Regression Testing of Object-Oriented Software based on Program Slicing

Subhrakanta Panda

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha 769 008, India

Regression Testing of Object-Oriented Software based on Program Slicing

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

Doctor of Philosophy

in

Computer Science and Engineering

by

Subhrakanta Panda

under the guidance of

Dr. Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha 769 008, India

April 2016

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India. `www.nitrkl.ac.in`

Dr. Durga Prasad Mohapatra
Associate Professor

April 22, 2016

Supervisor's Certificate

This is to certify that the thesis entitled **Regression Testing of Object-Oriented Software based on Program Slicing**, submitted by **Subhrakanta Panda, Redg. No. 511CS109**, a Institute Research Scholar, in the *Department of Computer Science and Engineering*, *National Institute of Technology*, *Rourkela*, *India*, for the award of the degree of **Doctor of Philosophy**, is a record of an original research work carried out by him under my supervision and guidance. The thesis fulfills all requirements as per the regulations of this Institute and in my opinion has reached the standard needed for award. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India. www.nitrkl.ac.in

April 22, 2016

Certificate of Examination

Roll Number: 511CS109
Name: Subhrakanta Panda
Title of Dissertation: Regression Testing of Object-Oriented Software based on Program Slicing

We the below signed, after checking the dissertation mentioned above and the official record book(s) of the student, hereby state our approval of the dissertation submitted in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computer Science and Engineering at National Institute of Technology Rourkela. We are satisfied with the volume, quality, correctness, and originality of the work.

Prof. D. P. Mohapatra,
Supervisor

Prof. S. K. Jena,
Member, DSC

Prof. P. M. Khilar,
Member, DSC

Prof. A. K. Sahoo,
Member, DSC

Prof. S. Bhattacharya,
External Examiner

Prof. S. K. Rath,
Chairperson, DSC

Prof. S. K. Rath,
Head of the Department

The idea is never to live for ever,
But, to create something that will.

All that I am, or hope to be, I owe to my beloved Mother.
This thesis is for you Maa.

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India. `www.nitrkl.ac.in`

Declaration of Originality

I, **Subhrakanta Panda, Redg. No. 511CS109** hereby declare that this dissertation entitled **Regression Testing of Object-Oriented Software based on Program Slicing**, represents my original work carried out as a doctoral student of *National Institute of Technology*, *Rourkela*, and to the best of my knowledge, it contains no material previously published or written by another person, nor any material presented for the award of any other degree or diploma of NIT Rourkela or any other institution. Any contribution made to this research by others, with whom I have worked at NIT Rourkela or elsewhere, is explicitly acknowledged in the dissertation. Work of other authors cited in this dissertation have been duly acknowledged under the section *Bibliography*. I have also submitted my original research records to the scrutiny committee for evaluation of my dissertation.

I am fully aware that in case of any non-compliance detected in future, the Senate of NIT Rourkela may withdraw the degree awarded to me on the basis of the present dissertation.

April 22, 2016
NIT Rourkela

Subhrakanta Panda

Acknowledgements

Thank you Almighty for these people who carved the person in me.
First, I would like to thank my supervisor Dr. Durga Prasad Mohapatra for giving me the guidance, encouragement, counsel throughout my research and painstakingly reading my reports. Without his invaluable advice and assistance it would not have been possible for me to complete this thesis.

I take this opportunity to extend my sincere thanks to Prof. S. K. Rath, Head, CSE, Prof. S. K. Jena, Prof. P. M. Khilar, and Prof. A. K. Sahoo for serving on my Doctoral Scrutiny Committee and for providing valuable feedback and insightful comments. I am grateful to Prof. Jens Grabowski, GA University, Goettingen, Germany, for having given me an opportunity to work with him and his team. Their critical comments has helped me in exploring the subtle aspects of my research work.

I gratefully acknowledge the support provided by the National Institute of Technology (NIT), Rourkela. I owe a sense of gratitude to Director, NIT Rourkela for his encouraging speeches that motivates many researchers like me. I am grateful to Prof. B. Majhi and to all the faculty members and staff of the CSE Department for their many helpful comments, encouragement, and sympathetic cooperation.

I wish to thank D. Munjal, a graduated student, for helping me with the programming and debugging. I also thank all my research colleagues and friends, especially Jagannath, Mukesh, Lov, Swatee, and Suman Devi, for their encouragement and moral support. I thank Sangharatna Godboley for standing by me in every ups and downs of this journey. I express my indebtedness to Mitali and Mamata Madam without their financial help I could not have even enrolled in Ph.D. I thank Pragyan for having made that difference in my life.

I am grateful to the blessings of my grand parents. I love you Bapa and Maa for this life. You rightly taught to do the best, and let God take care of the rest. I thank my brother, Soumyakanta, for bestowing blind faith on my capabilities even when I had doubts on my worth. I am indebted to the moral support of my Nua Bou, Nani, and Aru Bhaina along with Lali and Kanha. I thank my little angel Shagun, for her playful oil massages with tiny hands have emboldened my body and spirit to face the challenges of life. I thank Anand for having taken care of me and my family. I thank all those who have ever bestowed upon me their best wishes.

April 22, 2016 **Subhrakanta Panda**
NIT Rourkela

Abstract

As software undergoes evolution through a series of changes, it is necessary to validate these changes through regression testing. Regression testing becomes convenient if we can identify the program parts that are likely to be affected by the changes made to the programs as part of maintenance activity. We propose a change impact analysis mechanism as an application of slicing. A new slicing method is proposed to decompose a Java program into affected packages, classes, methods and statements identified with respect to the modification made in the program. The decomposition is based on the hierarchical characteristic of Java programs. We have proposed a suitable intermediate representation for Java programs that shows all the possible dependences among the program parts. This intermediate representation is used to perform the necessary change impact analysis using our proposed slicing technique and identify the program parts that are possibly affected by the change made to the program. The packages, classes, methods, and statements thus affected are identified by traversing the intermediate graph, first in the forward direction and then in the backward direction.

Based on the change impact analysis results, we propose a regression test selection approach to select a subset of the existing test suite. The proposed approach maps the decomposed slice (comprising of the affected program parts) with the coverage information of the existing test suite to select the appropriate test cases for regression testing. All the selected test cases in the new test suite are better suited for regression testing of the modified program as they execute the affected program parts and thus have a high probability of revealing the associated faults.

The regression test case selection approach promises to reduce the size of regression test suite. However, sometimes the selected test suite can still appear enormous, and strict timing constraints can hinder execution of all the test cases in the reduced test suite. Hence, it is essential to minimize the test suite. In a scenario of constrained time and budget, it is difficult for the testers to know how many minimum test cases to choose and still ensure acceptable software quality. So, we introduce novel approaches to minimize the test suite as an integer linear programming problem with optimal results. Existing research on software metrics have proven cohesion metrics as good indicator of fault-proneness. But, none of these proposed metrics are based on change impact analysis. We propose a change-based cohesion measure to compute the cohesiveness of the affected program parts. These cohesion values form the minimization criteria for minimizing the test suite.

We formulate an integer linear programming model based on the cohesion values to optimize the test suite and get optimal results.

Software testers always face the dilemma of enhancing the possibility of fault detection. Regression test case prioritization promises to detect the faults early in the retesting process. Thus, finding an optimal order of execution of the selected regression test cases will maximize the error detection rates at less time and cost. We propose a novel approach to identify a prioritized order of test cases in a given regression selected test suite that has a high chance of fault exposing capability. It is very likely that some test cases execute some program parts that are more prone to errors and have a greater possibility of detecting more errors early during the testing process. We identify the fault-proneness of the affected program parts by finding their coupling values. We propose to compute a new coupling metric for the affected program parts, named affected change coupling, based on which the test cases are prioritized. Our analysis shows that the test cases executing the affected program parts with high affected change coupling have a higher potential of revealing faults early than other test cases in the test suite.

