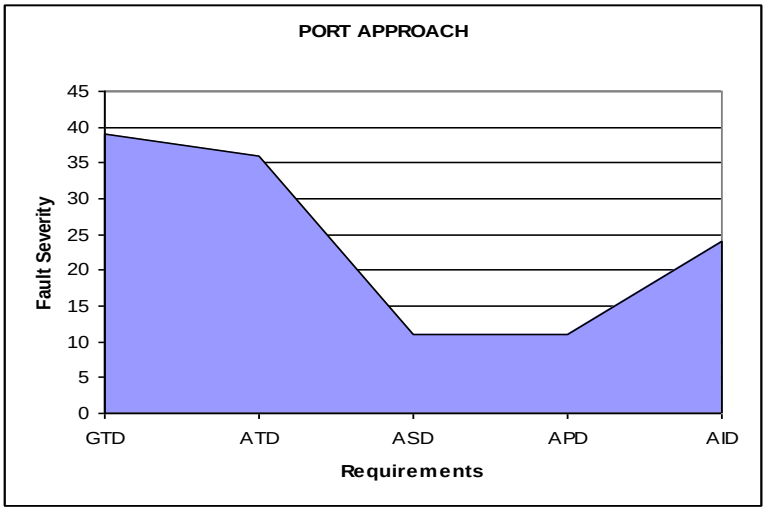


Figure 4.5: Graph obtained using PORT approach

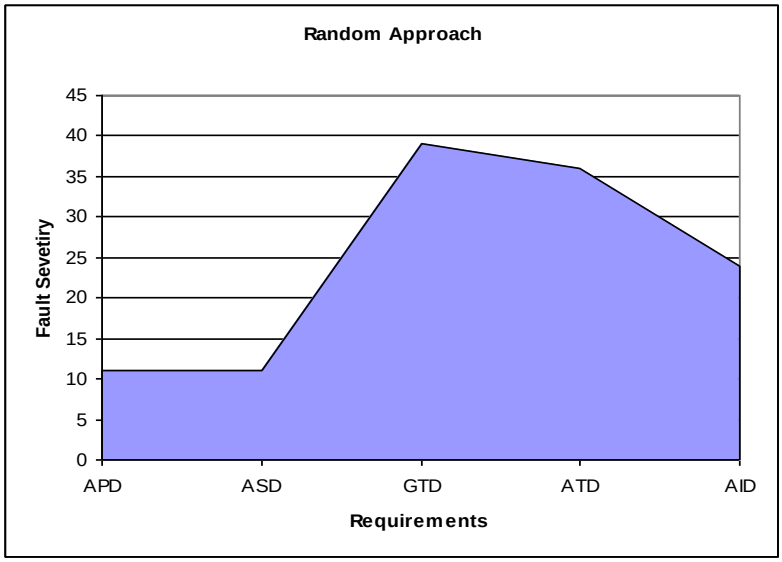


Figure 4.6: Graph for non – Prioritized test suite

By using the Formula 3.2 for calculating APFD given in chapter 3, APFD values were calculated for Proposed, PORT and Random approaches which are given below.

APFD For proposed HSTCP approach:

$$APFD = 1 - \frac{(1+10+18+8+10+31+21+28)}{57*5} + \frac{1}{2*57}$$

APFD= 1-107/285+1/114

APFD= .53

APFD For PORT approach:

$$APFD = 1 - \frac{(1+10+14+38+18+10+17+41)}{57*5} + \frac{1}{2*57}$$

APFD= 1-147/285+1/114

APFD= .47

APFD For random approach:

$$APFD = 1 - \frac{(1+11+34+51+42+20+24+49)}{57*5} + \frac{1}{2*57}$$

APFD= 1-207/285+1/114

APFD= .27

The comparison is drawn between proposed approach, non – prioritized and PORT approach. It indicates that value obtained for proposed approach is more than the previous methods, thereby showing the efficacy of prioritized method. In this way the proposed HSTCP technique based on requirements approach proves to be more effective as compared to other two approaches as shown in Figure 4.7.

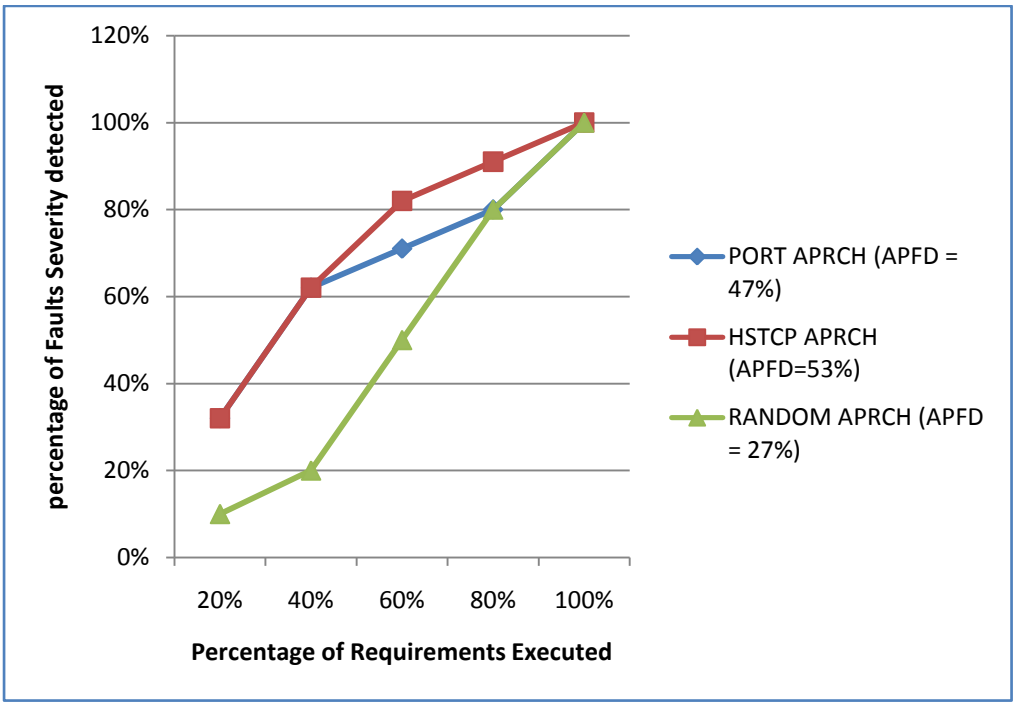


Figure 4.7: Comparison between Random, PORT, and Proposed HSTCP approach

4.6 IMPLEMENTAION

To implement the proposed approach a tool named as HSTCP has been developed in JAVA language [49]. This tool will help in prioritizing the requirements and further the modules and test cases in hierarchical manner. Using this tool the tester is able to execute the test cases in highly prioritized order, so that test cases may detect critical bugs earlier. Some of the snapshots of the tool developed are shown in Figures 4.8 to Figure 4.13.

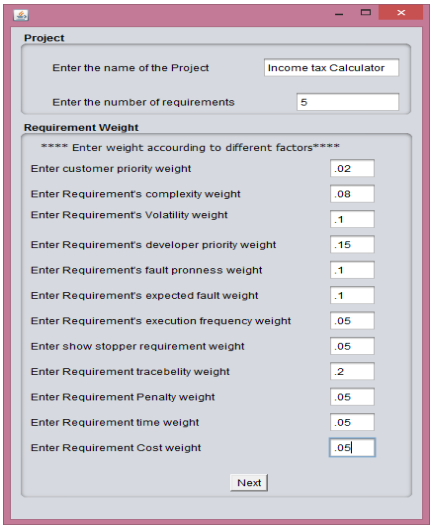


Figure 4.8: Snapshot 1of HSTCP tool

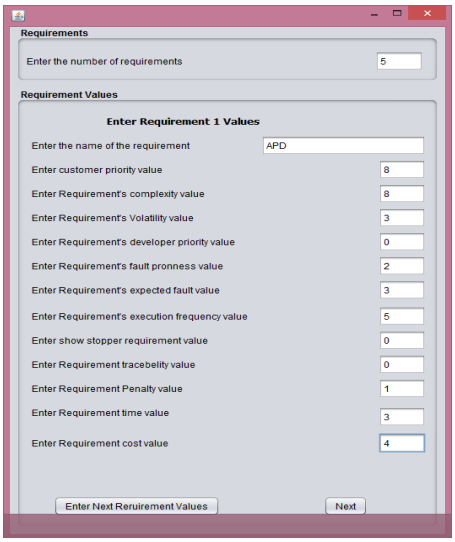


Figure 4.9: Snapshot 2of HSTCP tool

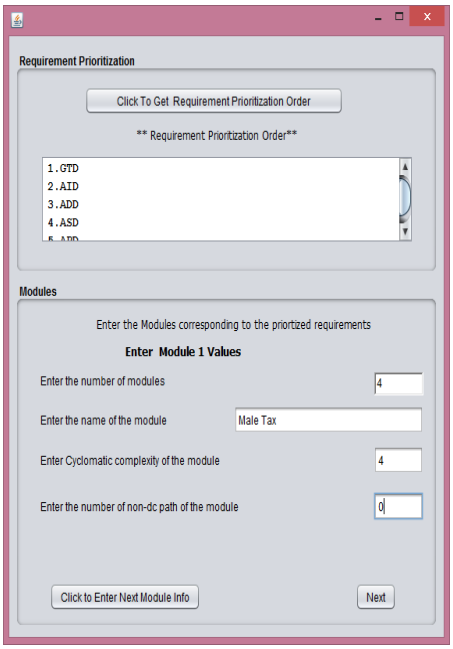


Figure 4.10: Snapshot 3 of HSTCP tool

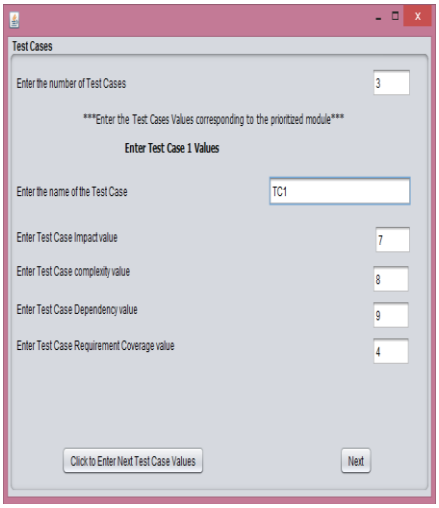


Figure 4.11: Snapshot 4 of HSTCP tool

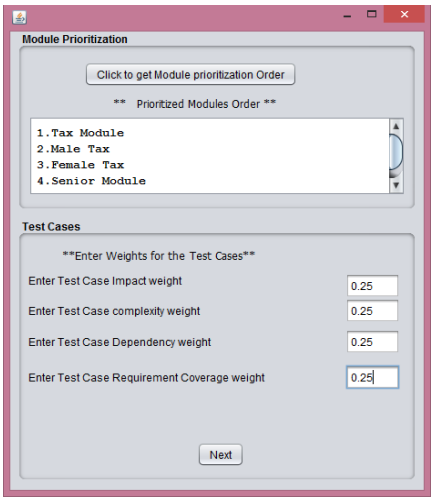


Figure 4.12: Snapshot 5of HSTCP tool

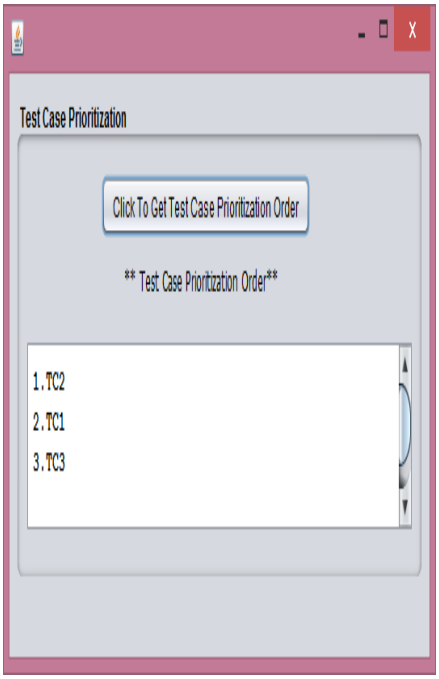


Figure 4.13: Snapshot 6 of HSTCP tool

4.7 CONCLUSION

A hierarchical system test case prioritization technique has been presented in this chapter. The proposed technique maps the requirement to its corresponding design modules and further mapped to the corresponding test cases. This approach can be used to improve the rate of severe fault detection for system testing. An experimental study of income tax calculator software is presented for comparing the effectiveness of proposed approach with previous approach (PORT) and with random prioritization approach [36]. The experimental results show that proposed new prioritization technique is promising in terms of ordering requirements so that faults are detected earlier in the testing phase. A tool has also been developed for demonstrating the proposed approach.

Chapter V

REGRESSION TEST CASE PRIORITIZATION: PROPOSED WORK

5.1 INTRODUCTION

This chapter is concerned with prioritizing the test cases for regression testing. Three techniques for prioritizing the test cases during regression testing have been proposed in this chapter. The first technique is module-coupling-effect based test case prioritization technique which basically finds out the badly affected module due to change in the modules of software under consideration. . The second approach prioritizes the test cases during regression testing using data flow testing concepts. The third technique is control structure weighted test case prioritization which is the extension for the second approach.

5.2 MODULE-COUPLING -EFFECT BASED TEST CASE PRIORITIZATION (MCETCP) TECHNIQUE

In this section a new technique for prioritizing the test cases while performing regression testing has been proposed [46]. This technique is based on the module dependence and coupling between the modules. Module coupling effect can be one of the criteria for prioritizing test cases in order to carry out the regression testing.

Whenever there is a change in a module, certainly there will be some effect on other modules which are coupled together. Based on the coupling information between the modules the highly affected module can be found out. Moreover, the effect is worse if there is high coupling between the modules causing the high probability of errors. This may be called as module-coupling effect. In this way if regression test case prioritization is done based on this module coupling effect, there will be high percentage of detecting critical errors that have been propagated to other modules due to any change in a module.

For example in Figure 5.1 the modules 17 and 18 are being called by multiple modules. If there is any change in module 17 and module 18, modules 9, 11 and 12, 13 will be affected respectively. If there is no prioritization, then as a part of regression testing process, all the test cases of all the affected modules will be executed thereby increasing the testing time and effort. Instead, if the coupling type between modules is known, then a prioritization scheme can be developed based on this coupling information. The modules having worst type of coupling will be prioritized over other modules and their test cases.

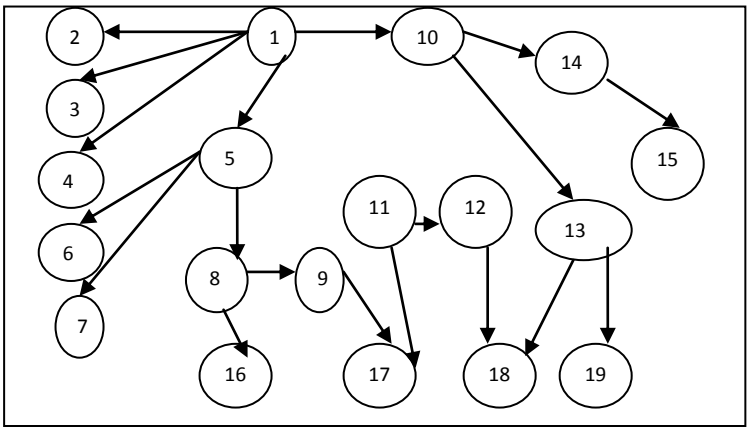


Figure 5.1: Call Graph Example

After finding out the affected module due to a change in a module, there is need to execute the test cases of this affected module. However, there may be a large number of test cases in this module. To prioritize the test cases of this affected module, test case prioritization technique discussed in Chapter 3, can be applied.

This section proposes a novel, effective and efficient method to accomplish test case prioritization method discussed above. The proposed method has been tested, verified and validated and the results have been presented in this section.

5.2.1 Module Dependency Matrix

Module dependence can be identified by coupling and cohesion. A quantitative measure of the dependence of modules will be useful to find out the stability of the design. The work is based on the premise that different values can be assigned for various types of module coupling and cohesion as shown in Tables 5.1 and 5.2.

Table 5.1: Coupling Types and their values

Coupling Types	**Value**
Content	0.95
Common	0.70
External	0.60
Control	0.50
Stamp	0.35
Data	0.20

Table 5.2: Cohesion types and their values

Cohesion Types	**Value**
Coincidental	0.95
Logical	0.40
Temporal	0.60
Procedural	0.40
Communicational	0.25
Sequential	0.20
Functional	0.20

A matrix can be obtained by using these two tables, which gives the dependence among all the modules in a program. This, dependence matrix describes the probability of having to change module i, given that module j has been changed. Module Dependence Matrix is derived using the following three steps.

5.2.2 Procedure of Making Module Dependence Matrix

STEP 1:

Determine the coupling among all of the modules in the program. Construct an m*m coupling matrix, where m is the number of modules in the program. Using Table 5.1 fill each element in the matrix C. Element Cij represents the coupling between module i and module j. The matrix is symmetric i.e.

$$Cij = Cji \quad \text{for all i \& j.}$$

Also elements on the diagonal are all 1(Cii =1 for all i)

STEP 2:

Determine strength of each module in the program. Using Table 5.2 record the corresponding numerical values of cohesion in module cohesion matrix.