Testing becomes convenient if we identify the changes that require rigorous retesting instead of laying equal focus to retest all the changes. Thus, next we propose an approach to save the effort and cost of retesting by identifying and quantifying the impact of crosscutting changes on other parts of the program. We propose some metrics in this regard that are useful to the testers to take early decision on what to test more and what to test less.

Keywords: Testing, Regression testing, Test case selection, Test suite minimization, Test case prioritization, Change impact, Program slicing, Intermediate graphs.

Contents

1 Introduction **1**

1.1 Regression Testing . . . 1

1.2 Program Slicing . . . 3

1.3 Motivations of our Research Work . . . 4

1.4 Objectives of our Research Work . . . 8

1.5 Contributions of the Thesis . . . 9

1.6 Organization of the Thesis . . . 11

2 Background **13**

2.1 Software Testing . . . 13

2.1.1 Test, Test Case, and Test suite . . . 16

2.1.2 Execution-Based Software Testing . . . 16

2.1.3 Regression Testing . . . 18

2.1.4 Test Adequacy Criteria . . . 21

2.2 Program Slicing . . . 23

2.2.1 Types of Program Slices . . . 23

2.2.2 Program Representation . . . 28

2.3 Precision and Correctness of a Slice . . . 36

2.4 Applications of Program Slicing . . . 38

2.4.1 Testing . . . 39

2.4.2 Debugging . . . 39

2.4.3 Software Maintenance . . . 40

2.4.4 Change Impact Analysis . . . 40

2.4.5 Software Quality Assurance . . . 41

2.4.6 Functional Cohesion . . . 41

2.4.7 Functional Coupling . . . 42

2.4.8 Other Applications of Program Slicing . . . 42

2.5 Summary 42

3 Review of Related Work 43

3.1 Program Slicing 43

3.1.1 Slicing of Object-Oriented Programs 45

3.1.2 Slicing of Java Programs 47

3.2 Regression Testing 49

3.2.1 Test Case Selection 50

3.2.2 Test Suite Minimization 51

3.2.3 Test Case Prioritization 52

3.3 Change Impact Analysis (CIA) 54

3.4 Summary 56

4 Regression Test Case Selection using Slicing 57

4.1 Background 59

4.2 Hierarchical Regression Test Selection 61

4.2.1 Proposed Intermediate Graph Representation:EOOSDG . . . 62

4.2.2 Removal of Redundant Edges 69

4.2.3 Proposed Hierarchical Decomposition (HD) Slicing Algorithm 71

4.2.4 Proposed Hierarchical Regression Test Case Selection (HRTS) Algorithm 74

4.2.5 Working of the Algorithms 74

4.2.6 Complexity Analysis of HRTS Algorithm 78

4.3 Implementation 79

4.3.1 The sample programs 79

4.3.2 Experimental settings 81

4.3.3 Architectural Model of Regression Test Case Selection 82

4.3.4 Result Analysis 83

4.3.5 Threats to Validity 86

4.4 Comparison with Related Work 87

4.5 Summary 91

5 Regression Test Suite Minimization 93

5.1 Motivating scenario 93

5.2 Proposed Approach for Test Suite Minimization 95

5.2.1 Minimization framework 96

5.2.2 Regression Test Case Selection 96

5.2.3 Affected Slice Graph (ASG) Construction using HD Slicing . 97
5.2.4 Computation of Affected Component Cohesion (ACCo) . . . 98
5.2.5 Modeling test suite minimization as binary ILP problem . . . 106
5.3 Experimental study 108
5.3.1 RQ1: Effectiveness 111
5.3.2 RQ2: Usefulness 112
5.3.3 Threats to validity 113
5.4 Comparison with related work 114
5.5 Summary 115

6 Regression Test Case Prioritization 117
6.1 Motivation 118
6.2 Coupling in Object-Oriented Programs 121
6.2.1 Affected Component Coupling (ACC) 124
6.2.2 Theoretical Validation 126
6.2.3 Framework Criteria 128
6.3 Our Proposed Approach for Regression Test Case Prioritization . . . 130
6.3.1 Construction of ASG 131
6.3.2 Computation of ACC 132
6.3.3 Clustering and Assigning Weights 133
6.3.4 Computation of Test Case Weights and Prioritization of Test Cases 135
6.4 Case Study 136
6.5 Correctness of the Algorithms 140
6.6 Complexity Analysis of the Algorithms 141
6.7 Implementation 142
6.7.1 Experimental Program Structure 143
6.7.2 Mutation Analysis 144
6.7.3 Results 147
6.7.4 Threats to Validity 149
6.8 Comparison with Related Work 150
6.9 Summary 156

7 Identifying and Quantifying the Effect of Changes 157
7.1 Background 158
7.1.1 Change Identification 158
7.1.2 Change impact analysis (CIA) 160

7.2 Proposed Metrics for Describing Program Changes 161
7.2.1 Structural program model 162
7.2.2 Proposed Change Cluster Graph (CCG) 163
7.2.3 Definition of the proposed metrics 165
7.2.4 Metrics Computation 167
7.3 Experimental Studies 170
7.3.1 The sample programs 170
7.3.2 Observations 171
7.4 Comparison with related work 176
7.4.1 Threats to validity 179
7.5 Summary 179

8 Conclusions 181
8.1 Contributions 181
8.1.1 Regression Test Case Selection 181
8.1.2 Regression Test Suite Minimization 182
8.1.3 Regression Test Case Prioritization 183
8.1.4 Identifying and Quantifying the Effect of Changes 184
8.1.5 Implementation 185
8.2 Future Work 185

BIBLIOGRAPHY 187

Dissemination 207

Biodata 208

Index 209

List of Figures

1.1 Outline of the thesis 11

2.1 A hierarchy of software testing. 16
2.2 A software testing process model. 17
2.3 Forward slice of a sample program. 24
2.4 Backward slice of a sample program. 25
2.5 Static slice of a sample program. 26
2.6 Dynamic slice of a sample program. 27
2.7 The CFG of the example program given in Figure 2.6a. 30
2.8 The PDG of the example program given in Figure 2.6a. 32
2.9 An example program consisting of a main program and two procedures. 34
2.10 The SDG of the example program given in Figure 2.9. 35
2.11 (a) An example program, and (b) its CHS 37
2.12 An example program 37

4.1 Model for Hierarchical Slicing. 59
4.2 An example Java program 61
4.3 EOOSDG of the example program in Figure 4.2. 63
4.4 Reduced EOOSDG (rEOOSDG) of the example program in Figure 4.2. 66
4.5 Time based comparison between EOOSDG and rEOOSDG for identifying the affected nodes with respect to some modification (slicing criterion). 69
4.6 Architectural model of the hierarchical regression test selection method 82
4.7 Summary of hierarchical test case selection for node 23 of rEOOSDG in Figure 4.4. 84
4.8 Hierarchical test case selection for different input nodes 84
4.9 Time based comparison between EOOSDG and rEOOSDG of different programs to detect their affected parts 85

4.10 A comparison of the percentage of test cases selected for regression testing. 86

5.1 Framework to minimize change-impact-based selected test-suite. . . 95
5.2 Affected Slice Graph (ASG) of the example Java program given in Figure 4.2. 97
5.3 ACCo computation of nodes of ASG in Figure 5.2. 102
5.4 ILP encoding of the test data given in Table 5.1. 108
5.5 % of fault detected by ST and MT. 108
5.6 Test suite minimization results for all the ten changes made to the program. 110
5.7 Fault detection results of the minimized test suite for all the ten changes made to the program. 111
5.8 Timing results of the minimized test suite for all the ten changes made to the program. 112

6.1 APFD measure for the test case orderings in Table 6.1. 119
6.2 Activities of Test Case Prioritization. 131
6.3 The calculated ACC values of different nodes of the ASG in Figure 5.2 and their weights. 133
6.4 K-Means Clustering of the ACC values of the nodes of ASG. 134
6.5 Mutation analysis of programs. 146
6.6 Average percentage of affected nodes covered by the prioritized test cases using the approach of Panigrahi and Mall. 146
6.7 Average percentage of fault prone affected nodes covered by the prioritized test cases using our approach. 147
6.8 Comparison of APFD values for different programs. 147

7.1 Change Ripple Graph (CRG) of the example Java program given in Figure 4.2. 159
7.2 Comparison between CIA approaches. 161
7.3 Change Cluster Graph (CCG) of the example Java program given in Figure 4.2. 164
7.4 Change impact analysis of the two changes made to the program given in Figure 4.2. 169
7.5 Box-plot of the time taken to compute the slices of the sample programs. 171

7.6 Change ripple analysis of programs. 172
7.7 Box-plot of the percentage of fault mutants present in affected parts of the programs. 172
7.8 Average percentage of affected nodes versus affected test cases. . . . 173
7.9 Crosscutting change analysis. 173
7.10 Box-plot of the percentage of faults detected in the sample programs. 174

List of Tables

4.1 Test case coverage distribution for the example program in Figure 4.2. 73
4.2 Summary of test case selection for the example program in Figure 4.2 74
4.3 Summary of change types in Java programs. 79
4.4 Result obtained for regression testing of different programs. 80
4.5 Comparison of Hierarchical Slicing versus HD slicing. 85

5.1 Test related data for the example program given in Figure 4.2. . . . 94
5.2 Comparison of our proposed change-based cohesion metric with different existing approaches. 109
5.3 Test-suite minimization result of different programs. 110

6.1 A sample test case distribution and the faults detected by them. . . 118
6.2 Comparison with mechanisms that measure coupling. 124
6.3 Test case coverage of fault prone affected nodes. 131
6.4 Impact types of ACC values. 133
6.5 Distribution of test case weights on the basis of fault prone impact. . 136
6.6 Result obtained for regression testing of different programs. 144
6.7 Overview of Mutation Operators 145

7.1 The list of the sample programs used in the study. 170
7.2 Degree of scattering and focus of the sample programs. 174

List of Algorithms

1 *Algorithm RER* 70
2 *HDslice(G, n)* 72
3 *HRTS* 75
4 *findACCo(G_a, n)* 100
5 Forward Traversal 135
6 Backward Traversal 135
7 *findWACC(ASG, n)* 137
8 *H-PTCACC(T, WACC)* 138

Chapter 1

Introduction

The change in the user requirements and growing expectations of the customers have forced the software to evolve at regular intervals of time. As the complexity of software increases, the cost and effort to maintain such complex software also increase. After making the required changes to the software, regression testing should be carried out in order to assure the validity of the modified part and to ensure that the changes do not affect other parts of the program. Therefore, regression testing has become an integral part of the software maintenance process. It is indispensable to make changes to an already tested program. Thus, the role of regression testing has become apparent in retesting the program. The retesting is based on the modifications done without compromising with the time and cost of retesting, while maintaining same testing coverage.

1.1 Regression Testing

In the software development life cycle, regression testing is considered to be an important part. Regression testing is defined as 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 [41]. A system is said to regress if 1) a new component is added, or 2) a modification done to the existing component affects other parts of the program. Therefore, it is essential to retest not only the changed code but also to retest the possible affected code due to the change. Regression testing is an expensive activity and typically accounts for half of the total cost of software maintenance [127]. It is essential to cut-down the cost of retesting the software by following a selective approach to identify and retest only those parts of the program that are affected by the change. Gupta et al. [81] have identified two important problems in selective regression testing: (1) identifying those existing tests that must be rerun since they may exhibit different behavior in the changed program and (2) identifying those program components that must be retested to satisfy some coverage criterion. Thus, the two problems of [81] can be elaborated as a process comprising of the following steps:

i selecting a set of test cases T to be executed on a program P,

ii selecting $T' \subseteq T$ and retesting P' with T' to establish the correctness of P' with respect to T', where P' is the modified version of program P,

iii creating T'', a set of new test cases for P', if required, and retesting P' with T'', so that we still get the same correctness of P' with respect to T'',

iv creating T''' from T, T', T'' and by adding some new test cases, if required, to test the correctness of P'.

All the above mentioned steps cover the following important problems associated with regression testing: regression test selection problem, coverage identification problem, test suit execution problem and test suit maintenance problem. There are four approaches by which the problem of regression testing of a software can be solved [41]. These are: *(i) Retest all approach*, *(ii) Test suite reduction* [57, 210], *(iii) Regression test selection* [90, 188], and *(iv) Test case prioritization* [104, 108, 175].

i. *Retest all approach*: In this approach, all the test cases available in the test suite are executed to test the changed version of the program. Test suite T effectively covers the modified program $P^{'}$.

ii. *Test suite reduction*: Even though all the test cases of a given test suite T can be executed to test a modified program, but the execution cost will be very high. Test case reduction/minimization approach [57, 210] focuses on those test cases that need to be eliminated permanently to reduce the cost of retesting because of the following reasons: i) the test case has become obsolete due to the changes done to the program, ii) there may also be some redundant test cases present in the test suite with respect to the code or exercised functionality.

iii. *Regression test selection*: This approach focuses on reducing the time required to retest a modified program by selecting a subset of the given test suite. Therefore, regression test selection techniques [81, 90, 188] attempt to identify only those test cases that can exercise the modified parts of the program and the parts that are affected by the modification to reduce the cost of testing.

iv. *Test case prioritization*: Test case prioritization focuses on reordering the sequence of execution of test cases [66, 69, 104, 108, 175, 184]. The sequencing of the test cases in a given test suite is done based on some established criteria. The test case having the higher priority is executed earlier than the test case with lower priority. There are two types of prioritization [41]:

   i. *General test case prioritization*: For a given program P and a test suite T, the test cases are prioritized such that the prioritization is useful to a succession of program modifications done to P, without the knowledge of the modification.

   ii. *Version specific test case prioritization*: In this approach, the test cases are prioritized whenever program P is modified to P', with the knowledge of the changes made in P.

1.2 Program Slicing

Program slicing has been proved as an effective and efficient technique for program analysis. The main applications of program slicing includes program debugging, change impact analysis, program comprehension, fault detection, testing, and maintenance in object-oriented software [54, 70, 79, 81, 89, 100, 104, 128, 149, 150, 186, 188, 203, 217]. Change impact analysis and regression testing are integral parts of software maintenance. A program slice at a statement s consists of a set of relevant statements of a program those directly or indirectly affect s. A slice at s can refer to the ripple effect of the change at s. Therefore, program slicing is the most favorable technique to study the effect of change. Thus, program slicing finds an application in ensuring the high integrity of the software after changes are made. The computed slice at the point of change reduces the effort of the tester allowing him to focus attention on the ripple effect of one change at a time. The precise and accurate computation of the slices to discover the affected program parts for examination, restores the confidence of the tester that a relevant section of the code has not been missed. This enables the inspection of the ripple effect in large sample of programs. According to the existing literature, the result of change impact analysis makes the technique of program slicing a suitable option for regression testing [81, 104, 188]. In traditional procedure-oriented programs, the approach for regression testing was based on the data flows and control flows within a procedure or among a group of procedures that were computed by graph reachability algorithms [100, 101, 161]. This was mainly achieved by slicing the program dependence graphs (PDG) using the above graph reachability algorithms to obtain the desired program slices. However, while applying the same techniques to object-oriented programs, it is observed that these techniques fail because of the presence of many other dependences originating from the object-oriented features. Although object-oriented features have improved program understandability and readability but at the same time these features have also complicated the maintenance activities. Besides the control and data dependences, the other dependences that may arise due to the class and object concepts are inheritance dependence, message dependence, data dependence, type dependence, reference dependence, concurrence dependence, etc. So, there is a pressing need to handle these dependences while performing regression testing of object-oriented software.