STEP 3:

Construct the Module dependence matrix D by the Formula 5.1.

$Dij = 0.15 (Si + Sj) + 0.7 Cij$, where $Cij \neq 0$

$Dij = 0$ where $Cij = 0$ $Dii = 1$ for all i. ---------- (5.1)

Prioritization of module can be done by comparing non zero entries of D matrix (Module Dependence Matrix). For Example if module number i has been modified then find all the existing parent modules (j, k, l…) of that changed module (i) and after that compare first order dependence matrix entries for particular links viz (i-j, i-k, i-l & so on). Link having highest module dependence matrix value will get highest priority & link with low module dependence matrix value will get low priority.

5.2.3 Proposed MCETCP Approach

The functioning of the proposed technique consists of the following components (shown in Figure 5.2).

- **Call Graph Producer**

  With this component, a call graph can be produced for the given program. Using this component, the calling sequence among the modules can be known.

- **Coupling and Cohesion Identifier**

  Using the call graph producer component the type of dependency among the modules is identified, i.e. coupling and cohesion.

- **First Order Dependence Matrix Calculator**

  This component performs four major functions which are described below.

  1. Creation of coupling matrix C.
  2. Creation of cohesion matrix S.
  3. On the basis of C and S create dependency matrix D.
  4. Assigning values (non zero and non one entries) to the edges of the call graph.

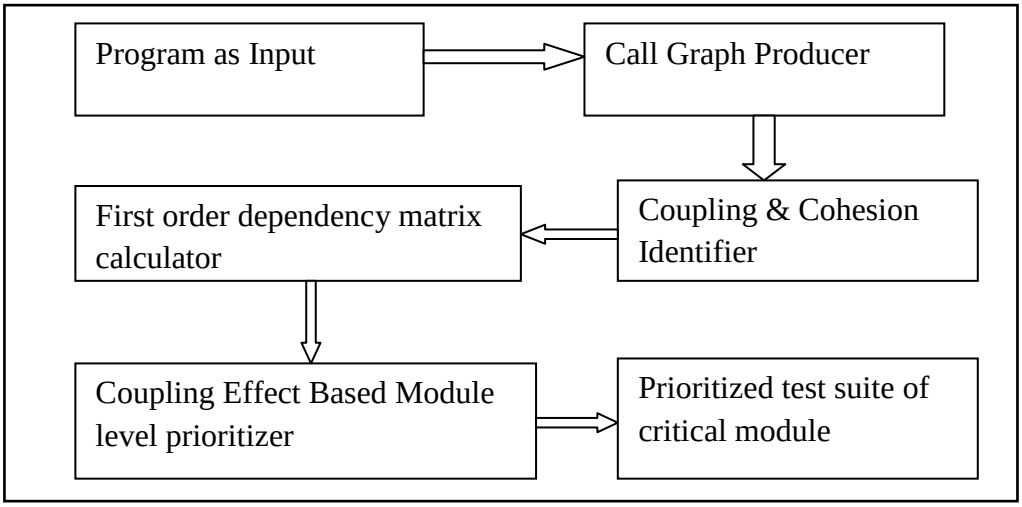


Figure 5.2: Components Showing the Process of MCETCP

- **Coupling Effect based module level prioritizer**

The first order dependence matrix provides the coupling values among the modules. Using these values, this component identifies the worst affected module due to the changed module. Thereby we get a prioritized module among several affected modules.

5.2.4 Proposed Algorithm for finding highly coupled module

First of all a coupling matrix is created by finding coupling values among different modules. After creating a coupling matrix, a cohesion matrix is created by identifying the type of cohesion in the individual module. Now, by using these two matrices a module dependence matrix is created. After this a module which is changing is identified. Finally the parent module of the changed module is identified with the help of module dependency matrix. The module with the highest value is prioritized over other modules. The proposed algorithm is shown in Figure 5.3.

PRIORITIZATION (P, n)
Begin
(Where P is the complete program and n is number of modules)

1. *Identify type of coupling between modules and create coupling matrix **C** using coupling values.*
2. *Identify type of cohesion in the individual module and create cohesion matrix **S** using cohesion values.*
3. *Using C and S Matrix construct first order dependence matrix **D.***
4. *Identify which module number is changing(c).*
5. *Identify parent (p) of changed module using first order dependence matrix (D) values. (Highest value module will get priority over other module)*

End

Figure 5.3: Algorithm for finding highly coupled module

5.2.5 Evaluation & Results of MCETCP approach

To evaluate the proposed MCETCP approach, a case study of software consisting of 10 modules has been taken whose source code has been given in Appendix D. The coupling and cohesion information of these modules are shown in Table 5.3 and Table 5.4. The Call graph for the case study software is shown in Figure 5.4.

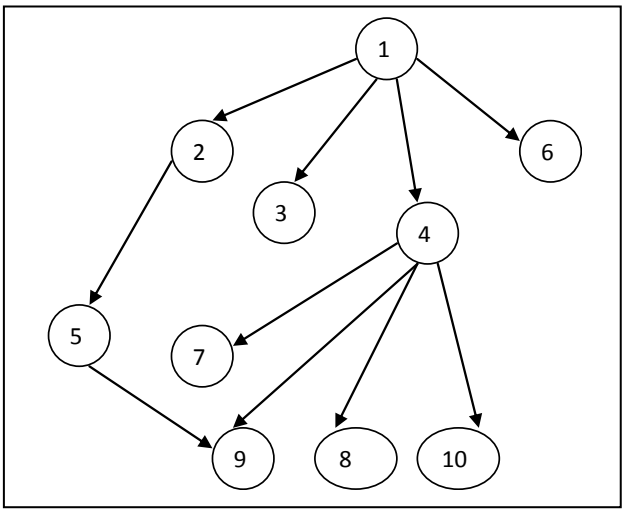


Figure 5.4: Call Graph of Case Study Software

Table 5.3 Coupling Information for case study software

Type of Coupling	**No. of modules in relation**	**Examples**
Data Coupling	3	1-2,1-4,1-6
Stamp Coupling	1	1-3
Control Coupling	4	4-7,4-8,4-9,4-10
Common Coupling	2	2-5,5-9
Message Coupling	1	1-5

Table 5.4: Cohesion Information for case study software

Module Number	**Cohesion Type**
1	Coincidental
2	Functional
3	Communicational
4	Logical
5	Procedural
6	Functional
7	Functional
8	Functional
9	Functional
10	Functional

By using the coupling values (See Table 5.3) among different modules a module coupling matrix is being prepared as shown in Table 5.5.

Table 5.5 Module Coupling Matrix(C)

1.0	0.2	0.35	0.2	0.95	0.2	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.70	0.2	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0	0.0	0.50	0.50	0.50	0.50
0.0	0.2	0.0	0.0	1.0	0.0	0.0	0.0	0.70	0.0
0.0	0.2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.2	0.0	0.0	0.0	1.0	0.6	0.0	0.0
0.0	0.0	0.0	0.2	0.0	0.0	0.6	1.0	0.2	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	1.0

By using the Cohesion values (See Table 5.4) among different modules a Cohesion Matrix(S) is being designed as shown in Table 5.6.

Table 5.6 Module Cohesion Matrix (S)

0.95	0.2	0.25	0.4	0.4	0.2	0.2	0.2	0.2	0.2

By using Formula 5.1, a Module dependence Matrix is being designed as shown in Table 5.7, wherein various module dependence values have also been shown.

Table 5.7 Module Dependence Matrix (D)

1.0	0.31	0.42	0.34	0.0	0.31	0.0	0.0	0.0	0.0
0.31	1.0	0.0	0.0	0.58	0.0	0.0	0.0	0.0	0.0
0.42	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.34	0.0	0.0	1.0	0.0	0.0	0.44	0.44	**0.44**	0.44
0.0	0.58	0.0	0.0	1.0	0.0	0.0	0.0	**0.58**	0.0
0.31	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	**0.44**	**0.58**	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	1.0

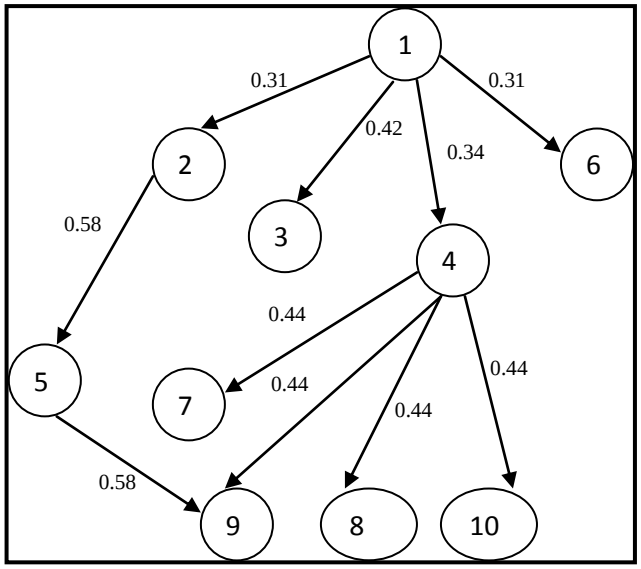


Figure 5.5: Case Study Software Call Graph with module dependence values

From the module dependence values obtained from call graph (See Figure 5.5) and from Module Dependence Matrix, we conclude that change in the Module 4 propagates to module 7, 8, 9 and module 10. Modules 7, 8, 9 and 10 are having same module dependence values (0.44), so the order of prioritization of test cases for these modules is same. Similarly the change in the module 1 propagates to Module 2, 3, 4 and 6 .The module dependence values for these modules shows that the module 3 is more affected module as compared to module 2, 4 and 6. So the test cases for the module 3 have to be prioritized first as compared to module 2, 4 and 6.

A tool is also implemented in C language which finds out the badly affected module due to change in a particular module. The inputs to this tool are the coupling and cohesion information among modules.

5.3 TEST CASE PRIORITIZATION USING DATA FLOW TESTING

Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. While identifying the test cases of data flow testing, there may be large number of test cases. It may not be possible for a tester to execute all the test cases identified in this huge test suite due to time and cost constraints. Therefore, there is need to prioritize the test cases so that the important test cases that identify the

critical bugs are executed first. For this purpose the concept of du-and dc-paths has been taken. There are du-paths which are variable usage paths and the paths wherein the variable is defined more than once called non-dc paths (non-definition clear paths) in data flow testing. The du-paths which are not dc-paths are problematic for a tester. It means that there may be more bugs in the du-paths which are not dc-paths.

Further, if a program is modified the probability of errors may increase because after modification new du paths may get introduced and some of these du paths may also be not definition clear, that is these paths may also be more prone for errors. Therefore, a test case prioritization technique is required that will take care of these problems. In this work, a new technique for test case prioritization is proposed that prioritized the test cases while performing regression testing [47]. For this purpose a list of newly introduced non-dc paths is prepared and the set of test cases corresponding to these paths are put at highest priority in the test suite of modified program. The set of test cases is referred as Set-1. It may happen that because of modification in the program that some existing dc paths may became non-dc.

The set of test cases corresponding to these paths are taken at the next priority and this set of test cases may be referred as 'set-2'. Further, there may be some non-dc paths of the original program, which are still non-dc after modification of the program. The set of test cases for such paths may be referred as 'set3'.Finally, there are some test cases which do not cover non-dc paths, i.e. these test cases cover dc-paths. This may be referred as 'set4'.Using these four concepts, an algorithm has been designed for prioritizing the test suite as shown in Figure 5.6.

Begin
Step1. Note down whether changes have introduced new definitions of Variables as well as uses.
Step2. Find out new du paths if introduced.
Step3. Design test cases for these new du paths if required.
Step4. List the non-dc paths out of these du paths.
Step5. Put the test cases corresponding to these listed non-dc paths on highest priority (Set-1) for testing in test suite.
Step6. List the du-dc paths of original program, which have now become non-dc after modification of program.
Step 7. Put the test cases corresponding to these listed non-dc paths (Set-2) on the next priority.
Step 8. List the non-dc paths of the original program, which are still non-dc after modification of the program.
Step 9. Put the test cases corresponding to these listed non-dc paths (Set-3) on the next priority.
Step 10. Put all the remaining test cases i.e. the test cases which do not cover any non-dc path (Set-4) at next but equal priority.
End

Figure 5.6: Algorithm for prioritizing the test cases using data flow testing

However, the test cases within a set are not prioritized. So to prioritize the test case within a set, an algorithm is designed which is shown in Figure 5.7.

Begin

Step1

If a test case covers more no. of paths in set-1
Then place it at the highest priority.

Step2

If two test cases (in Step1) covers equal no. of paths in set-1,
Then the test case which covers more no. of paths in set-2 in the modified program is placed at the next priority.

Step 3

If two test cases (in Step2) covers equal no. of paths in set-2 in the modified program
Then the test case which covers maximum no. of lines of code in the modified program is placed at next priority.

End

Figure 5.7: Algorithm for prioritizing the test cases within a set

5.3.1 Analysis of the proposed data flow testing approach

To analyze the efficacy of the proposed work it has been applied on three different modules of case study of Income tax calculator software [80].

A) Case Study 1: Income details for Income Tax Calculator

Source Code of income tax details module of Income tax calculator software has given below.

Source Code of Case Study 1:

```
float income_details_non_sal()
        {
1'      char source[20]="abc";
2'      float amount,total =0;
3'      int flag1=1,flag2=1,i;
4'      char income_ch='y';

1       while((income_ch=='y')||(income_ch=='Y'))
2       {
3       while(flag1==1)
4               {
5               printf("\nEnter SOURCE\t:");
6               gets(source);
```

```
7               for(i=0;i<strlen(source);i++)
8                       {
9               if(((toascii(source[i])  >= 65) && (toascii(source[i]) <= 122)) ||
                (toascii(source[i]) == 32))
10                              {
11                              flag1=0;
12                              }
13                      else
14                              {
15                                      printf("\nSource can contain only charcter Error
                                        at position number %d",i);
16                              flag1=1;
17                              break;
18                              }
19                      }//end for
20              if((strlen(source)<3)||(strlen(source)>20))
21                      {
22                      printf("\nSource can contain a max of 20 characters");
23                      flag1=1;
24                      }
25              }
26      while(flag2==1)
27              {
28              printf("\nEnter Amount\t:");
29              scanf("%f",&amount);
30              if(amount>0)
31                              {
32                              flag2=0;
33                              }
34                      else
35                              {
36                              printf("\nAmount cannot be less than or equal to 0");
37                              flag2=1;
38                              }
```

```
39              }
40              printf("\n\nPress any key to proceed");
41              getch();
42              clrscr();
43              patt("INCOME Details");
44              printf("\nSOURCE\t:%s",source);
45              printf("\nAMOUNT\t:%f",amount);
46              total=total+amount;
47              printf("\nDo you want to enter more(y/n)\t:");
48              income_ch=getche();
49              flag1=1;
50              flag2=1;
51              }
52              printf("\nTotal\t\t:%f",total);
53              return(total);
54              }
```

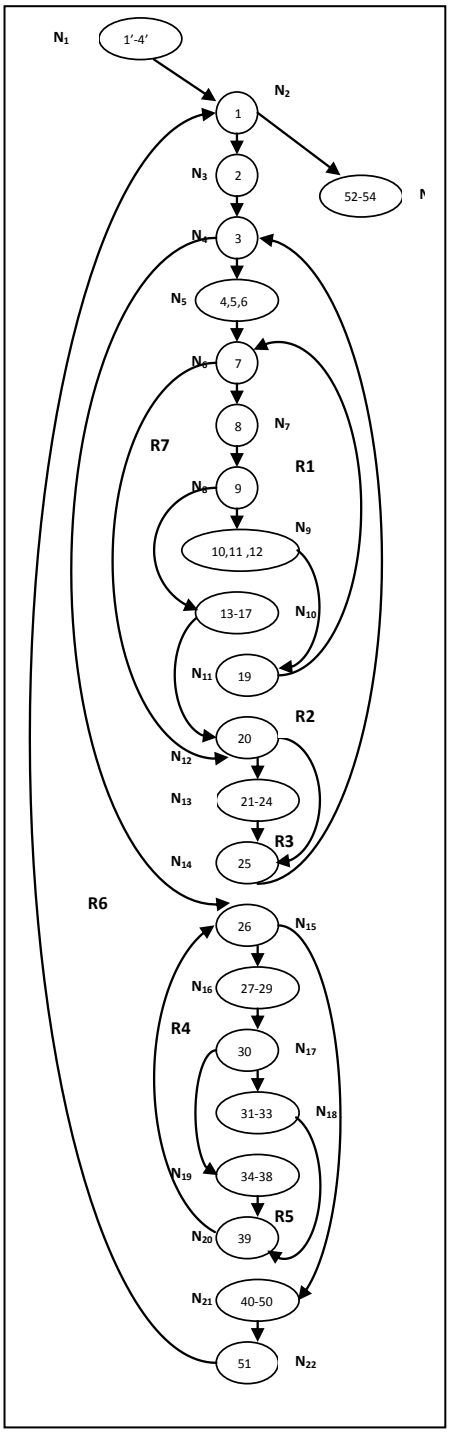


Figure 5.8: CFG for income details module

The control flow graph of income detail is depicted in Figure 5.8. Test cases for the same are shown in Table 5.8.Since the cyclomatic complexity of the graph is 8, so there will be 8 independent paths in the graph as shown below:

1) $N_1N_2N_{23}$
2) $N_1N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3) $N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4) $N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5) $N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6) $N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7) $N_1N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8) $N_1N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Table 5.8: Test Case Design for income detail from the Independent Paths

Test Case ID	**Inputs**		**Expected Output**	**Independent path covered by Test Case**	**Total lines of code covered by test case**
	Source	**Amount**			
1	Agriculture	400000	Source Agriculture Amount 400000 Do you want to enter more(y/n) Y	1), 2), 3), 5), 8)	42 lines
	Others	100000	Source Others Amount 100000 Do you want to enter more(y/n) N		
			Total 500000		
2	1234		Source can contain only character.	1), 2), 3), 4)	19 lines
3	Agriculture and others		Source can contain a max of 20 characters.	1), 2), 3), 6)	24 lines
4	Agriculture	0	Amount cannot be less than or equal to 0	1), 2), 3), 5), 7)	31 lines

Now the following modifications have been made to the program. ExamDiff tool is used to find out the changes made in modified version as compared to old version of program. The changes detected are as follows:

1. Line 7 is a new addition i.e (len=strlen(source)).
2. A new variable len is introduced in modified version.

3. ‘strlen(source)’ string is replaced by variable ‘len’ in lines 8 and 21.

The proposed data flow testing approach is applied on modified income detail module case study in the following steps:

1. Changes have introduced new definition of variable ‘len’ and also its use (See Table 5.9).

Table 5.9: Definition and use of variable ‘len’ for income detail module case study

Variable	**Defined at**	**Used at**
Len	7	8,21

2. New du paths introduced due to modification and the test cases which covers these paths are shown in Table 5.10.

Table 5.10: Du paths and test cases

Variable	**du Path(beg-end)**	**dc?**	**Test case which covers this path**
len	7-8	yes	1,2,3,4
	7-21	yes	1,3,4

3. There is no need to design any new test case because all the new du paths get covered by existing test cases.

4. No new non-dc path gets introduced in the program after modification.

5. No dc path of original program changed its status to non-dc after modifications in the program.

6. The test cases corresponding to non-dc paths of original program are:
   TC1, TC2, TC3, TC4

7. There is no remaining test case in this program.

So the prioritized test suite for this case study after applying the proposed approach is

{TC1, TC3, TC2, TC4}

To evaluate the effectiveness of the approach the test cases are executed in random order and in prioritized order. The APFD values for the same have been shown in the Figure 5.9.

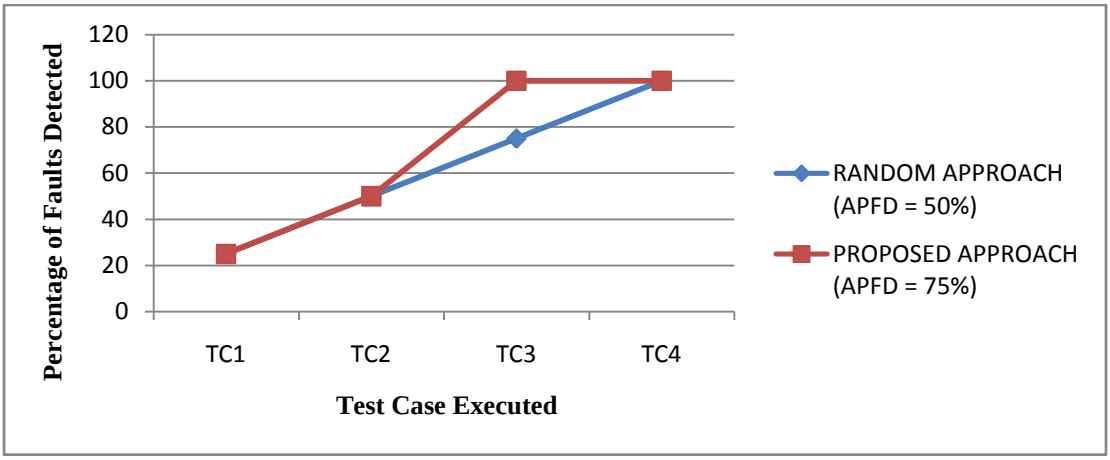


Figure 5.9: APFD values for random and proposed data flow TCP for income detail module

B) Case Study 2: Saving Module for Income Tax Calculator

Source Code of saving module of Income tax calculator software is given below.

Source Code for Case study 2:

```
float savings()
        {
1       char saving_type[20];
2       float amount,total=0;
3       int flag1=1,flag2=1,i;
4       char sav_ch='y';
5       while((sav_ch=='y')||(sav_ch=='Y'))
6       {
7       while(flag1==1)
8               {
9               printf("\nEnter Saving type\t:");
10              gets(saving_type);
11              for(i=0;i<strlen(saving_type);i++)
12                      {
```

```
13      if(((toascii(saving_type[i])>= 65) && (toascii(saving_type[i]) <= 122)) ||
            (toascii(saving_type[i]) == 32))
14                              {
15                              flag1=0;
16                              }
17                      else
18                              {
19      printf("\nSaving type can contain only character Error at position number
%d",i);
20                              flag1=1;
21                              break;
22                              }
23                      }
24              if((strlen(saving_type)<3)||(strlen(saving_type)>20))
25                      {
26                      printf("\nPlease enter between 3 to 20 characters ");
27                      flag1=1;
28                      }
29              }
30      while(flag2==1)
31              {
32              printf("\nEnter Amount\t:");
33              scanf("%f",&amount);
34              if(amount>0)
35                              {
36                              flag2=0;
37                              }
38                      else
39                              {
40                              printf("\nAmount cannot be less than or equal to 0");
41                              flag2=1;
42                              }
43              }
44              printf("\n\nPress any key to proceed");
```

```
45              getch();
46              clrscr();
47              patt("SAVING Details");
48              printf("\nSAVING TYPE\t:%s",saving_type);
49              printf("\nAMOUNT\t\t\t:%f",amount);
50              total=total+amount;
51              printf("\nDo you want to enter more(y for yes)\t:");
52              sav_ch=getche();
53              flag1=1;
54              flag2=1;
55              }
56              printf("\nTotal\t\t:%f",total);
57              return(total);
58              }
```

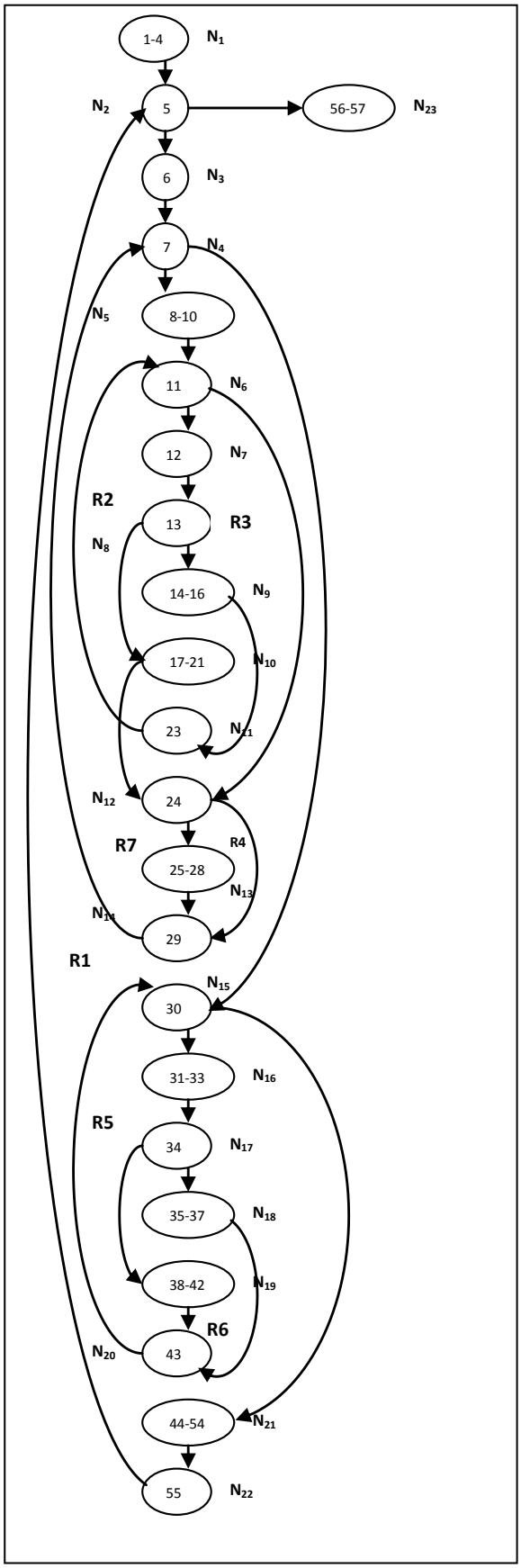


Figure 5.10: CFG for saving module of income tax calculator

The CFG for saving module case study is shown in Figure 5.10. Cyclomatic complexity of the graph is 8, so there are 8 independent paths in the graph as shown below and the test cases corresponding to these paths are shown in Table 5.11.

1) $N_1\ N_2\ N_{23}$
2) $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3) $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4) $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5) $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6) $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7) $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8) $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Table 5.11: Test Cases for saving module case study from the Independent Paths

Test Case ID	**Inputs**			**Expected Output**	**Independent path covered by Test Case**	**Total lines of code covered by test case**
	Saving Type	**Amount**	**Enter more?**			
1	NSC	5000		Saving type NSC Amount 5000	1), 2), 3), 5), 8)	43 lines
			Y			
	PPF	1200		Saving type PPF Amount 1200		
			N	Total 6200		
2	123			Saving type can contain only character	2), 3), 4),	18 lines
3	PF			Please enter between 3 to 20 characters	2), 3), 6)	23 lines
4	PPF	0		Amount cannot be less than or equal to 0.	2), 3), 7)	28 lines

The changes detected are as follows:

1. line 10a is a new addition (len = strlen(saving_type))

2. A new variable len is introduced in modified version

3. ‘strlen(savings_type)’ string is replaced by variable ‘len’ in lines 11 and 24.

The step wise execution of the proposed data flow approach for saving module case study is given below.

1. Changes have introduced new definition of variable ‘len’ and also its use shown in Table 5.12

Table 5.12: Definition and use of variable ‘len’ for saving module

Variable	**Defined at**	**Used at**
len	10a	11,24

2. New du paths introduced with the definition of variable ‘len’ are shown in Table 5.13.

Table 5.13: Du paths & Test case coverage of variable ‘len’

Variable	**du Path(beg-end)**	**dc?**	**Test case which covers this path**
len	10a-11	Yes	1,2,3,4
	10a-24	Yes	1,3,4

3. There is no need to design any new test case because all the new du paths get covered by existing test cases.

4. No new non-dc path gets introduced in the program after modification.

5. No dc path of original program changed its status to non-dc after modifications in the program.

6. The list of test cases corresponding to non-dc paths of original program are: TC1, TC2, TC3, TC4

7. There is no remaining test case in this program

So the prioritized test suite for this program after applying the proposed approach and technique applied for original program is

{TC1, TC3, TC4, TC2}

To evaluate the effectiveness of the approach the test cases are executed in random order and in prioritized order. The APFD values for the same have been shown in the Figure 5.11.

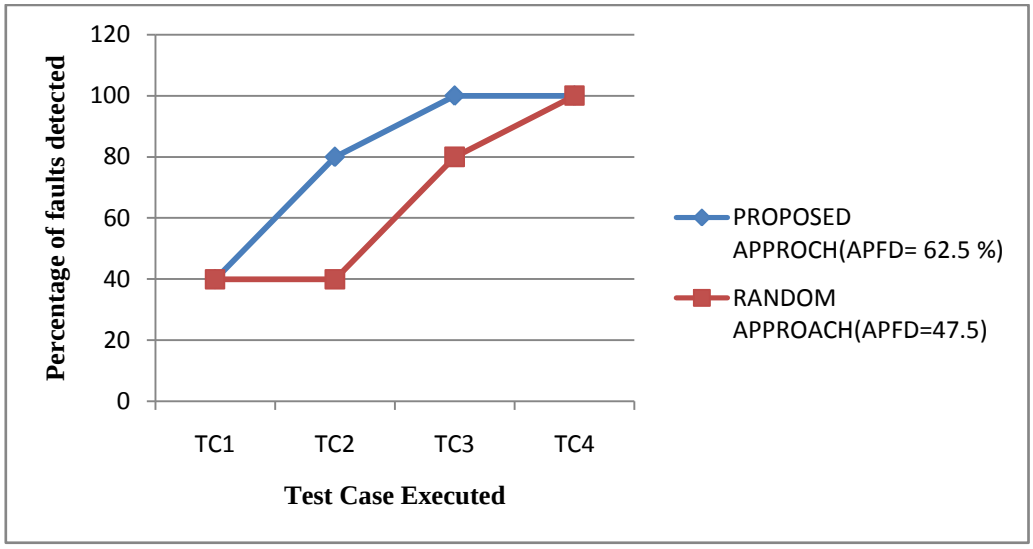


Figure 5.11: APFD values for random and proposed data flow TCP for saving module case study

C) Case Study 3: Income details module for Income Tax Calculator

Source Code of income details module of Income tax calculator software.

Source code of case study 3:

```
double income_details_sal()
{

1'      float t_d, d1, d2, sal1, sal2, sal3, t_sal, sal_all, sal_all_tot=0, ei, t_ei=0,
            net_t_sal=0, bal;
2'      char sal_ch='y',ei_ch='y';
3'      int f1=1,f2=1,f3=1,f4=1;
4'      double gross;

1       while(f2==1)
2               {
3               printf("\n1.\tGROSS SALARY\t:");
```

```
4               printf("\n\ta) Salary as per the provisions contained in the section
                17(1)\t:");
5               scanf("%f",&sal1);
6               printf("\n\tb) Value of the perquisites under section 17(2)\n(As per
                form number 12BA , wherever applicable )\t:");
7               scanf("%f",&sal2);
8               printf("\n\tc) Profits in lieu of salary under section 17(3)\n(As per form
                number 12BA , wherever applicable )\t:");
9               scanf("%f",&sal3);
10              if((sal1<0)||(sal2<0)||(sal3<0))
11                      {
12                      f2=1;
13                      }
14              else
15                      {
16                      f2=0;
17                      }
18              }
19              t_sal=sal1+sal2+sal3;
20              printf("\n\td)\tTotal\n\t\t\t\t:%f",t_sal);
21              sal_all_tot=0;
22              while((sal_ch=='y')||(sal_ch=='Y'))
23              {
24              while(f1==1)
25                      {
26                      printf("\n2.\tAllowance to the extent exempt under section
10\t:");
27                      scanf("%f",&sal_all);
28                      if(sal_all<0)
29                              {
30                              printf("\nEnter correct value");
31                              }
32                      else
33                              {
```

```
34                              f1=0;
35                              }
36                      }
37              sal_all_tot=sal_all_tot+sal_all;
38              printf("\nEnter more?(Y/N)\t:");
39              sal_ch=getche();
40              if((sal_ch=='y')||(sal_ch=='Y')||(sal_ch=='n')||(sal_ch=='N'))
41                              {
42                              f1=0;
43                              }
44              else
45                              {
46                              printf("\nPlease enter y or n");
47                              }
48              }
49              printf("\nTotal allowance\t\t:%f",sal_all_tot);
50              bal=t_sal-sal_all_tot;
51              printf("\nBalance\t:%f",bal);
52              while(f3==1)
53              {
54              printf("\n3.\tDeductions\t:");
55              printf("\n\tEntertainment allowance(EA)\t:");
56              scanf("%f",&d1);
57              printf("\tTax on employment (TE)\t:");
58              scanf("%f",&d2);
59              if((d1<0)||(d2<0))
60                      {
61                      f3=1;
62                      }
63              else
64                      {
65                      f3=0;
66                      }
67              }
```

```
68              t_d=d1+d2;
69              printf("\nTotal deductions\t:%f",t_d);
70              net_t_sal=bal-t_d;
71              printf("\n4.\tINCOME CHARGABLE UNDER THE HEAD
                SALARIES'\t:%f",net_t_sal);
72              while((ei_ch=='y')||(ei_ch=='Y'))
73              {
74              while(f4==1)
75              {
76              printf("\n5.\tAny other income reported by the Employee\t:");
77              printf("\n\t\tEnter Income\t:");
78              scanf("%f",&ei);
79              if(ei<0)
80                      {
81                      f4=1;
82                      }
83              else
84                      {
85                      f4=0;
86                      }
87              }
88              t_ei=t_ei+ei;
89              printf("\n\t\tEnter more?(Y/N)\t:");
90              ei_ch=getche();
91              }
92              gross=net_t_sal+t_ei;
93              printf("\n6.Gross Total Income:\t%f",gross);
94              return(gross);
95      }
```