1.3 Motivations of our Research Work

The existing slicing techniques based on system dependence graphs [119, 123, 154, 216] have considered C++ programs that are *partially* object-oriented in nature. In case of object-oriented programs, the programming complexity shifts from method interaction to object relations and communication among objects. The different dependences present in an object-oriented program need to be considered to find the erroneous parts for better program comprehension as they affect the behavior of other components of the program. In this context, it is essential to make a thorough analysis of the dependences between different programming constructs and to detect the critical parts of the programs. To identify these dependences among the program parts, it is essential to model the program with a suitable graphical representation. That's why we are motivated to consider Java programs for our work that is considered as a *true* object-oriented programming language.

But the existing slicing techniques cannot be applied to Java programs because of the presence of many new features that increase the dependences among the components of a Java program [44, 85, 116, 130, 198, 215]. The presence of the features like *packages*, *super*, *dynamic method dispatch*, *interface*, *exception handling*, *multi-threading*, etc, in Java add to the list of dependences and thus make the maintenance even more difficult. Their effects on the maintenance of the programs need to be considered separately.

Apart from this, there are many methods that depend on the type of data they are operating upon. For each type of data, there is a different function. It is essential for the intermediate graphical representation to exhibit all such dependences for an accurate comprehension of the program. Therefore, use of existing slicing techniques to slice the current *system dependence graph (SDG)* of Java programs, does not seem suitable for regression testing. So, there is a need to have a suitable graphical representation of the Java programs, and a new slicing algorithm that can correctly reflect the ripple effect of the changes.

It is essential to validate the modifications and ensure that no other parts of the program have been affected by the change. Incremental regression testing [2, 207] is a probable solution to validate the changes. Some simple observations related to incremental regression testing are as follows: (1) If a statement is not executed under a test case, it cannot affect the program output for that test case. (2) Not all statements in the program are executed under all test cases. (3) Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case. (4) Every statement does not necessarily affect every part of the program output. We can apply the above assumptions to Java programs at different levels such as packages, classes, methods and statements for an efficient selective regression testing. Instead of exhausting all the test cases to validate every change made to the programs, it is wise to select a subset of the test cases that actually cover the affected program parts. Therefore, an efficient approach for *change-based test case selection* is highly necessary for the testers to build the same confidence as it would have been in case of *retest all* approach.

For large programs, even the selected test suite can be quite large for the testers to execute with all the test cases. The adverse impact of this *retest-all* approach even with selected test suite may result in project deadline misses and may incur huge cost while retesting the system for every change. This requires further minimization of the test suite. Therefore, test suite minimization techniques [91, 134, 146] aim to reduce the redundant and obsolete test cases from the regression test suite such that the coverage achieved after reduction is still the same as the initial test suite. Previous work on test suite minimization [28, 57, 105, 131, 208] aimed at developing heuristics for defining the minimization problem. According to the survey in [210], no single heuristic is better than the other, because the heuristic that selects one test case may become redundant for the another. Thus, finding a heuristic that is more relevant to the change in the program is more essential to save the cost of regression testing. *Cohesion* of the affected components can be computed as an effective indicator of change(fault)-proneness [8, 9, 55, 213], thus making it a suitable heuristic for minimizing the test suite. Therefore, minimization problem with respect to regression testing should aim to find the essential test cases concerning the change.

The empirical studies in [61, 67, 175, 176, 210] suggest that the order of test cases execution plays a vital role in detecting faults early in the testing process. An early feedback on the presence of faults can enable the testers to locate the bugs early. It gives an indication to the testers about the test cases that should be exercised first in case the testing has to be prematurely halted. Thus, test case prioritization [66, 69, 104, 108, 113, 148, 162, 163, 175, 177, 184, 214] finds a schedule for the test cases so that if executed in that sequence, it maximizes its effectiveness in meeting some performance goals. Performance goals are the criteria set by the testers based on their expertise and intuition. For example some performance goals can be to *maximize code coverage*, *branch coverage*, *MCDC* [78], *frequency of features coverage*, etc. One of the popular performance goals is *rate of fault detection*. However, fault detection ability of the test cases cannot be known apriori. Therefore, testers rely on surrogates to overcome the difficulty of knowing the test case that has higher ability to detect faults [67]. The assumption is that early maximization of the surrogate property will enhance the likelihood of fault detection. In many empirical studies [10, 15, 29, 35, 36, 47, 65, 117, 150], *coupling measures* are proven to have strong correlation with fault-proneness. But none of the empirical studies on prioritizing approaches [34, 58, 61, 67, 74, 79, 151, 176, 184, 210, 213] reports the use of the coupling measures to prioritize the test cases.

Though testing is a process carried out to discover as many faults as possible to confirm the quality of the software, but testers are sometimes conditioned to fail. The testers may not have the liberty of exhaustive retesting of every change made to the program in a looming scenario of time and cost (due to project deadline, customer impatience, market pressure, etc.). Therefore, the tester needs to test less without sacrificing the quality [98]. Under such circumstances the tester needs to decide, is it always possible and necessary to validate every change through an equal amount of retesting? The answer to this question is quite intuitive based on Pareto principle that suggests, not all changes will require the same amount of retesting. Thus, the tester has to make a decision about what to test and what not to test, what to test more and what to test less, and also in what order to test. The questions about what to test and what not to test, and in what order to test are answered through regression test case selection and prioritization. But, the answer to what to test more and what to test less can come only through some mechanism that can quantify the effect of change. This require metrics to be proposed and defined with respect to the changes that are made to the program. To the best of our knowledge, no such metrics has been defined or proposed in the existing literature [1, 17, 76, 77, 96, 121, 126, 168, 169, 178, 180, 192] on change impact analysis.

Thus, the motivations behind our research work on program slicing-based change impact analysis and its application to regression testing are summarized below:

- The features of a Java program add more complexity by inducing many more dependences among its program parts [198]. Thus, the existing graph based regression test case selection techniques are not suitable for Java programs. So there is a pressing need to develop suitable techniques for regression testing of Java programs.

- Many of the regression testing techniques are unsafe, imprecise, and computationally expensive [27, 172]. Very few techniques considering Java programs focus on safe regression test case selection [90, 188]. A safe technique selects those test cases that have high probability of revealing faults. Therefore, it is essential to develop a safe regression test case selection technique concerning Java programs.

- The empirical studies [8, 10, 16, 34, 55, 56, 79, 99, 117, 150, 151, 213] prove the correlation of cohesion measure [7, 14, 40, 46, 82, 118, 149, 218, 219] with the fault proneness of the components, making it a suitable candidate for test suite minimization heuristic. So, a new minimization approach based on the changes and its impact using the cohesiveness of the affected components is desirable to give a concrete solution.

- The existing coupling measures [15, 36] proposed for object-oriented programs are not concerned with the change and its impact. Therefore, a change-based coupling measure, if used for prioritizing the test cases, can promise better likelihood of fault detection.