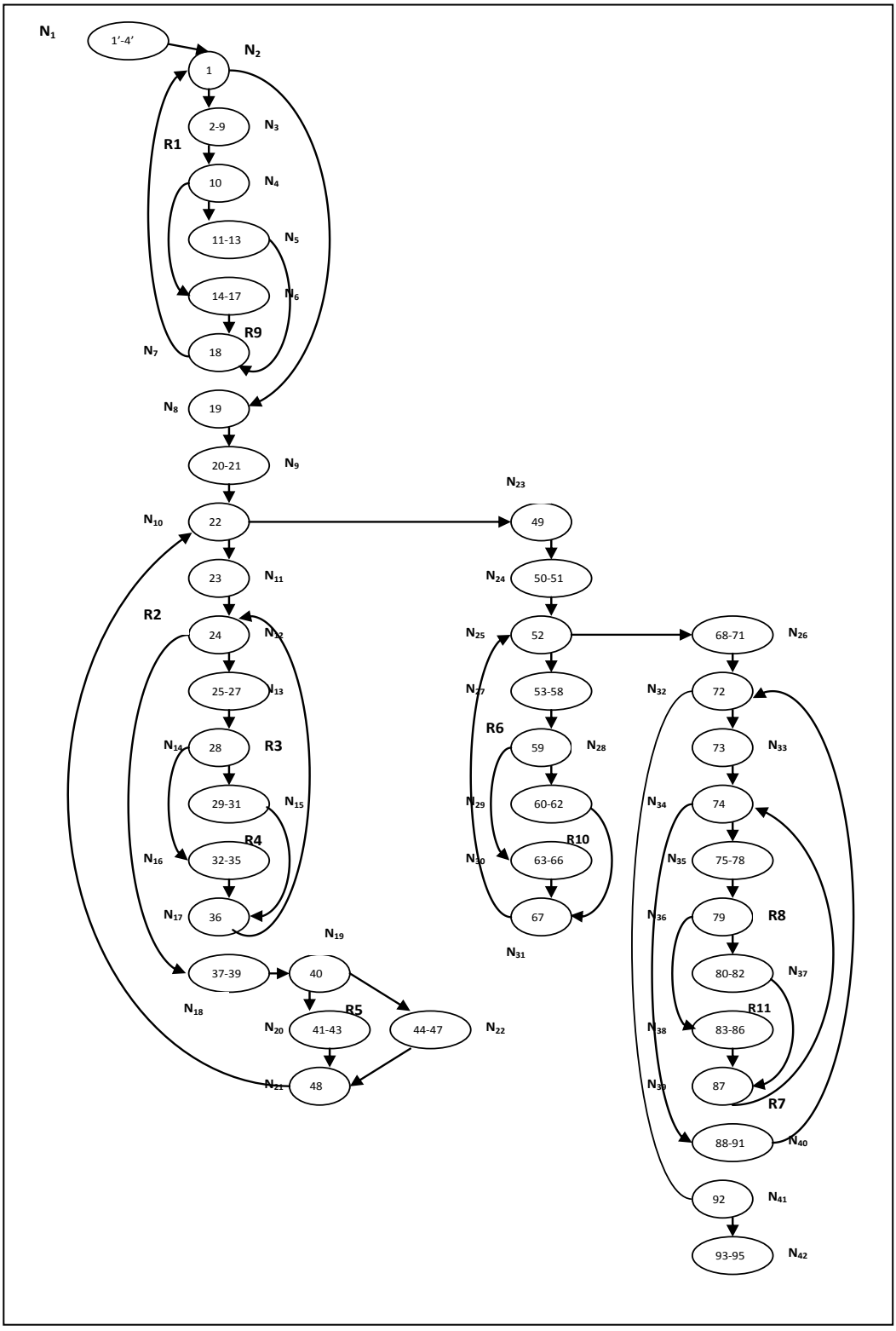


Figure 5.12: CFG for income details module

The CFG for income detail module case study is depicted in Figure 5.12. Cyclomatic complexity of the graph is 12, so there will be 12 independent paths in the graph as

shown below:

1) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
2) N_1 N_2 N_3 N_4 N_6 N_7 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
3) N_1 N_2 N_3 N_4 N_5 N_7 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
4) N_1 N_2 N_8 N_9 N_{10} N_{11} N_{12} N_{18} N_{19} N_{22} N_{21} N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
5) N_1 N_2 N_8 N_9 N_{10} N_{11} N_{12} N_{13} N_{14} N_{16} N_{17} N_{12} N_{18} N_{19} N_{22} N_{21} N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
6) N_1 N_2 N_8 N_9 N_{10} N_{11} N_{12} N_{13} N_{14} N_{15} N_{17} N_{12} N_{18} N_{19} N_{22} N_{21} N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
7) N_1 N_2 N_8 N_9 N_{10} N_{11} N_{12} N_{18} N_{19} N_{20} N_{21} N_{10} N_{23} N_{24} N_{25} N_{26} N_{32} N_{41} N_{42}
8) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{27} N_{28} N_{30} N_{31} N_{25} N_{26} N_{32} N_{41} N_{42}
9) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{27} N_{28} N_{29} N_{31} N_{25} N_{26} N_{32} N_{41} N_{42}
10) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{27} N_{28} N_{30} N_{31} N_{25} N_{26} N_{32} N_{33} N_{34} N_{40} N_{32} N_{41} N_{42}
11) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{27} N_{28} N_{30} N_{31} N_{25} N_{26} N_{32} N_{33} N_{34} N_{35} N_{36} N_{38} N_{39} N_{34} N_{40} N_{32} N_{41} N_{42}
12) N_1 N_2 N_8 N_9 N_{10} N_{23} N_{24} N_{25} N_{27} N_{28} N_{30} N_{31} N_{25} N_{26} N_{32} N_{33} N_{34} N_{35} N_{36} N_{37} N_{39} N_{34} N_{40} N_{32} N_{41} N_{42}

Table 5.14: Test Case Design for income detail case study from independent paths

Test Case ID	**Inputs**								**Expected Output**	**Independent path covered by Test Case**
	Sal 1	Sal 2	Sal 3	Sal_all	Enter more?	EA	TE	Other Income		
1	2000	4000	9000						Total 15000	1), 2), 5), 7), 8), 10), 11)
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								12000	Enter more? Y	

								12000	Enter more? N Gross Total Income: 35500	
2	2000	-300	500							2), 3)
3	2000	4000	9000						Total 15000	2), 6)
				-300					Enter correct value	
4	2000	4000	9000						Total 15000	2), 4), 5), 7)
				1000	t				Please enter y or n	
5	2000	4000	9000						Total 15000	2), 5), 7), 9)
				1000	n				Total Allowance 1000 Balance 14000	
						-100	200			
6	2000	4000	9000						Total 15000	2), 5), 7), 8), 10), 12)
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								-1000		

The definition nodes and usage nodes for different variables are shown in Table 5.15.

Table 5.15: Definition nodes and Usage nodes of variable or f income detail case study

Variable	Defined At	Used At
t_d	68	69, 70
d1	56	59, 68
d2	58	59, 68
sal1	5	10, 19
sal2	7	10, 19
sal3	9	10, 19
t_sal	19	20
sal_all	27	28, 37
sal_all_tot	21, 37	37, 49, 50
Ei	78	79, 88
t_ei	1', 88	88, 92
net_t_sal	1', 70	71, 92
Bal	50	70
sal_ch	2', 39	40
ei_ch	2', 90	72
f1	3', 34, 42	24
f2	3', 12, 16	1
f3	3', 61, 65	52
f4	3', 81, 85	74
Gross	92	93

The du and dc paths with their test case coverage are shown in Table 5.16.

Table 5.16: Du and dc paths with test coverage for income details module

Variable	du Path(beg-end)	dc?	Test case which covers this path
t_d	68-69	Yes	1,6
	68-70	Yes	1,6
d1	56-59	Yes	1,5,6
	56-68	Yes	1,6
d2	58-59	Yes	1,5,6
	58-68	Yes	1,6
sal1	5-10	Yes	1,2,3,4,5,6
	5-19	Yes	1,3,4,5,6
sal2	7-10	Yes	1,2,3,4,5,6
	7-19	Yes	1,3,4,5,6
sal3	9-10	Yes	1,2,3,4,5,6
	9-19	Yes	1,3,4,5,6
t_sal	19-20	Yes	1,3,4,5,6
sal_all	27-28	Yes	1,3,4,5,6
	27-37	Yes	1,4,5,6
sal_all_tot	21-37	No	1,4,5,6
	21-49	No	1,5,6

	21-50	No	1,5,6
	37-37	No	1,6
	37-49	Yes	1,5,6
	37-50	Yes	1,5,6
Ei	78-79	Yes	1,6
	78-88	Yes	1
t_ei	1’-88	No	1
	1’-92	No	1
	88-88	No	1
	88-92	Yes	1
net_t_sal	1’-71	No	1,6
	1’-92	No	1
	70-71	Yes	1,6
	70-92	Yes	1
Bal	50-70	Yes	1,6
sal_ch	2’-40	No	1,4,5,6
	39-40	Yes	1,4,5,6
ei_ch	2’-72	Yes	1,6
	90-72	Yes	1
f1	3’-24	Yes	1,3,4,5,6
	34-24	Yes	1,4,6
	42-24	Yes	1,6
f2	3’-1	Yes	1,2,3,4,5,6
	12-1	Yes	2
	16-1	Yes	1,3,4,5,6
f3	3’-52	Yes	1,5,6
	61-52	Yes	5
	65-52	Yes	1,6
f4	3’-74	Yes	1,6
	81-74	Yes	6
	85-74	Yes	1
Gross	92-93	Yes	1

We have total 10 faults is this program in line numbers 22,30,42,46,61,72,74,81,85,92 which are given fault Id's as F1,F2,F3,F4,F5,F6,F7,F8,F9,F10 respectively, which are shown in Table 5.17.

Table 5.17: Fault and Test Cases

Fault ID	**Fault detected by**					
	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
F1	*	-	*	*	*	*
F2	-	-	-	-	-	*
F3	*	-	-	-	*	*
F4	-	-	-	*	-	-
F5	-	-	-	-	*	-
F6	*	-	-	-	-	*
F7	*	-	-	-	-	*
F8	-	-	-	-	-	*
F9	*	-	-	-	-	-
F10	*	-	-	-	-	-

Random (unprioritized) test suite is {TC1, TC2, TC3, TC4, TC5, TC6}

APFD for random (unprioritized) test suite:

$$APFD = 1 - \frac{1+6+1+4+5+1+1+6+1+1}{6*10} + \frac{1}{2*6}$$

$$= 0.63, \text{ i.e. } 63\%.$$

After this the following changes have been made to this original program

1. Line 9a is added which results in introducing new definitions of variable 't_sal'.
2. Line 19 is modified
3. Line 51a is a new addition which has introduced new definitions of variable 't_d'.
4. Line 68 is modified.

The modified code is as follows:

```
9a      t_sal=0;
19      t_sal= t_sal+sal1+sal2+sal3;
51a     t_d=0;
```

68 t_d=t_d+d1+d2;

Now applying the proposed algorithm for this program stepwise we obtain the following:

1. Changes have introduced new definition and uses of variables ‘t_sal’ and ‘t_d’ shown in Table 5.18.

Table 5.18: New definition and uses of variables

Variable	**Defined at**	**Used at**
t_sal	9a,19	19,20
t_d	51a,68	68,69,70

2. New du paths introduced are shown the Table 5.19.
3. There is no need to design any new test case because all the new du Paths get covered by existing test cases.
4. New non-dc paths get introduced in the program (Set-1) after modifications are :{ 9a-19, 9a-20, 51a-68, 51a-69, 51a-70} as shown in Table 5.19.
   Test cases corresponding to these listed paths are {TC1, TC3, TC4, TC5, TC6}.
5. The dc paths of the original program which changed their status to non-dc after modifications in the program are :{ 19-20,68-69,68-70}
   Test cases corresponding to these listed paths (Set-2) are {TC1, TC3, TC4, TC5, TC6}
6. The test cases corresponding to non dc paths(Set-3)of the original program are:
   {TC1,TC4,TC5,TC6}
7. The remaining test cases (Set-4) in this program are :{TC2}

Table 5.19: New du paths introduced

Variable	du Path(beg-end)	Previous status dc?	New status of path dc?	Test case which covers this path
t_sal	9a-19	New path	No	1,3,4,5,6
	9a-20	New path	No	1,3,4,5
	19-20	Yes	No	1,3,4,5
t_d	51a-68	New path	No	1,6
	51a-69	New path	No	1,6
	51a-70	New path	No	1,6
	68-69	Yes	No	1,6
	68-70	Yes	No	1,6

The prioritized set after the above process is shown in the Table 5.20.

Table 5.20: Set of test cases after applying data flow TCP approach

Test cases Set	Test cases
Set-1	TC1,TC3,TC4,TC5,TC6
Set-2	TC1,TC3,TC4,TC5,TC6
Set-3	TC1,TC4,TC5,TC6
Set-4	TC2

Since set 1 and set 2 contains same no. of test cases and set 3 is a subset of set 1and set 2 ,the test suite consists of the test cases as TC1,TC3,TC4,TC5,TC6,TC2. However these test cases are not prioritized. So to prioritize the test cases within a set, algorithm shown in Figure 5.2 is applied.

After analysing case study 3, we obtain the following data as shown in the Table 5.21.

Table 5.21: Data obtained after analyzing the income detail case study

Test case id	Total newly introduced non-dc paths covered in modified program	Total paths covered which has changed from dc to non-dc in modified program	Total lines covered in the modified program
TC1	5	3	83
TC2	0	0	18
TC3	2	1	35
TC4	2	1	45
TC5	2	1	59
TC6	4	2	75

Finally the prioritized test suite for this case study after applying the algorithm (See Figure 5.7) is:

{TC1, TC6, TC5, TC4, TC3, TC2}

Now the APFD value for the prioritized test suite is:

$$APFD = 1- \frac{1+2+1+4+3+1+1+2+1+1}{6*10} + \frac{1}{2*6}$$

=0.80, i.e. 80%.

Further, the modified program when applied to the previous approach [119] of Yogesh Singh et. al, the prioritized order of test cases obtained is TC1, TC3, TC4, TC5, TC6, TC2.

The APFD for the test suite obtained using previous approach is:

$$APFD = 1- \frac{1+5+1+3+4+1+1+5+1+1}{6*10} + \frac{1}{2*6}$$

=0.70, i.e. 70%.

From the above calculations it is clear that prioritized test suite with the proposed approach gives better APFD value as shown in Figure 5.13.

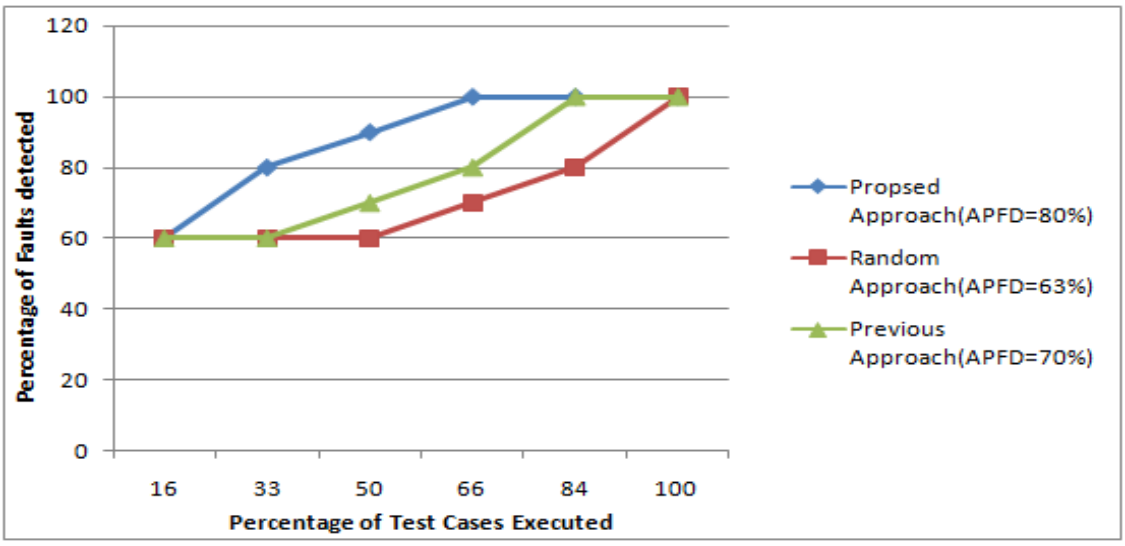


Figure 5.13: Comparison of Random, Previous and Proposed data flow testing approach

5.4 CONTROL-STRUCTURE-WEIGHTED TEST CASE PRIORITIZATION (CSWTCP) TECHNIQUE

After prioritizing test cases for modified program on the basis of newly introduced non-dc paths and changed status of existing dc paths to non-dc paths, there remains a big set of test cases corresponding to dc paths which are given the same priority. So next step is to make a criterion to prioritize the left out same priority test cases to make test suite more effective. For this purpose, a control-structure weighted test case prioritization method is being proposed in this work .In this method complexity of the statements where the variable has been used is taken considering various aspects of structure of programming.

5.4.1 Proposed CSWTCP Approach

In this subsection, a test case prioritization technique has been proposed for the case when a program has been modified. The prioritization is based on the nature of statements where the modification has taken place. In order to accomplish this task, a control-structure weighted test case prioritization method is being proposed in this work. In the method complexity of the statements is taken considering various aspects of structure of programming.

In the proposed technique, Variable Dependence Graph (VDG) is prepared for variables whose definition or use has changed in new version of software and directly and indirectly, affected variables by these changed variables is listed out. This is followed by listing the du paths of these variables and giving weights to the paths considering following factors:

- Type of control structure present in the statement where variables are used.
- Number of Boolean conditions present in the statement.
- Number of p-use statements present in the path.
- Nesting level of the statement.
- Nesting type of the statement.

This is followed by creating a sorted list of du-paths according to the calculated weight based on above factors. The test cases corresponding to these sorted du-paths are given order to make a prioritized test suite. The complete procedure of applying this method is shown in Figure 5.14 and various algorithms are explained in further sections of the chapter.

Begin
Step1: *Notify the changes in the new version of program as compared to old version.*

Step 2: *Note the line numbers in the code of new version where the use of variable has been changed.*

Step 3: *Make a VDG by using algorithm (See Figure 5.15).*

Step 4: *Looking at the VDG, find out different types of variables in the graph and put the 'weight of every node' (WV) in VDG according to the algorithm (See Figure 5.18).*

Step 5: *Make a list of all the variables in the VDG and their DU-dc paths.*

Step 6: *Calculate the 'weight of modified statement' (WMS) considering four factors according to the procedure explained in section 5.4.4..*

Step 7: *Calculate the weight of du-paths by using WMS and two more factors according to the procedure explained in section 5.4.5.*

Step 8: *Make a sorted list du paths according to the weight of du-path (WDU) thus calculated*

Step 9: *Make a set of prioritized test cases corresponding to sorted du-path list.*

End

Figure 5.14: Process of CSWTCP

5.4.2 Preparing a VDG

In this section the procedure for preparing a VDG has been explained. The proposed algorithm works on the statements of a program and finds the assignment statements. After this it looks for the variables and makes nodes correspondingly. The proposed algorithm for making VDG is shown in Figure 5.15.

Begin
Step1. Scan the modified statements where assignment is being used.

Step2. Go to first assignment statement.

Step3. Make a node for the variable on the left side.

Step4. Note down the variables on the right side of the statement

Step5. Make nodes for these variables as children of the parent node made in step 3

Step6. Go to next assignment statement.

Step7. Repeat step 3-6.
End

Figure 5.15: Algorithm for making VDG

The following sample program exemplifies the above algorithms. The VDG corresponding to code shown in Figure 5.16 is shown in Figure 5.17.

```
void main()
{
int x,y,z,k;
x = 5;
y = 2;
z = x + y;
k = z * 2 ;
printf("Value of k is = %d", k);
}
```

Figure 5.16: Code for Sample program

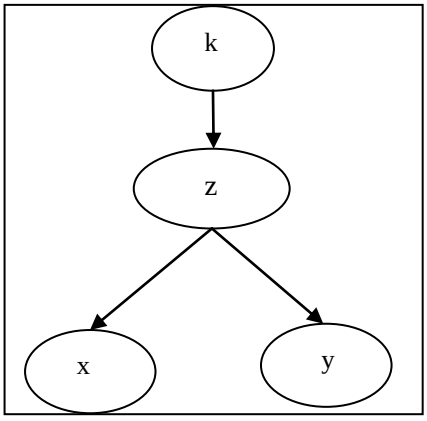


Figure 5.17: VDG for sample program

5.4.3 Calculation of the Weight of a Node in VDG

To calculate the weight of a node in VDG, two types of variables in VDG are taken into consideration as given below:

(i) **Directly Changed Variables:** Variables which are at most one edge away from the changed variable node.

(ii) **Affected Variables:** Variables which are more than one edge away from the changed variable node.

The proposed algorithm for calculating the weight of node in VDG is shown in Figure 5.18.

Begin
Step 1. Note down the modified variables.

Step 2. Put the weight 1 (one) for directly changed variables in the VDG.

Step3. Put the weight half of its immediate parent node at each above level from the directly changed variables.

Step 4. If a node has more than one child then add up the weight because of all these nodes.

Step 5. Repeat steps 3 and 4 till the weight is assigned to all the nodes in the VDG.
End

Figure 5.18: Algorithm for assigning the weight of nodes in VDG

The weights assigned to the different nodes using algorithm (See Figure 5.18) for the VDG shown in Figure 5.17 are shown in Figure 5.19 and also the assigned values are shown in Table 5.22.

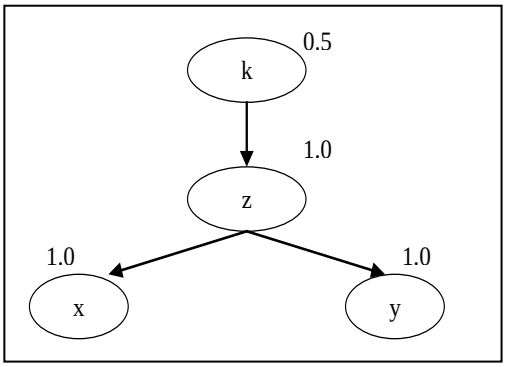


Figure 5.19: VDG with assigned weights to nodes for the sample program

Table 5.22: Weight assigned to the different nodes of VDG of Figure 5.17

Variable Name	**Status in VDG**	**Node weight in VDG**
X	Directly changed variables	1
Y	Directly changed variables	1
Z	Affected variables	0.5+0.5=1.0
K	Affected variables	0.25+0.25=0.5

5.4.4 Calculating the Weight of Modified Statement (WMS)

To calculate the weight of modified statement (WMS) the following four factors are proposed.

1. Control structure present in the statement
2. Number of Boolean conditions present in the statement
3. Nesting level of statement
4. Nesting type of the statement

The procedure for calculating WMS considering all above mentioned factors is explained below:

- **Weight of control structure (WC):** The control structure present in the statement is assigned a weight according to the Table 5.23.

Table 5.23: Proposed control structure weights

Sr. No.	Control structure	Weight
1	IF-THEN-ELSE	1
2	SWITCH-CASE	0.5
3	WHILE-DO	0.1
4	DO-WHILE	0.11
5	RECURSION	0.01
6	NO CONTROL STRUCTURE	–

- **Boolean conditions weight (NB):** NB depends upon the number of Boolean conditions present in the statement where variable is being used. So the weight assigned is equal to the number of boolean conditions present in the statement i.e., If n number of boolean conditions are present, then weight assigned is n.

- **Nesting level weight (NLW):** NLW depends upon the nesting level of statement in structure of program. Its value is assigned as 10^{-n+1} where 'n' is the nesting level of the statement.

- **Nesting type weight (WNLT):** WNLT depends upon the nesting type of the statement where variable is being used program. Its value is assigned as per weight given in Table 5.24.

Table 5.24: Proposed nesting type weight

Sr. No.	Nesting level type	Weight
1	Loop under loop	0.5
2	condition statement under loop	1.0
3	loop under conditional statement	1.5
4	condition statement under condition statement	2.0
5	c-use statement under loop	2.5
6	c-use statement under main /No nesting	3.0

The total weight (WT) of all the four factors considered above is calculated using Formula 5.2.

$$WT = WC*NB*NLW*WNLT \quad \text{---------- (5.2)}$$

Weight of Modified statement (WMS) is calculated according using Formula 5.3.

$$WMS = C*e^{-WT} \quad \text{---------- (5.3)}$$

Here C is a constant, its value is taken as 10.

5.4.5 Calculating the Weight of du Paths (WDU)

Weight of du-paths (WDU) depends upon two more following factors besides WMS. These proposed factors are:

- P-use statements in du path
- Directly changed and Affected Variables in VDG

The criteria for assigning weight to these two actors for finally calculating the weight of du-path is explained below:

- **P-use statements weight (WPU):** WPU depends upon the no. of p-use statements present in the du path. Its value is taken as equal to no. of p-use statements present in the du-path.
- **Directly changed and Affected Variables Weights in VDG (WV):** WV is calculated according to the algorithm (See Figure 5.18).

Considering all these factors and the weights assigned to these factors, the final weight of a du-path is calculated by using Formula 5.4 given below:

$$WDU = WMS + WPU + WV \quad \text{---------- (5.4)}$$

Thus du-paths on the same priority level have been considered for prioritization based on control structure weights and their corresponding complexity.

5.4.6 Evaluation & Analysis of CSWTCP approach

For analyzing this proposed approach the case study discussed section 5.3.1.3 has been taken. A tool is implemented in PHP language. All the values and implementations shown in this section for this case study have been taken with the help of this tool.

The given case study is modified and the changes with their line numbers in the modified case study are as follows:

16 t_sal=0;

26 t_sal= t_sal+sal1+sal2+sal3;

60 t_d=0;

77 t_d=t_d+d1+d2;

First of all VDG for the program considering changed variable is created as shown in Figure 5.20.

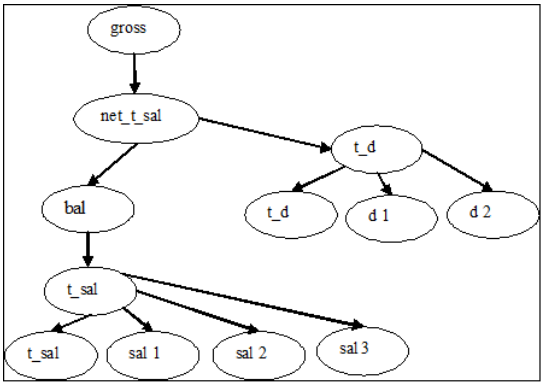


Figure 5.20: VDG for modified program

Then the nodes in this VDG are assigned weights according to the algorithm (See Figure 5.18) and the weights assigned are shown in Figure 5.21 and Table 5.25.

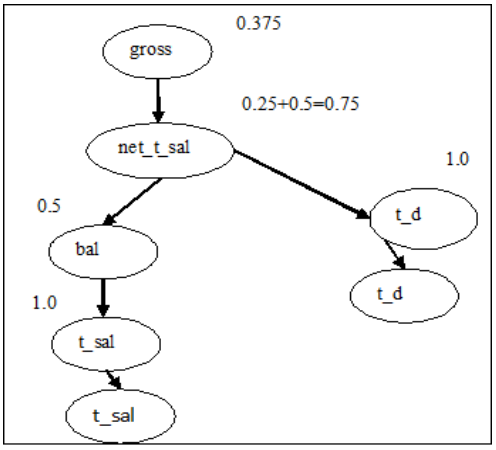


Fig. 5.21: VDG for modified program with assigned weights of nodes

Table 5.25: Directly changed variables (DCV) & affected variables (AV) and their node weights in VDG

Variable Name	**Status of variable in VDG**	**Node weight in VDG**
t_sal	DCV	1
t_d	DCV	1
Bal	AV	0.5
net_t_sal	AV	0.25+0.5=0.75
Gross	AV	0.125+0.25=0.375

The list of du paths of the changed and affected variables shown in VDG (see Figure 5.20) is prepared as shown in Table 5.26.

Table 5.26: List of du paths of DCV & AV

Variable name	**du-path**
Bal	59-60
	59-80
Gross	102-103
	102-104
net_t_sal	3-81
	3-102
	80-81
	80-102
t_d	61-78
	61-79
	61-80
	78-78
	78-79
	78-80
t_sal	17-27
	17-28
	17-59
	27-27
	27-28
	27-59

The WT for these du paths is calculated considering four factors by using Formula 5.2. The values of these four factors and WT is shown in Table 5.27.

Table 5.27: Weights of different factors and WT for du paths

Var name	**Du path**	**WC**	**NB**	**NLW**	**WNLT**	**Total weight(WT)**
Bal	59-60	-	-	1	-	1
Bal	59-80	-	-	1	-	1
Gross	102-103	-	-	1	-	1
Gross	102-104	-	-	1	-	1
net_t_sal	3-81	-	-	1	-	1
net_t_sal	3-102	-	-	1	-	1
net_t_sal	80-81	-	-	1	-	1
net_t_sal	80-102	-	-	1	-	1
t_d	61-78	-	-	1	-	1
t_d	61-79	-	-	1	-	1
t_d	61-80	-	-	1	-	1
t_d	78-78	-	-	1	-	1
t_d	78-79	-	-	1	-	1
t_d	78-80	-	-	1	-	1
t_sal	17-27	-	-	1	-	1
t_sal	17-28	-	-	1	-	1
t_sal	17-59	-	-	1	-	1
t_sal	27-27	-	-	1	-	1
t_sal	27-28	-	-	1	-	1
t_sal	27-59	-	-	1	-	1

The WMS for the modified statements is calculated using WT. Table 5.28 shows the values for the WMS calculated using values of WT for different du-paths using Formula 5.3.

Table 5.28: Calculated WMS Values using WT value

Var name	**Du path**	**Total weight(WT)**	**C =10**	**e^{-wt} e=2.71828**	**WMS**
bal	59-60	1	10	0.3678	3.67
bal	59-80	1	10	0.3678	3.67
gross	102-103	1	10	0.3678	3.67
gross	102-104	1	10	0.3678	3.67
net_t_sal	3-81	1	10	0.3678	3.67
net_t_sal	3-102	1	10	0.3678	3.67
net_t_sal	80-81	1	10	0.3678	3.67
net_t_sal	80-102	1	10	0.3678	3.67
t_d	61-78	1	10	0.3678	3.67
t_d	61-79	1	10	0.3678	3.67
t_d	61-80	1	10	0.3678	3.67
t_d	78-78	1	10	0.3678	3.67
t_d	78-79	1	10	0.3678	3.67
t_d	78-80	1	10	0.3678	3.67
t_sal	17-27	1	10	0.3678	3.67
t_sal	17-28	1	10	0.3678	3.67
t_sal	17-59	1	10	0.3678	3.67
t_sal	27-27	1	10	0.3678	3.67
t_sal	27-28	1	10	0.3678	3.67
t_sal	27-59	1	10	0.3678	3.67

Then WDU(weight of du-path) is calculated and du paths are arranged according to these weights and test cases are arranged in test suite according to this sorted list of weights of du paths.

Table 5.29 shows the WDU (weight of du-path) values, calculated using values of WMS, WPU and WV by using the Formula 5.4.

Table 5.29: WDU Values for various du paths

Var name	**Du path**	**WMS**	**WPU**	**WV**	**WDU**
bal	59-60	3.67	0	0.5	4.17
bal	59-80	3.67	2	0.5	6.17
gross	102-103	3.67	0	0.375	4.05
gross	102-104	3.67	0	0.375	4.05
net_t_sal	3-81	3.67	8	0.75	12.42
net_t_sal	3-102	3.67	11	0.75	15.42
net_t_sal	80-81	3.67	0	0.75	4.42
net_t_sal	80-102	3.67	3	0.75	7.42
t_d	61-78	3.67	2	1	6.67
t_d	61-79	3.67	2	1	6.67
t_d	61-80	3.67	2	1	6.67
t_d	78-78	3.67	0	1	4.67
t_d	78-79	3.67	0	1	4.67
t_d	78-80	3.67	0	1	4.67
t_sal	17-27	3.67	1	1	5.67
t_sal	17-28	3.67	1	1	5.67
t_sal	17-59	3.67	5	1	9.67
t_sal	27-27	3.67	0	1	4.67
t_sal	27-28	3.67	0	1	4.67
t_sal	27-59	3.67	4	1	8.67

Now a sorted list of du paths is prepared by arranging du paths in the descending values of WDU for them as shown in Table 5.30.

Table 5.30: List of du paths arranged in descending values of WDU

Variable used	**Du path**	**WDU**	**Test Cases Covered**
net_t_sal	3-102	15.42	1
net_t_sal	3-81	12.42	1,6
t_sal	17-59	9.67	3,4,5
t_sal	27-59	8.67	3,4,5
net_t_sal	80-102	7.42	1,6
t_d	61-78	6.67	1,6
t_d	61-79	6.67	1,6
t_d	61-80	6.67	1,6
Bal	59-80	6.17	1,6
t_sal	17-27	5.67	3,4,5
t_sal	17-28	5.67	3,4,5

t_sal	27-27	4.67	3,4,5
t-sal	27-28	4.67	3,4,5
t_d	78-78	4.67	1,6
t_d	78-79	4.67	1,6
t_d	78-80	4.67	1,6
net_t_sal	80-81	4.42	1,6
Bal	59-60	4.17	1,6
Gross	102-103	4.05	1
Gross	102-104	4.05	1

Since some du paths are covered by multiple test cases ,the final weight of test case after the modification has been taken place in the program is obtained by adding the corresponding weight of du paths and are shown in Table 5.31.

Table 5.31: Test cases and their weights

Test Cases	**Total Weight**
TC1	92.14
TC3	39.02
TC4	39.02
TC5	39.02
TC6	68.62

On the basis of the weights shown in Table 5.31, we can prioritize the test cases as, TC1,TC6,TC3,TC4,TC5. Since TC2 is not covering any modified path, so in the test suite it is placed at the least priority. Hence the final prioritized test suite is.