- It is necessary to test less without sacrificing the quality [98]. But, unfortunately no metrics have been suggested that can quantify the effect of a change to help the tester decide what to test more and what to test less. The presence of dependence communities [83] among the changes made to a program can help the testers to save on regression testing time.

1.4 Objectives of our Research Work

Our focus is to reduce the execution of the existing tests so as to achieve comparable rate of fault detection and be confident about the quality of the software. Our objective is to save regression testing time by reducing the cost and time of retesting of the modified as well as affected parts of the program through change-based selection of regression test cases, minimization of these selected test cases, and prioritization of the minimized test cases. We also aim to propose a mechanism that enables a tester to decide the relevant changes in a program that require immediate attention and thus save regression testing time. To realize this broad objective, we identify the following goals based on the motivations outlined in the previous section.

- To develop an efficient algorithm that selects the affected test cases based on the slices of the affected program parts. To address this major objective, we form the following sub-objectives:
  - To construct a suitable intermediate graph to represent the Java programs under test.
  - To develop an efficient slicing algorithm that works on the proposed intermediate graph and correctly identifies the program parts those are affected by the ripple effect of the changes made to the program.
- To develop a heuristic based on the cohesion values of the affected program parts for minimizing the selected change-based test suite. For this we plan to develop:
  - An efficient algorithm for correctly computing the proposed change-based cohesion measure of the affected program parts.
- To develop an approach based on the coupling values of the affected program parts for prioritizing the test cases within the selected/minimized test suite. This requires us:
  - To develop an efficient algorithm for correctly computing the proposed change-based coupling measure of the affected program parts.
- To identify and quantify the impact of change for enabling the tester to save time during regression testing of the modified programs. This requires us to set the following sub-objectives:

- To construct a change cluster graph using the concept of dependence communities for representing the dependences among various changes that are made to a program.
- To propose and define the change impact metrics that can quantify the scattering and tangling of the changes.

1.5 Contributions of the Thesis

Based on our objectives mentioned in the previous section, our research work makes the following contributions:

- We propose a novel regression test case selection approach by decomposing an object-oriented (OO) program into packages, classes, methods and statements that are affected by some modification made to the program. This decomposition is based on slicing of an OO program and is named *hierarchical decomposition (HD) slicing.* We select a subset of the regression test suite to retest the modified program by mapping the decomposed program parts with the coverage of the existing test suite. The HD slicing works on a suitable intermediate graph proposed for representing an OO program. This intermediate graph correctly represents all the possible dependences among the different parts of an OO program. The advantage of HD slicing approach is that it is context sensitive and correctly selects a precise number of affected nodes in less time as compared to the approach in [130, 188].

- Software testers always face the dilemma of whether to retest the software with all the test cases or select a few of them based on their fault detection ability. Regression test case selection promises to detect the faults with few test cases. However, sometimes the selected test suite can still appear enormous for regression testing in strict timing constraints. Hence, it is essential to further minimize the test suite. In constrained time and budget, it is difficult for the testers to know how many minimum test cases to choose and still ensure acceptable software quality. We introduce a novel approach to minimize the test suite as an *integer linear programming (ILP)* problem with optimal results. The minimization method uses the proposed *affected component cohesion (ACCo)* values of the program parts affected by the change made to the program. The hypothesis is that the program parts with low cohesion values are more prone to errors. This assumption is validated with respect to the mutation fault detection ability of the test cases. The experimental results show that the minimized test suite can efficiently reveal the errors and ensure acceptable software quality.

- Test case prioritization focuses on finding a suitable order of execution of the test cases in a test suite to meet some performance goals like detecting faults early. It is likely that some test cases execute the program parts that are more prone to errors and will detect more errors if executed early during the testing process. Finding an optimal order of execution for the selected regression test cases saves time and cost of retesting. We present a static approach for prioritizing the test cases based on the proposed *affected component coupling (ACC)* of the program parts. We determine the fault-proneness of the affected program parts by computing their respective ACC values. We assign higher priority to those test cases that cover the program parts with higher ACC values. Our analysis with mutation faults shows that the test cases executing the fault-prone program parts have a higher chance to reveal faults earlier than other test cases in the test suite.

- As crosscutting concerns degrade the software quality, similarly, the crosscutting changes can cause regression testing difficult. Software undergoes evolution through a series of changes. As a result, it becomes necessary to validate these changes through regression testing. But, is it always possible to validate every change through an equal amount of retesting? The intuition says, not all changes will require the same amount of retesting. The job of the testers become convenient if they can find out those changes that should undergo more rigorous retesting instead of laying equal focus on all the changes. We make a novel contribution to quantify the impact of crosscutting changes to save the effort and cost of retesting. We propose some metrics to quantify the severity of the changes that act as indicators for the amount of regression testing required to validate the change. The results of our experimental study show that our proposed metrics are better able to quantify the changes. These metrics are useful indicators of the fault-proneness and can be used by the testers to make essential estimation of the testing effort.

The relationships among the contributions are shown in Figure 1.1. The details of each contribution is discussed in Section 1.5.

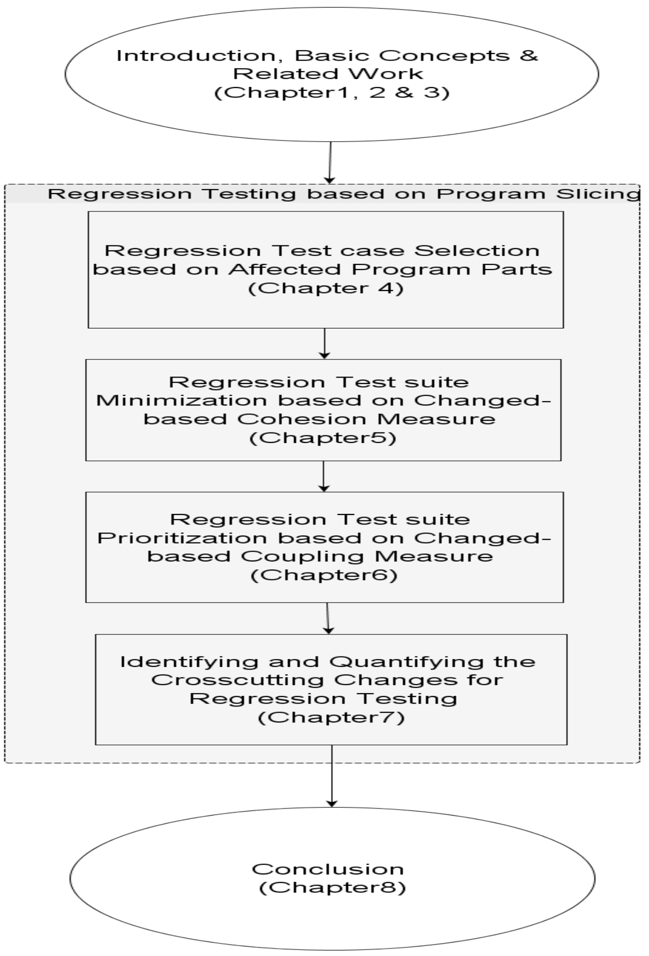

Introduction, Basic Concepts & Related Work (Chapter1, 2 & 3)
Regression Testing based on Program Slicing
Regression Test case Selection based on Affected Program Parts (Chapter 4)
Regression Test suite Minimization based on Changed-based Cohesion Measure (Chapter5)
Regression Test suite Prioritization based on Changed-based Coupling Measure (Chapter6)
Identifying and Quantifying the Crosscutting Changes for Regression Testing (Chapter7)
Conclusion (Chapter8)

Figure 1.1: Outline of the thesis

1.6 Organization of the Thesis

The rest of the thesis is organized into chapters as follows:

- *Chapter 2* talks about the basic concepts and techniques used in the reification of the proposed objectives in the rest of the thesis.
- *Chapter 3* provides a brief review of the existing work that are closely related and relevant to our contributions. We start with a discussion on the evolution of program slicing techniques over the years. This is followed by some earlier contributions related to regression testing especially on test case selection, minimization, and prioritization. Finally, we provide some of the relevant work on change impact analysis and highlight their limitations that motivated us to quantify the effect of changes.