{TC1, TC6, TC3, TC4, TC5, TC2}

To measure the effectiveness of the proposed CSWTCP approach, the faults were taken in the sample program [80]. We have total 10 faults is this program in line numbers 22,30,42,46,61,72,74,81,85,92 which are given fault Id's as F1,F2,F3,F4,F5,F6,F7,F8,F9,F10 respectively, shown in Table 5.32.

Table 5.32: Faults and Test Cases

Fault ID	**Fault detected by**					
	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
F1	*	-	*	*	*	*
F2	-	-	-	-	-	*
F3	*	-	-	-	*	*
F4	-	-	-	*	-	-
F5	-	-	-	-	*	-
F6	*	-	-	-	-	*
F7	*	-	-	-	-	*
F8	-	-	-	-	-	*
F9	*	-	-	-	-	-
F10	*	-	-	-	-	-

Random (unprioritized) test suite is {TC1, TC2, TC3, TC4, TC5, TC6}

APFD for random (unprioritized) test suite:

$$APFD = 1- \frac{1+6+1+4+5+1+1+6+1+1}{6*10} + \frac{1}{2*6}$$

$$= 63.30\%.$$

The prioritized test suite is: {TC1, TC6, TC3, TC4, TC5, TC2}

Now the APFD value for the prioritized test suite is:

$$APFD = 1- \frac{1+2+1+4+5+1+1+2+1+1}{6*10} + \frac{1}{2*6}$$

$$=76.70\%.$$

From the above calculations it is clear that the proposed approach gives better APFD value as shown in Figure 5.22.

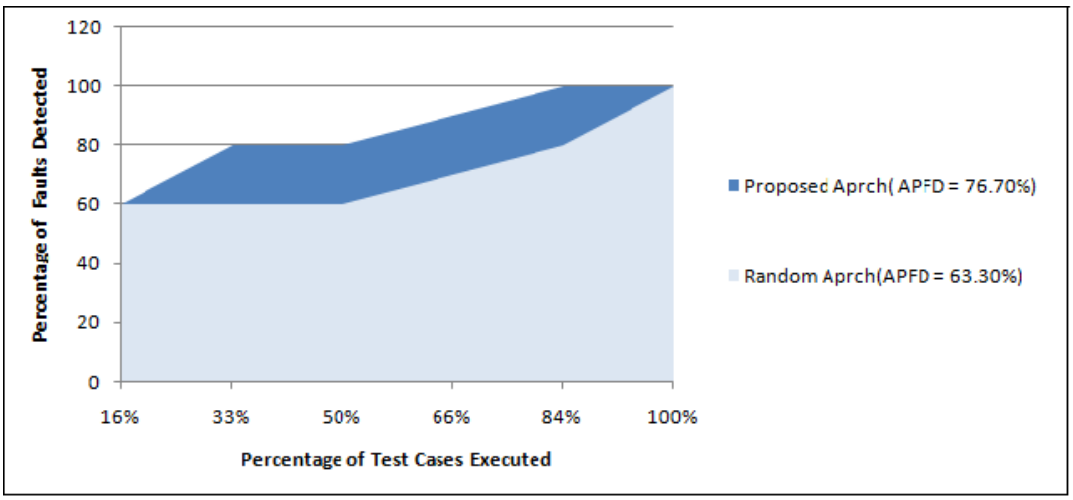


Figure 5.22 Comparison of APFD values of Random and CSWTCP Approach

5.5 CONCLUSION

In this chapter three techniques for regression test case prioritization has been discussed. The first technique is based on module coupling information among the modules. The proposed technique helps in finding the badly affected module due to change in a module. The second technique prioritizes the test cases while performing regression testing using data flow testing concepts. The third approach is control structure weighted test case prioritization technique which is the extension of the second approach. The proposed approaches have been applied on certain case studies and the results have been validated.

Chapter VI

CONCLUSIONS AND FUTURE SCOPE

6.1 CONCLUSIONS

This chapter presents the achievements of this research and lists the scope of future work. The outcome of this research contributed in designing of various techniques in the area of test case prioritization and development of various tools for the proposed techniques have been designed. This research will help the software testers in minimizing the efforts and cost incurred in software testing process.

6.2 BENIFITS OF THE PROPOSED WORK

- **Identification of the Badly Affected Module**

  The work proposed in this thesis will help the testes in finding the badly affected module due to change in one module which results in reducing the efforts and time incurred in software testing process. Once the badly affected module has been identified, the test cases for this particular module can be prioritized.

- **Managing Risks in Software Projects through Test Case Prioritization**

  The ultimate goal of the test case prioritization process is the early fault detection. The identification of critical bugs at early stages of development process helps in managing the risks associated with a software project.

- **Tools for Test Case Prioritization**

  To help the software testers during the process of software testing, some tools for the proposed test case prioritization techniques have been designed. These tools will help the testers in prioritizing the test cases for system testing and at regression test levels.

6.3 FUTURE SCOPE

The work presented in this thesis can be extended with the following list of possible future research issues.

- **Test Case Prioritization for Object Oriented Software**

  The present work has been tested with procedural programs. The next step may be to identify the factors in Object oriented paradigm so that the proposed test case prioritization techniques can be extended to object oriented software.

- **Testing the Proposed Techniques for the large projects**

  The proposed test case prioritization techniques have been tested on small projects. It would be better if these are applied on large scale industry projects.

- **Acceptance Test Case Prioritization**

  In this thesis the test case prioritization process has been done at unit, system and regression testing levels. But there may be large number of tests cases while performing acceptance testing. The future work may be related to analyze the factors that must be considered for acceptance testing and thereby helps in prioritizing the test cases in acceptance testing.

APPENDIX-A

Source Code of Employee Record Case Study

```
1.  /*
2.  *C program to creat employee file
3.  */
4.  #include<stdio.h>
5.  #include<stdlib.h>
6.  #include<errno.h>
7.  #include<string.h>
8.  .
9.  struct  emprec
10. {
11. Int empid;
12. char *name;
13. };
14. typedef  struct  emprec  emp;
15. .
16. void insert (char *a);
17. void display (char *a);
18. void update (char *a);
19. int count;
20. void main(int argc, char *argv[]);
21. {
22. int choice;
23. .
24. while(1)
25. {
26. printf("enter the choice\n");
27. printf("1. Insert a new record\n2. Display the record");
28. printf("3. Update a record");
29. scanf("%d",&choice);
30. switch(choice)
31. {
```

```
32. case 1:
33. insert(argv[1]);
34. break;
35. case 2:
36. display(argv[1]);
37. break;
38. case 3:
39. update(argv[1]);
40. break;
41. case 4:
42. exit(0);
43. default :
44. printf(“enter the correct choice\n”);
45. }
46. }
47. }
48. .
49. /* to insert a new record into the file*/
50. void insert (char *a)
51. {
52. FILE *fp1;
53. emp *temp1=(emp*)malloc(size of(emp));
54. temp1->name=(char*)malloc(200*size of (char));
55. .
56. fp1=fopen(a,”a+”);
57. if(fp1==NULL);
58. perror(“”);
59. else
60. {
61. printf(“enter the emplyeee id\n”);
62. scanf(“%d”,&temp1->empid);
63. fwrite(“&temp1->empid,sizeof(int),1,fp1”);
64. printf(“enter  employee name\n”);
65. scanf(“%[^\n]s”,temp1->name);
```

```
66. fwrite(temp1->name,200,1,fp1);
67. count++;
68. }
69. fclose(fp1);
70. free(temp1);
71. free(temp1->name);
72. }
73. .
74. /*to display the records
75. void diplay(char *a)
76. {
77. FILE fp1;
78. char ch;
79. int var=count;
80. emp *temp=(emp*)malloc(sizeof(emp));
81. temp->name=(char *)malloc(200*sizeof(char));
82. .
83. fp1=fopen(a,"r");
84. if(count==0)
85. {
86. printf("no record to display\n");
87. return;
88. }
89. if(fp1==NULL)
90. perror(" ");
91. else
92. {
93. while(var)
94. {
95. fread(&temp->empid, sizeof(int),1,fp1);
96. printf("%d"temp->empid);
97. fread(temp->name,200,1,fp1);
98. printf("%s\n",temp->name);
99.          var--;
```

```
100.        }
101.        }
102.        fclose(fp1);
103.        free(temp);
104.        free(temp->name);
105.        }
106.        .
107.        /*to update the given record*/
108.        void update(char *a)
109.        {
110.        FILE *fp1;
111.        char ch, name[200];
112.        int var=count,id,c;
113.        emp *temp=(emp *)malloc(sizeof (emp));
114.        temp->name= (char *)malloc(200*sizeof(char));
115.        .
116.        fp1=fopen(a,"r++");
117.        if(fp1==NULL)
118.        perror(" ");
119.        else
120.        {
121.        while(var)
122.        {
123.        fread(&temp->empid, sizeof(int),1,fp1);
124.        printf("%d",temp->empid);
125.        fread(temp->name,200,1,fp1);
126.        printf("%s\n",temp->name);
127.        var--;
128.        }
129.        printf("enter which employee id to be updated\n");
130.        scanf("%d",&id);
131.        fseek(fp1,0,0);
132.        var=count;
133.        while(var)
```

```
{
fread(&temp->empid, sizeof(int),1,fp1);
if(id==temp->empid)
{
printf("enter employee name for update");
scanf("%[^\n]s",name);
c=fwrite(name,200,1,fp1)
break;
}
fread(temp->name,200,1 ,fp1);
var--;
}
if(c==1);
pritnf("update successful\n");
else
printf("update unsuccessfull\n");
fclose(fp1);
free(temp);
free(temp->name);
}
}
```

APPENDIX-B

Source Code of Saving Module

```
float savings()
1       char saving_type[20];
2       float amount,total=0;
3       int flag1=1,flag2=1,i;
4       char sav_ch='y';
5       while((sav_ch=='y')||(sav_ch=='Y'))
6       {
7       while(flag1==1)
8               {
9               printf("\nEnter Saving type\t:");
10              gets(saving_type);
11              for(i=0;i<strlen(saving_type);i++)
12                      {
13      if(((toascii(saving_type[i])>= 65) && (toascii(saving_type[i]) <= 122)) ||
            (toascii(saving_type[i]) == 32))
14                              {
15                              flag1=0;
16                              }
17                      else
18                              {
```

```
19                          printf("\nSaving type can contain only charcter Error at
                            position number %d",i);
20                          flag1=1;
21                          break;
22                          }
23                  }
24          if((strlen(saving_type)<3)||(strlen(saving_type)>20))
25                  {
26                  printf("\nPlease enter between 3 to 20 characters ");
27                  flag1=1;
28                  }
29          }
30  while(flag2==1)
31          {
32          printf("\nEnter Amount\t:");
33          scanf("%f",&amount);
34          if(amount>0)
35                          {
36                          flag2=0;
37                          }
38                  else
39                          {
40                          printf("\nAmount cannot be less than or equal to 0");
```

```
41                          flag2=1;
42                          }
43          }
44          printf("\n\nPress any key to proceed");
45          getch();
46          clrscr();
47          patt("SAVING Details");
48          printf("\nSAVING TYPE\t:%s",saving_type);
49          printf("\nAMOUNT\t\t\t:%f",amount);
50          total=total+amount;
51          printf("\nDo you want to enter more(y for yes)\t:");
52          sav_ch=getche();
53          flag1=1;
54          flag2=1;
55          }
56          printf("\nTotal\t\t:%f",total);
57          return(total);
58}
```

APPENDIX-C

Source code of infix to postfix conversion

```
#include <stdio.h>

#include <stdlib.h>

Int  top = 10;

//create a structure called node

1. struct node
2. {
3. char ch;
4. struct node *next;
5. struct node *prev;
6. }  *stack[11];

//type define the structure

typedef struct node node;

 //Create a function for push

1. void push(node *str)
2. {
3. if (top <= 0)
4. printf("Stack is Full ");
5. else
6. {
7. stack[top] = str;
8. top--;
9. }
10. }

 //create a function called pop which return a node pointer

1. node *pop()
```

```
2. {
3. node *exp;
4. if (top >= 10)
        a. printf("Stack is Empty ");
5. else
        b. exp = stack[++top];
6. return exp;
7. }
```

// The convert function takes the expression as the input and converts it

```
1. void convert(char exp[])
2. {
3. node *op1,  *op2;
4. node *temp;
5. int i;
6. for (i=0;exp[i]!='\0';i++)
7. if (exp[i] >= 'a'&& exp[i] <= 'z'|| exp[i] >= 'A' && exp[i] <= 'Z')
8. {
9. temp = (node*)malloc(sizeof(node));
10. temp->ch = exp[i];
11. temp->next = NULL;
12. temp->prev = NULL;
13. push(temp);
14. }
15. else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/' ||
16. exp[i] == '^')
17. {
```

```
18. op1 = pop();
19. op2 = pop();
20. temp = (node*)malloc(sizeof(node));
21. temp->ch = exp[i];
22. temp->next = op1;
23. temp->prev = op2;
24. push(temp);
25. }
26. }
```

// The display function displays the expression

```
1. void display(node *temp)
2. {
3. if (temp != NULL)
4. {
5. display(temp->prev);
6. printf("%c", temp->ch);
7. display(temp->next);
8. }
9. }
```

//Finally the main function

```
1. void main()
2. {
3. char exp[50];
4. clrscr();
5. printf("Enter the postfix expression :");
6. scanf("%s", exp);
```

```
7.  convert(exp);
8.  printf("\nThe Equivalant Infix expression is:");
9.  display(pop());
10. printf("\n\n");
11. getch();
12. }
```

APPENDIX-D

Source code of Case Study Software

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
//Structure Declaration
struct emp
{
int empnum;
char empname[20];
float salary;
};
typedef struct emp ER;
ER e1;
//Global variable declaration
int sum=1;
float pro_inc;
void main()
{
clrscr();
//Prototype declaration
int addnum(int,int);
ER record_change(ER);
void work(int);
void sum_len_strings();
```

```
void relevant1(int,int);
//variable declaration
int n1,n2,result,flag,len,n,a;
//example of Data Coupling
printf("\n Enter two numbers(integers)");
scanf("%d %d",&n1,&n2);
result=addnum(n1,n2);
printf("Modified value of %d and %d = %d",n1,n2,result);
//example of Stamp Coupling
printf("\n ENTER THE EMPLOYEE INFORMAION:\n\n");
printf("Enter Employee number(between 1-50)\n");
scanf("%d",&e1.empnum);
printf("Enter Employee name(atleast 5 character)\n");
scanf("%s",&e1.empname);
printf("Enter Employee salary(five figures)\n");
scanf("%f",&e1.salary);
printf("\n\nEmployee information before change is:\n");
printf("\nEMPLOYEE NUMBER = %d",e1.empnum);
len=strlen(e1.empname);
if(len>=5)
printf("\nEMPLOYEE NAME = %s",e1.empname);
else
printf("\nEMPLOYEE NAME = Defaulter");
printf("\nEMPLOYEE SALARY = %7.2f",e1.salary);
e1=record_change(e1);
printf("\n\nEmployee information after change is:\n");
printf("\n(NEW) EMPLOYEE NUMBER = %d\t",e1.empnum);
```

```
printf("\n(NEW) EMPLOYEE SALARY = %7.2f",e1.salary);
//example of control coupling
printf("\n Enter flag(between 1-4) value for the employee");
scanf("%d",&flag);
work(flag);
//example of common coupling
sum_len_strings();
//example of data coupling
printf("\n Enter number of Entries in a match=");
scanf("%d",&n);
printf("\n Enter interested candidates=");
scanf("%d",&a);
relevant1(n,a);
getch();
}
//Module 2
int addnum(int val1,int val2)
{
  sum=val1+val2+sum;
  return(sum);
}
//Module 3
ER record_change(ER e1)
{
ER e2;
if(e1.empnum<=50)
e2.empnum=1000+e1.empnum;
```

```
else
e2.empnum=0;
if(e1.salary>=10000)
e2.salary=10*e1.salary;
else
e2.salary=0;
return(e2);
}
//Module 4
void work(int f)
{
void rules_for_job();
void budget_company();
void producers_details(float);
void items_purchased();
switch(f)
{
case 1:printf("\n HR department");
      rules_for_job();
      break;
case 2:printf("\n FINANCE department");
      budget_company();
      break;
case 3:printf("\n PRODUCTION department");
      pro_inc=2.35;
      producers_details(pro_inc);
      break;
```

```
case 4:printf("\n PURCHASE department");
      items_purchased();
      break;
default: printf("\n you havent entered the correct value");
}
}
//Module 7
void rules_for_job()
{
printf("\n Rule1: Qualification Graduade");
printf("\n Rule2: Computer knowledge");
printf("\n Rule3: Percentage >75%");
}
//Module 8
void budget_company()
{
float total;
float salaries,infra,maint;
printf("\n Enter cost of salaries,infrastructure,maintenance");
scanf("%f %f %f",&salaries,&infra,&maint);
if(salaries>=50000)
{
total=salaries+infra+maint;
printf("\nCost=%7.2f",total);
}
else
{
```

```
total=0;
printf("\nCost=%7.2f",total);
}
}
//Module 9
void producers_details(float pro_inc)
{
printf("\n producers are 200 in number");
printf("\n producers are from NCR region");
printf("\n for a successful producer 10 years experience is required");
printf("\n %f lakhs",pro_inc);
}
//Module 10
void items_purchased()
{
int no;
float tot,cost;
printf("\n enter no. of item to be purchased");
scanf("%d",&no);
printf("\n enter cost of each item");
scanf("%f",&cost);
if(no>=1)
{
tot=no*cost;
printf("\nTotal money spent=%7.2f",tot);
}
else
```