- *Chapter 4* presents our contribution in developing a novel regression test case selection approach for object-oriented software.
- *Chapter 5* proposes an approach to compute the *cohesion measure* of the affected program parts and use the results to *minimize* the change-based selected test suite. This takes the work done in Chapter 4 to one step ahead in regression testing.
- *Chapter 6* proposes an approach to compute the *coupling measure* of the affected program parts and uses the result to *prioritize* the test cases of the test suite minimized in Chapter 5.
- *Chapter 7* presents an approach to *quantify* the effect of the changes made to the program. We propose some metrics in this regard that help in identifying the changes that require more attention of the tester.
- *Chapter 8* concludes the thesis with a summary of our contributions. We also briefly provide some insights into the possible extensions to our work.

Chapter 2

Background

This chapter provides a general idea of the underlying theories on which the rest of the thesis is based upon. For the sake of conciseness and to avoid trivial discussions, we do not provide a detailed and minute description of these underlying theories. We provide a short introduction on the underlying theories to highlight only on those non-trivial concepts and definitions that contribute to the understanding of this thesis.

This chapter is organized as follows: we give a brief description of the artifacts associated with software testing in Section 2.1. Regression testing is an indispensable part of the software testing process. We define the problem of regression testing and discuss various approaches to address this problem in Section 2.1.3. These approaches to regression testing forms our contributory work. We provide a brief introduction to the techniques of program slicing in Section 2.2. The contributions of this thesis are based on an intermediate representation. So, a brief account of the evolution of some of these intermediate representations is given in Section 2.2.2. Since its inception, program slicing has been used in various applications. We discuss some of these usages that relate to our contributions in Section 2.4. Finally, we summarize our discussion on the basic concepts in Section 2.5.

2.1 Software Testing

When a program is developed as an implementation of any algorithm or logic, the developers are always doubtful about its performance and correctness. The developers must have the confidence that the software achieves certain level of quality. Software testing can be appropriately used to ensure the quality of the software to a certain level. A quality software should be correct. A software can only be correct, iff it computes results for the entire domain of input, and all the results it computes are specified. Thus, software requires exhaustive testing to validate the input domain. A software testing approach can only suggest the presence of faults and cannot highlight their absence if it is not exhaustive. According to Karner et al. [107], exhaustive testing of software is not possible for the following reasons:

- the input domain is too large, for example to test the greatest among two numbers, the input domain can be any number n that belongs to the Integer set, I.
- there are too many possible input paths to test, so the difficulties alluded to by this assertion are exacerbated by the fact that certain execution paths in a program could be infeasible [158, 191].
- design and specifications can change during software development and are thus difficult to test, this is because software testing is an algorithmically unsolvable problem and specification errors cause major design errors [41]. According to Manna and Waldinger [144], it is not possible to surely know the correctness of the specifications.

Hamlet et al. [84] have formally stated the goals of a software testing methodology and Morell et al. [155] have highlighted its limitations. Young and Taylor [212] observed that there was always a trade-off between exhaustive testing and computational cost because the presence of defects was always undecidable. Therefore, no testing technique can be completely accurate and generic to all the programs. Even though the testing process is challenged with many limitations, but the consistent application of a testing technique in an intelligent manner can ensure an acceptable level of software quality. Therefore, testing is an important phase in software development life cycle. This phase incurs 60% of the total cost of the software. Therefore, it becomes highly essential to devise proper testing techniques in order to design the test cases so that the software can be tested properly. Testing strategies are based on verification and validation. The static techniques available for testing map to the verification process without executing the code, whereas the dynamic testing techniques map to the validation process by executing the code.

Figure 2.1 shows the hierarchical decomposition of the testing strategies along with their association with different test adequacy criteria. The decomposition shown in Figure 2.1 follows the definitions given in [41, 220]. This thesis follows the *execution-based testing* strategy. The execution-based testing techniques are decomposed into either *program-based*, *specification-based*, or *combined* as shown in Figure 2.1. The chosen adequacy criterion C determines the types of test cases that belong to the test suite, T. A *program-based testing* approach creates T by analyzing the source code of a program, P, based upon its structure and attributes. A *specification-based testing* technique creates the desired test suite from the functional and/or non-functional requirements for P. Whereas, *combined testing* uses both program-based and specification-based testing approaches to generate T. Based on the kind of testing strategy that is followed to create T, the test cases are categorized into three types:

- **Black box**, test cases that are created without knowledge of P's source code and are based only the functional specifications. Thus, these test cases are only based on the input and output behavior and do not depend upon the internal structure of P. The important types of black box testing are equivalence class partitioning and boundary value analysis.

- **White box**, test cases that are created considering the entire source code of P and are based on some heuristics. It is an important approach for unit testing. The different types of white box testing are fault-based testing, coverage based testing, data flow based testing, etc.

- **Grey box**, test cases that are created only by considering the design models of the program P. The different types of grey box testing are state-model-based testing, use case-based testing, class diagram-based testing, and sequence diagram-based testing.

Figure 2.1 also shows the association of these testing strategies with the corresponding test adequacy criteria. A structurally-based criterion requires T to satisfy exercising of certain control structures and variables within P, such as statement coverage, branch coverage, condition coverage, path coverage, etc. Therefore, structurally-based test adequacy criterion requires program-based testing. Fault-based test adequacy criterion ensures that the types of faults that are commonly introduced into P by the programmers are revealed by T. Finally, *error-based testing* approaches rely upon the fact that T does not deviate from the specifications in any way. Therefore, error-based adequacy criteria motivate for specification-based testing approaches.

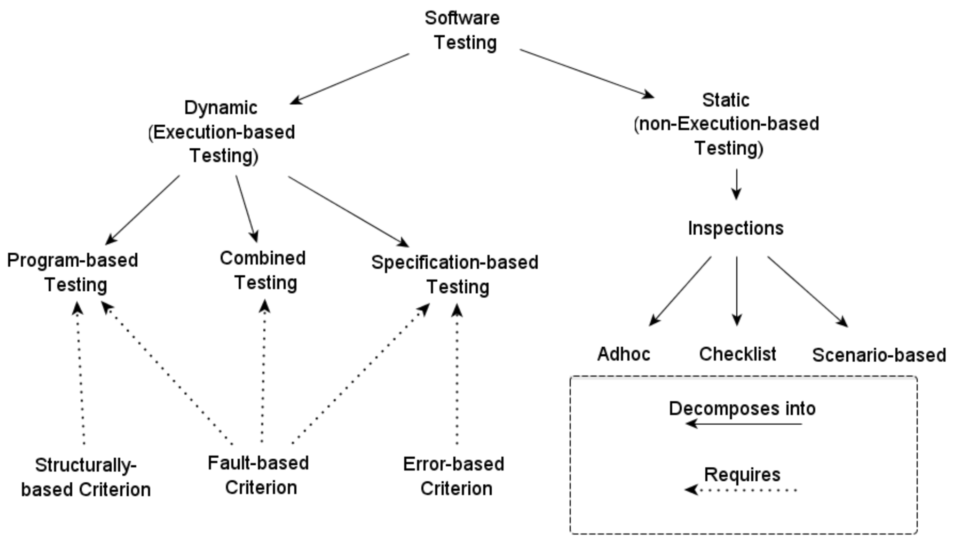


Figure 2.1: A hierarchy of software testing.

2.1.1 Test, Test Case, and Test suite

In the context of software testing, *test* and *test cases* are often used interchangeably. A *test case* consists of an initial state, inputs, and expected outputs. The state refers to pre-conditions, if any, i.e. circumstances that hold prior to the test execution. Each test case is also identified by a unique identification number. The process of testing is to check whether the inputs yield the expected outputs or not. The test case is said to fail if the actual output differs from the expected output. If the test case fails, then it requires debugging to reach the cause of this failure. An efficient test case has a very high probability of revealing the defect. Therefore, it is essential to design test cases based on the identified weak areas of the program. A set of these test cases designated to test an application is called *test suite* . A test suite may be segregated into set of successful and unsuccessful test cases. Any information related to the test cases within a test suite are maintained for future reference.

2.1.2 Execution-Based Software Testing

Figure 2.2 shows the process of *execution-based* software testing. In this figure, the rectangles denote the testing activities and the parallelograms denote the outcome of these activities. The testing process starts with a system under test, P, and a test adequacy criterion, C, as input. The testing process is iterative and stops iterating

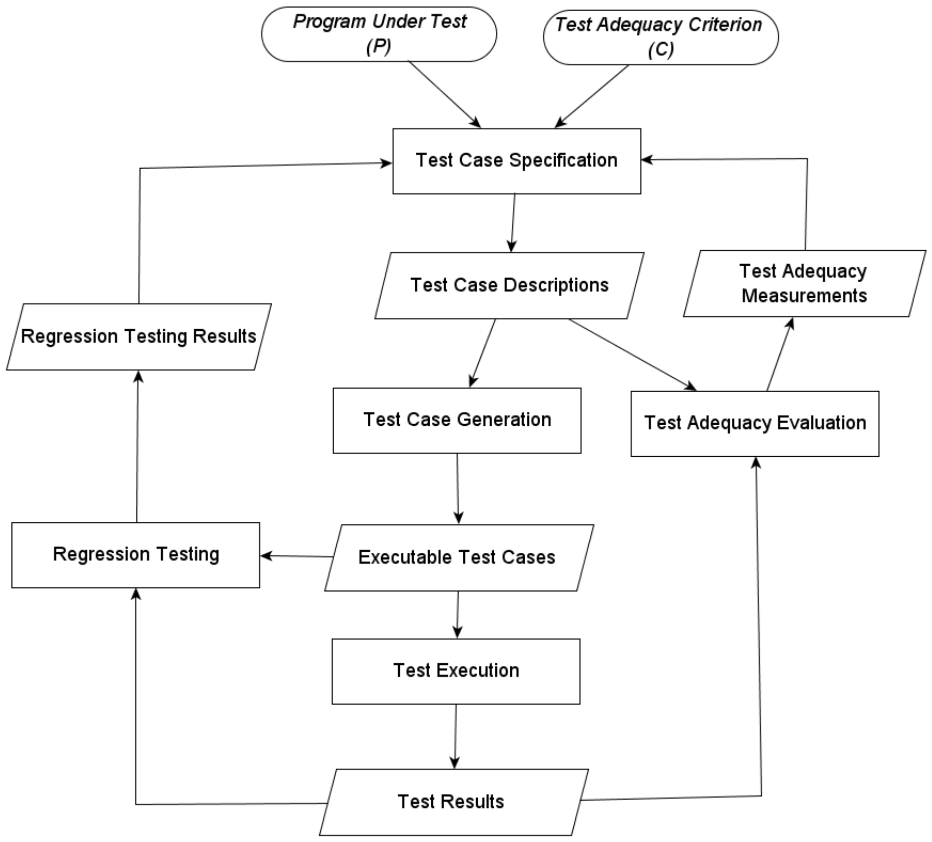


Figure 2.2: A software testing process model.

when the test cases in test suite T satisfies the adequacy criterion C and assures some level of confidence in the quality of P [220]. However, the testing process can also stop in case of deadline misses or budget overrun. The testing process can also halt if the tester gets an intuition of achieving some acceptable level of quality. Even if the testing process stops or meets with C, it does not guarantee that all the defects have been revealed by the test cases. Therefore, testers set many adequacy criteria (different coverage criteria, software metrics, etc.) to build the confidence on quality. The test adequacy criteria depend on the chosen program representation and definition of some quality parameters that T should satisfy. In Section 2.1.4, we discuss the test adequacy criteria considered in this thesis and some test adequacy metrics that exist in the literature and practiced frequently. The test specification stage evaluates P in the context of chosen C in order to construct an adequate T. Once the test case descriptions for P have been generated, the test case generation phase begins. A lot of different techniques and tools have been proposed to manually or automatically generate the test cases. The generation of test cases is beyond the purview of this thesis because of the availability of JUnit test cases available in *Software-artifact Infrastructure Repository (SIR)* [59] for the experimental programs. After the generation of the test cases, the execution of the test cases starts. Once again, the execution of the tests within T can be performed in a manual or automated fashion. The results from the execution of the test cases are analyzed using JaButi framework [196] to determine the quality of individual test cases in terms of coverage. Thus, iterative testing of P continues throughout its initial development. However, it is also important to continue testing after P undergoes changes in maintenance phase. Regression testing is an important software maintenance activity carried out to ensure that the changes made does not adversely affect the correctness of P. All of the previously mentioned stages iteratively continue for the regression testing process based on the existing test cases (new test cases may be added) and the adequacy measurements defined for these tests [2, 41, 107, 127, 173].

2.1.3 Regression Testing

Regression testing is considered as a part of the validation activity and seems to pose a big problem in testing the software. It becomes a big challenge to manage the retesting process with respect to the time and cost, especially when the test suite becomes too large. A software system undergoes changes in the form of bug fixes or addition/deletion of functionality. During the process of maintenance the software needs to be regression tested to validate that these changes introduced no defects. Figure 2.2 shows that the regression testing process has to go through all the testing stages for every change made to the program. Thus, regression testing ensures that the evolution of an application does not inadvertently lowers the software quality. Indeed, the importance of regression testing is well understood. However, many software development teams may not afford thorough regression testing because they often expense one-half the cost of software maintenance [127]. The execution of the test suite often makes regression testing an expensive activity. According to Rothermel et al. [176], complete regression testing of a software of 20,000 lines of code require around seven weeks of continuous execution. This necessitates development of many techniques to enhance the efficiency of regression testing (selection, minimization, and prioritization). Thus, the problem of regression testing is formally defined as follows:

Definition 2.1 (Regression testing)**.** *Given a program P, its modified version P', and a test suite T that is used to test P, regression testing finds a way to exercise T to restore confidence on the correctness of P'.*

Selective Regression Testing

Many researchers have devised methods as an attempt to reduce the cost and time of regression testing. Regression test selection approaches aim to reduce the cost of regression testing by selecting some relevant subset of the existing test suite. An obvious approach to select the test cases is to use the source code of a program to determine the tests that are appropriate with respect to the changes [172]. Therefore, selective retest techniques [27] attempt to identify those test cases that can exercise the modified parts of the program and the parts that are affected by the modification to reduce the cost of testing. The features of selective retest technique are as follows:

a. The resources required to retest a modified version of the program are minimized.

b. This is achieved by minimizing the number of test cases to be exercised.

c. The test suite grows uncontrollably due to the continuous modifications done to the programs for which selective retesting is required.

d. The relationship between the test cases and the program parts that are covered by the test cases can be analyzed better.