```
{
tot=0*cost;
printf("\nTotal money spent=%7.2f",tot);
}
}
//Module 5
void sum_len_strings()
{
void producers_details(float);
char firstn[10];
char lastn[10];
int l1,l2,i,ch,flag;
printf("\n Enter first name of the employee(Only English Alphbets)");
scanf("%s",&firstn);
printf("\n Enter last name of the employee(Only English Alphabets)");
scanf("%s",&lastn);
l1=strlen(firstn);
l2=strlen(lastn);
for(i=0;i<=l1-1;i++)
{
ch=isalpha(firstn[i]);
 if(ch!=0)
  flag=1;
 else
  flag=0;
}
```

```
for(i=0;i<=l2-1;i++)
{
ch=isalpha(lastn[i]);
 if(ch!=0)
  flag=1;
 else
  flag=0;
}
if(flag==1)
{
sum=l1+l2+sum;
printf("Length of FULL NAME = %d",sum);
producers_details(pro_inc);
}
else
printf("\n You havent entered first name and last name correctly");
}

//Module 6
void relevant1(int n,int a)
{
int s,b,i,x;
if(a>=0 && n>0)
{
x=1;
b=a+x;
a=a+1;
```

```
i=1;
s=0;
while(i<=n)
{
 if(b>0)
 {
        if(a>1)
        {
        x=2;
        }
 }
 s+=x;
 i++;
}
printf("\nVALUE =%d",s);
}
else
printf("\n not possible");
}
```

APPENDIX-E

A research survey was done while working on this thesis. The Questionnaire was prepared and distributed to a group of researchers, students, faculty members and software developers. In total, we received 120 responses. The details regarding the Questionnaire prepared and its result analysis are given here.

Survey for Ph.D. work

While doing structured programming there are various factors which have a great potential of introducing the errors in the program. I have pointed out the following factors which are given in the table below. You are kindly requested to spare your valuable time for providing the weights to these factors on a scale from 0 to 1. Here weights represent the potential of a factor to introduce error in the program while performing structured programming.

Sr. No.	Factors	0≤Weight< 0.2	0.2≤Weight <0.5	0.5≤Weight<0.8	0.8≤Weight<1
1.	Line of Code				
2.	Type Casting				
3	Predicate Statement				
4.	File Access				
5.	Dynamic memory Allocation				
6.	Number of Input Variable				
7.	Number of Output Variable				
8.	Assignment Statement				

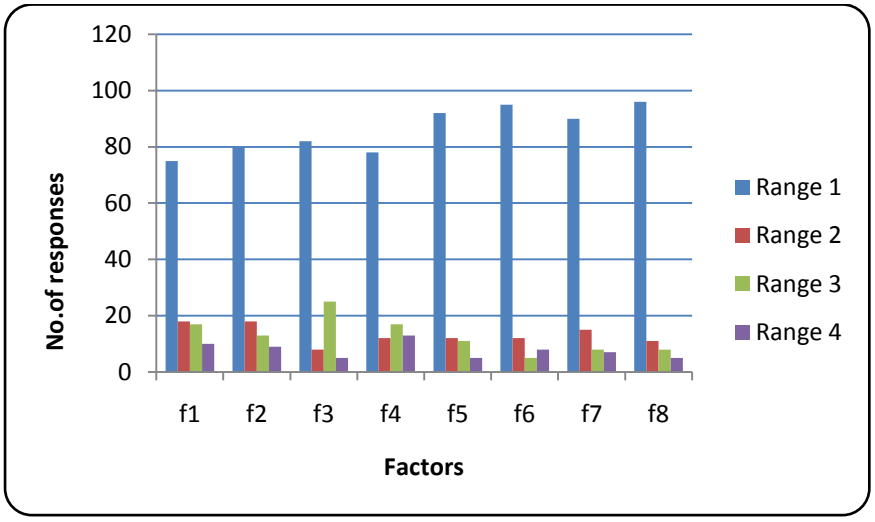
The information regarding Name, Designation and Organization is Optional.

Name : _____

Designation: _____

Organization: _____

The graph represents the result analysis of the survey conducted.



BRIEF PROFILE OF RESEARCH SCHOLAR

Harish Kumar is perusing his Ph.D. in Computer Engineering from YMCA University of Science & Technology, Faridabad. He has done his M.Tech. (CE) from M.D.U. Rohtak in year 2006, B.Tech. (CE) from M.D.U. Rohtak in the year 2004. He has 12 years experience in teaching various computer subjects. Presently he is working as Assistant Professor in Department of Computer Engineering in YMCA University of Science & Technology, Faridabad. His interests include Computer Programming, Software Project Management, Software Engineering, IT Management and Software Testing. He has published 14 research papers in various, International Journals and International Conferences.

REFERENCES

[1] Aditya P. Mathur, "Foundation of Software Testing, Pearson Education," 2nd Edition, 2008.

[2] Prabu, M., Narasimhan, D., Raghuram, S., "An Effective Tool for Optimizing the Number of Test Paths in Data Flow Testing for Anomaly Detection," Computational Intelligence, Cyber Security and Computational Models, *In proceedings of ICC3* 2015, pp. 505- 518, Singapore.

[3] Akira K.Onoma, Wei-Tek Tsai, Mustafa Poonawal and Hiroshi Suganuma, "Regression Testing in an Industrial Environment," Communication *of the ACM*, Vol. 41, No. 5, 1998, pp. 81-86.

[4] Alexy G. Malishevsky, Gregg Rothermal and Sebastian Elbum, "Modelling the cost-benefits tradeoffs for regression testing techniques," in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society, Washington, DC., USA. , 2002, pp. 204-214.

[5] Cagatay Catal and Deepti Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal,* Vol. 21, No. 3, September 2013, pp.445-478.

[6] Andrea Hermann, Maya Daneva, "Requirement Prioritization based on Benefit and Cost Prediction: An Agenda of Future Research," In *16th IEEE International Requirement Engineering Conference*, 2008.

[7] Anna Börjesson, Lena Holmberg, Helena Holmström, Agneta Nilsson, "Use of Appreciative Inquiry in Successful Process Improvement," In *Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda,* Vol. 235 Series IFIP International Federation for Information Processing, 2007, pp. 181-196.

[8] Antonio Mauricio et. al., "A systematic Review of Software Requirements Selection and Prioritization Using SBSE Approaches," *Search Based Software Engineering: Lecture Notes in Computer Science*, Vol. 8084, 2014, pp. 188-208.

[9] K.K. Aggrawal, Yogesh Singh, and A. Kaur, "Code coverage based technique for prioritizing test cases for regression testing," *SIGSOFT Software Engg. Notes,* Vol. 29, No. 5, September 2004, pp. 1-4.

[10] Muthusamy Boopati,Ramlingam Sujata,Chandran Senthil Kr & Srinivasan Narasimman, "Quantification of S/w Code Coverage Using Artificial Bee Colony Optimization based on Markov Approach", *Arabian Journal of Science and Engg.(Springer).*, vol. 42, Issue no. 8, 2017, pp. 3503-3519.

[11] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Comput. Surv.* Vol. 50, No. 2, Article 29 May 2017, pp. 1-32.

[12] Avesani, P, Bazzanella, C, Perini, A & Susi, A 2005, 'Facing scalability issues in requirements prioritization with machine learning techniques', *In Proceedings of the 13th IEEE International Conference on Requirements Engineering*, 2005 pp. 297-305.

[13] B. Korel, J.Laski, "Algorithmic software fault localization," *In Annual Hawaii International Conference on System Sciences*, pp. 246-252, 1991.

[14] Berander, Patrik , Andrews, Anneliese, "Requirements Prioritization," *Engineering and Managing Software Requirements*, 2005, pp. 69–94.

[15] Bertolino and E. Marchetti, "Software testing," (chapt.5).In *P. Bourque and R. Dupuis*: editors, *Guide to Software Engineering, Body of Knowledge*, IEEE Computer Society, 2004.

[16] C. Simons, E.C. Paraiso, "Regression test cases prioritization using Failure Pursuit Sampling," In *10th International Conference on Intelligent Systems Design and Applications (ISDA)*, IEEE, 2010,pp. 923 - 928 .

[17] Camila Loiola Brito Maia, Thiago do Nascimento Ferreira, Fabrício Gomes de, "An evolutionary approach to software test allocation*," Computational Intelligence and Information Technology*, Vol. 250, No. 1, 2013,pp. 637-641.

[18] Changyu Dong, NarankerDulay, "Shinren: Non-monotonic Trust Management for Distributed Systems," *Trust Management IV*, Vol. 321 of the *series IFIP Advances in Information and Communication Technology*, pp 125-140.

[19] Claudio Bartolini, Cesare Stefanelli, Mauro Tortonesi, "SYMIAN: A Simulation Tool for the Optimization of the IT Incident Management Process," *Managing Large-Scale Service Deployment*, Vol. 5273, 2009,pp. 83-94.

[20] Claudio Bartolini, Mathias Sallé, "Business Driven Prioritization of Service Incidents," *Utility Computing*, Vol. 3278, 2004,pp. 64-75.

[21] Cristopz Malz & Peter Gohner, "Agent Based Test Case Prioritization," in *Fourth International Conference on Software Testing, Verification & Validation Workshops*, 2011, pp 149-152.

[22] David LB Schwappach, "The equivalence of numbers: The social value of avoiding health decline: An experimental web-based study," *BMC Medical Informatics and Decision Making*, 2002, pp 1-12.

[23] Dennis Jeffrey, Neelam Gupta, "Test Case Prioritization using Relevant Slices," *in Proceedings of 30th Annual International Computer software and applications conference (COMPSAC '06)*, Vol. 1, 2006, pp. 411-420.

[24] Django Armstrong et. al., "Contextualization: dynamic configuration of virtual machines," *Journal of Cloud Computing*, First Online, 2015, pp 1-15.

[25] Duggal, G & Suri, B, "Understanding regression testing techniques," In Proceedings of the 2nd National Conference on Challenges and Opportunities', COIT,2008.

[26] Dominique Mirandolle, Inge van de Weerd, SjaakBrinkkemper, "Incremental Method Engineering for Process Improvement - A Case Study," Engineering Methods in the Service-Oriented Context, Volume 351 of the series IFIP Advances in Information and Communication Technology,2011, pp 4-18.

[27] Emmanuele Zambon, SandroEtalle, Roel J. Wieringa, Pieter Hartel, "Model-based qualitative risk assessment for availability of IT infrastructures," *Software & Systems Modelling*, Vol. 10, No. 4, 2011, pp. 553-580.

[28] Engstro m., E. Runeson, "Improving Regression Testing transparency and Efficiency with History based Prioritization," An Industrial Case Study Software testing Verification and Validation, in *IEEE fourth International Conference*, IEEE, 2011, pp376-379.

[29] Nancy E. Parks, "Testing & quantifying ERP usability," In *Proceedings of the 1st Annual conference on Research in information technology* (RIIT '12), ACM, New York, NY, USA, 2012, pp. 31-36.

[30] F.I. Vokolos, P.G. Frankl, "Pythia a regression test selection tool based on textual differencing," in *3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, IFIP TC5 WG5.4, Chapman & Hall, 1997, pp. 3–21.

[31] Freitas, Jerffeson Teixeira de Souza, "An Ant Colony optimization approach to the software testing with dependent requirements," *International Symposium Search based Software Engg*, 2011,pp 142-157.

[32] G. Rothermel, et. al., "Prioritizing Test Cases for Regression Testing," IEEE Trans. Software Eng., vol. 27, no. 10, 2001, pp. 929-948.

[33] G.J. Myers, "The Art of Software Testing," John Wiley & Sons, 1979.

[34] Goldberg, E., "Genetic Algorithms in Search, Optimization and Machine learning," Pearson Publication, 1989.

[35] H. Agarwal, J.R. Horgan, E.W. Krauser, S. London, "Incremental Regression Testing," *In IEEE International Conference on Software Maintenance,* 1993, pp. 348-357.

[36] H. Srikanth, L. Williams, J. Osborne, "Towards the Prioritization of system test cases," *Software testing Verification and reliability,* Vol. 24, Issue 4,2014, pp 320-337.

[37] H.Lenng and L.White, "Insights into regression Testing," *In Proceedings of the International Conference on Software Maintenance,* 1989, pp. 60-69.

[38] Hadi Hemmati et. al., "Reducing the cost of Model- Based Testing through Test case Diversity," *Testing Software and System,* Vol. 6435, 2010, pp. 63-78.

[39] H. Do, S. M. Mirarab, L. Tahvildari, and G. Rothermel, "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*,2008, pp.71–82.

[40] Huang, Y. C. ; Chin-Yu Huang; Jun-Ru Chang; Tsan-Yuan Chen, "Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History," *In Computer Software and Applications Conference (COMPSAC),* 2010 IEEE 34th Annual ,2010, pp.413-418.

[41] Hans Heerkens, "Designing and Accessing a Course on Prioritization and Importance Assessment in Strategic non routine Requirements in Engineering Processes," Requirements Engineering, 2014, First Online, pp. 1-16.

[42] Hartmann, J. and Robson, D.J. , "Approaches to regression testing," *In Proceedings of the Conference on Software Maintenance - 1988* (IEEE Cat. No.88CH2615-3). IEEE Computer Society Press, 368-72, 1988.

[43] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," In *ICSE-16. 16th International Conference on Software Engineering* (Cat. No.94CH3409-0). IEEE Computer Society Press, pp. 191-200, 1994.

[44] Harish Kumar & Naresh Chauhan, "A Coupling effect based test case prioritization technique," In *Computing for sustainable global development,2nd international conference on,*2015,pp 1341-1345.

[45] Harish Kumar & Naresh Chauhan, "A Hierarchical System Test Case Prioritization Technique based on Requirements," In *13th Annual International Software Testing Conference*, 04-05 December 2013, Bangalore India, 2013.

[46] Harish Kumar & Naresh Chauhan, "A Module Coupling Slice Based Test case Prioritization Technique", *International Journal of Modern Education and Computer science(IJMECS),* Vol. 7 , No. 7, 2015, pp 8-16.

[47] Harish Kumar & Naresh Chauhan, "A Regression Testing Technique Using Du-Dc Paths," *YMCAUST IJR (YMCAUST International Journal of Research),* Vol. 3, No. 1, 2015, pp 99-104.

[48] Harish Kumar & Naresh Chauhan, "A Unit – Test Case Prioritization Technique Based on Source Code Analysis," *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol. 5, No. 4, 2015.

[49] Harish Kumar & Naresh Chauhan, "HSTCP: A Tool for Hierarchical System Test Case Prioritization," International Journal of Knowledge Based Computer Systems, Vol. 3, No. 1, 2015, pp 8-12.

[50] Harish Kumar & Naresh Chauhan, "Identifying and analyzing the research challenges in Test case prioritization" International Journal of Computer Science & Engineering System, Vol. 6, No. 3, 2012, pp. 88-98.

[51] Harish Kumar & Naresh Chauhan, "Test Case Prioritization Technique using Aggregate Weight of the independent path," Journal of Computer Science and Software Engineering, Vol. 1, No. 1, 2015, pp. 8-16.

[52] Harish Kumar & Naresh Chauhan, "A novel approach to test case prioritization for regression testing", *in International Federation of Information processing(IFIP) and South-East Asia regional computer*

confederation(SEARCC), organised by CSI, 2015,BVICAM Delhi. (Proceedings to be published by Springer).

[53] Hinton, G. E. and Sejnowski, T., "Optimal perceptual inference," *In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1983, pp. 448-453.

[54] Hirsh, H. , "Explanation-based generalization in a logic programming environment," *In Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (IJCAI-87), Milan, Italy. Morgan Kaufmann, 1987, pp221-227.

[55] S. Biswas, M. S. Kaiser and S. A. Mamun, "Applying Ant Colony Optimization in software testing to generate prioritized optimal path and test data," *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*,Dhaka,2015,pp.1-6.

[56] Aitor Arrieta,Shuai Wang,Goiuria Sagardui and D. Marijan, A. Gotlieb, S. Sen, "Test case prioritization for continuous regression testing: An industrial case study", *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, pp. 540-543, 2013.

[57] Huaizhong Li, C. Peng Lam, "Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams," In International Conference on Testing of Communicating Systems, 2005,pp 69-80.

[58] IEEE Standard 610 (1990) definition of test cases [online].

[59] Joao Felipe et. al., " Revealing influence of model structure and test profile on the prioritization of test cases in the context of model based testing," Journal of Software Engineering Research and Development, Online First, 2015, pp. 1-28.

[60] Jedlitschka, A.and Pfahl, D., "Reporting Guidelines for Controlled Experiments in Software Engineering," *In Proceedings of ACM/ IEEE International Symposium on Empirical Software Engineering*, 2005,pp 95-104.

[61] Juristo, N., Moreno, A.M., Vegas, S., and Solari, M., "In search of what we experimentally know about unit testing [software testing]," *IEEE Software*, Vol. 23, No. 6, 2006,pp72-80.

[62] Jos. J. M. Trienekens et. al., “Quality specifications and metrication results from a case study in a mission critical software domain,” Software Quality Journal, 2010, Vol. 18, No. 4, pp. 459- 490.

[63] Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria, “Test Case Prioritization of Configurable Cyber-Physical Systems with Weight-Based Search Algorithms,” In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (GECCO '16), Tobias Friedrich (Ed.). ACM, New York, NY, USA, pp. 1053-1060.

[64] Kampenes Vigdis, B., Dybå, T., Hannay Jo, E., and Sjöberg Dag, I.K., “A systematic review of effect size in software engineering experiments,” *Information and Software Technology* Vol. 49, No. 11, 2007,pp. 1073-1073.

[65] Kakali Chatterjee, et. al., “A Framework for the development of secure software,” CSI Transactions on ICT, Vol. 1, No. 2, 2013, pp. 143-157.

[66] Burnstein, A. Homyen, R. Grom and C.R. Carlson, “A Model to Assess Testing Process Maturity,” CROSSTALK 1998, Software Technology Support Center, Hill Air Force Base, Utah.

[67] Kim, J.-M., Porter, A., and Rothermel, G., “An empirical study of regression test application frequency. Software Testing, Verification and Reliability,” Vol. 15, No. 4, 2005,pp. 257-279.

[68] Kim, J.M., A.Porter, “A history based test case prioritization technique for regression testing in resource constrained environment,” In Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 119-129.

[69] Kitchenham, B.A et. al. , “Systematic literature reviews in software engineering .A tertiary study,” *Information & Software Technology .INFSOF* , Vol. 52, No. 8, 2010,pp. 792-805.

[70] Leung, H.K.N. and L. White, “A cost model to compare regression test strategies” In Proceedings Conference on software maintenance, IEEE Computer Society Press, 1991, pp. 201-208.

[71] Louise Tamres, Introducing Software Testing, Pearson Education, 1st Edition, 2002.

[72] M. Hutchins, H. Foster, T. Goradia and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria," Proc. 16th International Conference of. Software Engg., IEEE, 1994,pp. 191-200.

[73] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse., "Test case prioritization for regression testing of service-oriented business applications," In *Proceedings of the 18th international conference on World wide web* (WWW '09). ACM, New York, NY, USA, 2009, pp. 901-910.

[74] Ma Z., Zhao J., "Test Case Prioritization Based on Analysis of Program Structure," *In 15th Asia-Pacific Software Engineering Conference,* APSEC '08, IEEE, 2008,pp. 471 – 478.

[75] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei, "A Unified Test Case Prioritization Approach," *ACM Trans. Software Engg. Methodol,* Vol. 24, No. 2, Article 10, December 2014, pp. 1-31.

[76] Matthew J.Rummel, Gregory M.Kapfhammer and Andrew Thall, "Towards the prioritization of regression test suites with data flow information," in Proceedings of the 2005 ACM symposium on Applied Computing New York, NY, USA, 2005.

[77] Md. Imrul Kayes, "Test Case Prioritization for Regression Testing based on fault dependency," in IEEE 3rd International Conference on Electronics Computer Technology, India, 2011.

[78] Mohammad Hashemian, Kevin Stanley, Nathaniel Osgood, "Leveraging H1N1 infection transmission modelling with proximity sensor micro data," *BMC Medical Informatics and Decision Making*, December 2012.

[79] MuraleedharanNavarikuth, Subramanian Neelakantan, KalpanaSachan, UdayPratap Singh, Rahul Kumar, AntashreeMallick, "A dynamic firewall architecture based on multi-source analysis," *CSI Transactions on ICT,* Vol. 1, No. 4, 2013, pp. 317-329

[80] Naresh Chauhan, Software Testing – Principle and Practice, Oxford University Press, 1st Edition, 2010.

[81] Nilam Kaushik, Mark Moore, "Dynamic Prioritization in Regression Testing," in *Fourth International Conference on Software Testing, Verification and Validation Workshops,* 2011, pp. 135-138.

[82] Pankaj Jalote, "An Integrated Approach to Software Engineering," Narosa Publishing House, Second Edition, 2003.

[83] S. Elbaum, A. G. Malishevsky and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, 2002, pp. 159-182.

[84] P. Hallman, "Prioritization with Precedence," *In 3rd International Symposium on Search Based Software Engineering,* Hungary, 2011.

[85] Qingfeng, D. "An improved algorithm for basis path testing," In Bussiness management and electronic information, IEEE, 2011, pp. 175-178.

[86] Qu, B., C.Nie, B.Xu and X.Zhang, "Test Case prioritization for black box testing," in *Proceedings of 31st Annual Intentional Computer Software Application Conference*, 2007, pp. 465-474.

[87] Panda, Namita, Acharya, Arup Abhinna, Bhuyan, Prachet, Mohapatra, Durga Prasad, "Test Case Prioritization Using UML State Chart Diagram and End-User Priority," *Computational Intelligence in Data Mining: Proceedings of the International Conference on CIDM,* 10-11 December 2016, pp. 573-580.

[88] Hema Srikanth, Mikaela Cashman, and Myra B. Cohen, "Test case prioritization of build acceptance tests for an enterprise cloud application," *Journal of System Software,* Vol. 119, September 2016, pp. 122-135.

[89] Rothermel, G., Elbaum, S., Malishevsky, A.G., Kallakuri, P., and Xuemei, Q. "On test suite composition and cost-effective regression testing". ACM Transactions on Software Engineering and Methodology, Vol. 13, No. 3, 2004, pp 227-331.

[90] R. W. Kristen, "Prioritizing Regression Test Suites for Time-Constrained Execution Using a Genetic Algorithm," Department of Computer Science, Allegheny College, 2005.

[91] K. K. Aggrawal, Yogesh Singh, and A. Kaur, "Code coverage based technique for prioritizing test cases for regression testing," *SIGSOFT Software Engg. Notes,* Vol. 29, No. 5, September 2004, pp. 1-4.

[92] R. Kavitha, Dr. N. Suresh Kumar "Factors oriented test case prioritization technique in regression testing," *European Journal of Scientific Research,* Vol.55, No. 2, 2011, pp. 261-274.

[93] Rothermal, G. Elbaum, S., "Putting your best tests forward," *IEEE Software*, Vol. 20 No. 5, 2003, pp. 74-77.

[94] Rothermel and M.J. Harrold, "A Framework for evaluating regression test selection techniques," In *Proceedings of 16th International Conference on Software Engineering,* 1994.

[95] Rothermel and M.J. Harrold, “Analyzing regression test selection techniques,” *IEEE Transactions on Software Engineering*, Vol. 22 ,No. 8 ,1996, pp. 529-551.

[96] Rothermel and M.J. Harrold, “Empirical studies of a safe regression test selection technique,” *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, 1998, pp. 401-419.

[97] D. Marijan, A. Gotlieb, S. Sen, "Test case prioritization for continuous regression testing: An industrial case study", *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 540-543.

[98] Rummel J. M. et. al., “Towards the prioritization of regression test suites with data flow information,” *In Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 1499-1504.

[99] S. Mirarab, L. Tahvildari, An Empirical Study on Bayesian Network-based Approach for Test Case Prioritization, *1st International Conference on Software Testing, Verification, and Validation*, 2008,pp. 278 – 287.

[100] Sanjukta Mohanty, Arup Abhinna Acharya, Durga Prasad Mohapatra, “A Model Based Prioritization Technique for Component Based Software Retesting Using UML State Chart Diagram,” In *3rd International Conference on Electronics Computer Technology*, IEEE, 2011.

[101] X. Wang and H. Zeng, "History-Based Dynamic Test Case Prioritization for Requirement Properties in Regression Testing," *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, Austin, TX, 2016, pp. 41-47.

[102] Sherriff, M., et. al. ,”Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records,” *In The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, 2007,pp. 81 – 90.

[103] Sanjeev, A.S.M. and Wibowo, B., “Regression test selection based on version changes of components” *In Tenth Asia-Pacific Software Engineering Conference*, IEEE Computer Society, 2003,pp. 78-85.

[104] Siavash Mirarab, LadanTahvildari, “A Prioritization Approach for Software Test Cases Based on Bayesian Networks,” In International conference *Fundamental Approaches to Software Engineering*, Vol. 4422, 2007, pp. 276-290.

[105] Simone Barbagallo et. al., “Optimization and Planning of operating theatre activities: an original definition of pathways and process modelling,” *BMC Medical Informatics and Decision Making*, 2015, pp. 1-16.

[106] Simone Barbagallo, Luca Corradi, Jean de Ville de Goyet, Marina Iannucci, Ivan Porro, Nicola Rosso , Elena Tanfani, Angela Testi, “Optimization and planning of operating theatre activities: an original definition of pathways and process modelling,” *BMC Medical Informatics and Decision Making*, 2015.

[107] Siripong R., Jirapun D., “Test Case Prioritization Techniques,” *Journal of theoretical and applied information technology*, Vol. 18, No.2,2010,pp. 45-60.

[108] Siripong Roongruangsuwan and Jirapun Daengdej, “A Test Case Prioritization Method with Practical Weight Factors,” *Journal of Software Engineering*, Vol. 4, No. 3, 2010, pp. 193 – 214.

[109] Avinash Gupta, Anshu Gupta and Dharmender Kushwaha, "Test Case Reduction using Decision Table for Requirements Specifications", In proceedings of the International Congress on Informatics and Communication Technology, June 2016, pp. 411-418.

[110] Y. Bian, Z. Li, R. Zhao and D. Gong, "Epistasis Based ACO for Regression Test Case Prioritization," *IEEE Transactions on Emerging Topics in Computational Intelligence*, Vol. 1, No. 3, 2017, pp. 213-223.

[111] T. Gyimothy, A. Beszedes, I. Forgacs, “An efficient relevant slicing method for debugging,” *ACM SIGSOFT Foundations of Software Engineering*, 1999, pp. 303-321.

[112] T. McCabe, “A Complexity Measure,” *IEEE Trans. On Software Engineering*, Vol.2, No.4, 1976, pp. 308-320.

[113] Thangavel Prem Jacob & Thavasi Anandam Ravi, “A novel approach for test suite prioritization,” *Journal of Computer Science*, Vol. 10, No. 1, 2014, pp.138-142.

[114] Thillaikarasi Muthusamy, Seetharaman.K, "Effectiveness of Test Case Prioritization techniques based on Regression Testing," *International Journal of Software Engineering and Applications (IJSEA)*, Vol. 5,No.6,2014, pp. 113--123.

[115] Toshihiko, K., Shingo, T., and Norihisa, D., “Regression test selection based on intermediate code for virtual machines,” *In Proceedings*

International Conference on Software Maintenance ICSM IEEE Comput. Soc, Vol. 420, No. 9, 2003.

[116] Y. Zhang, D. Towey, T. Y. Chen, Z. Zheng and K. Y. Cai, "A random and coverage-based approach for fault localization prioritization," *2016 Chinese Control and Decision Conference(CCDC),*Yinchuan,2016,pp.3354-3361

[117] Wesley K. G. Assuncao et. al., "A Mapping Study of Brazilian SBSE community," *Journal of Software Engineering and Development*, Online First, 2014, pp. 1-16.

[118] Wesley KG Assunção, Márcio de O Barros, Thelma E Colanzi, Arilo C Dias-Neto, Matheus HE Paixão, Jerffeson T de Souza, Silvia R Vergilio, "A mapping study of the Brazilian SBSE community," *Journal of Software Engineering Research and Development*, Vol. 2,No. 3,2014.

[119] Yogesh Kumar, Arvinder Kaur & Bharti Suri, "Empirical Validation of variable based Test Case Prioritization/Selection Techniques," *International Journal of Digital Content Technology and its applications*, Vol. 3, No. 3, 2009.

[120] Yoo S., Harman M., "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification & Reliability*, John Wiley and Sons Ltd ,Vol. 22, No. 2, 2012, pp. 67-120.

[121] Zhang Zhonglin, M.L., "An improved Method of acquiring basis path for software testing," *in ICCSE'* , 2010, pp.1891- 1894.

[122] Zultner R., "Quality Function Deployment for Software: Satisfying Customers," *American Programmer*, 1992, pp. 28-41.

[123] Reeta Sahoo, C Projects, Khanna Book Publishing,4th Edition,2013.

[124] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, "Prioritizing Test cases For Regression Testing". IEEE Transactions on Software Engineering, vol. 27, no. 10, October, 2001, pp 929-948.

[125] Kamna Solanki & Yudhvir Singh, "Novel Classification of Test Case Prioritization Techniques", International Journal of Computer Applications (0975 – 8887) Volume 100– No.12, August 2014, pp 36-42.

[126] Z.Q. Zhou, "Using Coverage Information to Guide Test case Selection in Adaptive Random Testing", in Proceedings of the 34th Annual IEEE Computer Software and Applications Conference Workshops (COMPSAC 2010). IEEE Computer Society, 2010, pp. 208-212.

LIST OF PUBLICATIONS OUT OF THESIS

List of Published Papers in International Journals

S. No	Title of the paper	Name of the Journal where Published	No.	Volume & Issue	Year	Pages
1.	Identifying and analyzing the research challenges in Test case prioritization.	International Journal of Computer Science & Engineering System	0974-4406	Volume 6, Issue 3.	2012	88-98
2.	A Regression Testing Technique Using Du-Dc Paths.	YMCA UST International Journal of Research	2319-9377	Volume3, Issue 1.	2015	99-104
3.	HSTCP: A Tool for Hierarchical System Test Case Prioritization	International Journal of Knowledge Based Computer Systems	2321-5623	Volume 3, Issue 1	2015	8-12
4.	A Unit – Test Case Prioritization Technique Based on Source Code Analysis	International Journal of Advanced Research in Computer Science and Software Engineering	2277-128X	Volume 5, Issue 4	2015	405-410
5.	A Module Coupling Slice Based Test case Prioritization Technique	International Journal of Modern Education and Computer Science (IJMECS),	2075-017X	Volume 7, Issue 7	2015	8-16

List of Communicated Papers in International Journals

S. No	Title of the paper	Name of the Journal where Published	No.	Volume & Issue	Year	Pages
1.	A Hierarchical System Test Case Prioritization (HSTCP) Technique based on Requirements.	International Journal of software engineering, Technology and Applications(IJSETA)	2053-2474			
2.	A Control Structure Weighted Test Case Prioritization Technique.	International Journal of Information Technology (BJIT).	2511-2112			

LIST OF RESEARCH PAPERS

List of Published Papers

INTERNATIONAL CONFERENCES

1. Harish Kumar, Vedpal & Naresh Chauhan, "A Hierarchical System Test Case Prioritization Technique based on Requirements", Published in 13th Annual International Software Testing Conference in India, 04 – 05, December 2013, Bangalore, India.
2. Harish Kumar & Naresh Chauhan, "A Coupling effect based test case prioritization technique," Computing for sustainable global development, 2nd international conference on,2015,pp 1341-1345.
3. Harish Kumar & Naresh Chauhan, "A novel approach to test case prioritization for regression testing", in International Federation of Information processing(IFIP) and South-East Asia regional computer confederation(SEARCC), organised by CSI, 2015,BVICAM Delhi. (Proceedings to be published by Springer).

INTERNATIONAL JOURNALS

1. Harish Kumar & Naresh Chauhan, "Identifying and analyzing the research challenges in Test case prioritization", in International Journal of Computer Science & Engineering System, Vol. 6, No. 3, 2012, pp. 88-98.
2. Harish Kumar & Naresh Chauhan, "A Regression Testing Technique Using Du-Dc Paths" in YMCAUST IJR (YMCAUST International Journal of Research) Vol.3 Issue I. Jan, 2015 pp 99-104 ISSN: 2319-9377.
3. Harish Kumar & Naresh Chauhan, "A Unit – Test Case Prioritization Technique Based on Source Code Analysis" in International Journal of Advanced Research in Computer Science and Software Engineering, Volume 5 Issue 4 April 2015 ISSN: 2277 128X.
4. Harish Kumar & Naresh Chauhan, "HSTCP: A Tool for Hierarchical System Test Case Prioritization", in International Journal of Knowledge Based Computer Systems, Vol. 3 Issue 1, 2015, pp 8-12.

5. Harish Kumar & Naresh Chauhan, "A Module Coupling Slice Based Test case Prioritization Technique", in International Journal of Modern Education and Computer Science(IJMECS) Vol. 7 ,No. 7, July,2015,pp 8-16. ISSN: 2075-0161 (Print), ISSN: 2075-017X (Online).

NATIONAL JOURNAL

1. Harish Kumar & Naresh Chauhan, "Test Case Prioritization Using Aggregate Weight of the Independent Path", Published in Journal of Computer Science Engineering and Software Testing,Volume 1 Issue 1, 2015.

List of Communicated Papers

1. Harish Kumar, Vedpal & Naresh Chauhan, "A Hierarchical System Test Case Prioritization (HSTCP) Technique based on Requirements, Communicated in International Journal of software engineering, Technology and Applications (IJSETA). Inderscience Publishers.
2. Harish Kumar & Naresh Chauhan "A Control Structure Weighted Test case Prioritization Technique" communicated in International Journal of Information Technology(BJIT), Published by Springer.