Thus, regression test case selection is formally defined as follows:

Definition 2.2. *Given the program P, its modified version P', and a test suite T, find a subset T' of the test suite $T = \{t_1, t_2, \ldots, t_n\}$ comprising of n test cases, i.e. $T' \subset T$ such that $\forall\, t_i \in T,\ t \in T' \Leftrightarrow P'(t) \neq P(t), 1 \leq i \leq n$, i.e. t_i executes all the code affected with respect to the modifications.*

Test Suite Minimization

Test suite minimization techniques aim to identify a reduced test suite that can still assure software quality. The size of the reduced test suite should therefore be much smaller than the original test suite. The minimization problem described in Definition 2.3 follows the definition given in [210]. This definition is considered as the minimal hitting set problem. This is so because it is assumed that a single test case satisfies each test requirement r_i. However, in reality, it could be different. For example, a functional test requirement may require more than one test case to satisfy. Therefore, the functional granularity of the test cases need to be defined in order to apply the given formulation of the problem. Owing to the fact that the test suite minimization problem is NP-complete by nature, many researchers have encouraged the application of different heuristics [28, 105, 131, 164, 174, 208, 210] while formulating the minimization problem. The test suite minimization as formally defined by Harman et al. [210] is given below:

Definition 2.3. *Given a test suite T, a set of test requirements $\{r_1, r_2, \ldots, r_n\}$ that must be satisfied to provide the desired adequate testing of the program, and subsets $\{T_1, T_2, \ldots, T_n\} \subset T$, each of them associated with each of the r_i such that any one of the test cases $t_j \in T_i$ can be used to achieve requirement r_i. Find a representative set, $T' \subset T$ that satisfies all r_i.*

When every test requirement in $\{r_1, r_2, \ldots, r_n\}$ is satisfied by T' then the testing criterion is said to be satisfied.

Test Case Prioritization

Regression test prioritization techniques [66, 69, 104, 108, 175, 184] attempt to find an order for executing the test cases so that the likelihood of detecting the defects is early and maximum in the regression testing process [66, 177]. There are two types of prioritization [41]:

i. *General test case prioritization*: For a given program P and a test suite T, the test cases are prioritized such that the prioritization is useful to a succession of program modifications done to P, without the knowledge of the modification.

ii. *Version specific test case prioritization*: In this approach, the test cases are prioritized whenever program P is modified to P', with the knowledge of the changes made in P.

All the existing regression test case prioritization approaches [66, 69, 104, 108, 162, 163, 175, 184] target to find an optimal ordering of the test cases based on the rate of fault detection or rate of satisfiability of coverage criterion under consideration. More formally the prioritization problem as defined by Rothermel et al. [175] is as follows:

Definition 2.4. *Given a test suite T, the set of permutations of T denoted as PT, a function f, from PT to the real numbers. Find $T' \in PT$ such that*

$$(\forall T'' \left(T'' \in PT\right) \cap \left(T'' \neq T'\right) \left[f\left(T'\right) \geq \left(T''\right)\right]),$$

where, PT is the set of all possible orderings of the test cases in T and f is a function that maps the ordering with an award value.

Rothermel et al. [175] proposed a metric to ensure the efficiency of any of the existing prioritizing techniques. This metric is called *Average Percentage of Fault Detected (APFD)* and is used by many researchers to evaluate the effectiveness of their proposed techniques. APFD measure is calculated by taking the weighted average of the number of faults detected during execution of a program with respect to the percentage of test cases executed.

Let T be a test suite and $T^{'}$ be a permutation of T. The APFD for $T^{'}$ is defined as follows:

$$APFD = 1 - \frac{\sum_{i=1}^{n-1} F_i}{n * l} + \frac{1}{2n} \tag{2.1}$$

Here, n is the number of test cases in T, l is the total number of faults, and F_i is the position of the first test case that reveals the fault i.

The value of APFD can range from 0 to 100 (in percentage). Higher is the APFD value for any ordering of the test cases in the test suite, higher is the rate at which software faults are discovered [60, 175].

Throughout our discussion of regression testing in the rest of this thesis, we will continue to use the notations described in this chapter. Therefore, we will use P' to denote a modified version of program P under test. It is important to note that any attempt to solve regression testing worth mentioning that any attempt to address the problem of regression testing in a more cost-effective way will essentially be build upon regression test selection, minimization, and prioritization, in conjunction with or in isolation from one another.

2.1.4 Test Adequacy Criteria

As mentioned in Section 2.1.2, test adequacy criteria is based on the underlying representation of program P. Many researchers have suggested different graphical representations for the programs, some of these representations are discussed in Section 2.2.2. Harrold and Rothermel [92, 93] have surveyed a number of graph-based representations along with the tool support used to construct these representations. Some more discussion on the suitability of graphical representations for object-oriented programs is given in [100, 119, 161, 198]. It should be noted that the representation chosen for the program under test will influence the assessment of the quality of existing test suites and generation of new test cases if required. These graph based representations for the programs under test can be created using Soot [194], JavaPDG [181], JOANA [1], and Java SDG API [2] [120], and many other tools in [92]. Test adequacy metrics can be viewed in light of intermediate graph representation of the program and can be defined in terms of program paths, variable values, branches, conditions, nodes, etc. that require to be exercised. We can formally state that if every test suite that satisfies C_x also satisfies C_y, then C_x subsumes C_y [143]. If C_x subsumes C_y and vice-versa, then the two adequacy criteria are equivalent; otherwise C_x is said to strictly subsume C_y. The definitions of the test adequacy criteria considered in this thesis are still applicable when different program representations are chosen. The basis of our criteria is that it is impossible to reveal a fault in P unless the faulty node from $P's$ graph is covered by some test case within T. Therefore, a test adequacy criterion should ensure the execution of all statements (nodes) in a program. Thus, we define below the **all-nodes** (statement) coverage criterion for a test suite T that tests a program P.

Definition 2.5. *Let the graphical representation of program P be $G = (N, E)$, where N refers to the nodes corresponding to the program statements, and E refers to the edges corresponding to the dependences between these statements, then a test suite T for graph G satisfies the **all-nodes** test adequacy criterion iff all $t_i \in T$, for $i = 1, 2, \ldots, k$, create a set of covered nodes N_c that include all $n \in N$ at least once.*

Intuitively, the *all-nodes* criterion is not a strong criterion because even if T satisfies this criterion, it still may not have covered all the transfer of controls (represented as dependence edges) within graph G [220]. A *while loop* in program P explains this situation. A test suite T can satisfy the coverage criterion by only executing the iteration only once. However, in the case of all-nodes(statement) coverage, T will not execute the edge that denotes the control transfer to the beginning of *while* loop. Thus, it is a necessity that **all-edges** criterion defined in Definition 2.6 should be satisfied. Definition 2.6 states that a test suite must exercise every edge within the graph.

[1]The IFC(Information flow control) console and Graph Viewer, http://pp.ipd.kit.edu/projects/joana/.

[2]A JSDG (Java System Dependence Graph) API, http://www4.comp.polyu.edu.hk/csclo/teaching/SDGAPI/.

Definition 2.6. *A test suite T for graph $G = (N, E)$ satisfies the all-edges test adequacy criterion if all $t_i \in T$, for $i = 1, 2, \ldots, n$, create a set of edges that include all $e \in E$ at least once.*

Since the inclusion of every edge in a control flow graph implies the inclusion of every node within the same CFG, it is clear that the branch coverage criterion subsumes the statement coverage criterion [220]. However, it is possible to explore and define other criteria (such as *all-path* coverage) that subsumes the all-edge criteria, which is beyond the scope of this thesis. This thesis uses the above two adequacy criteria (*all-nodes* and *all-edges*) selectively for the experimental programs.

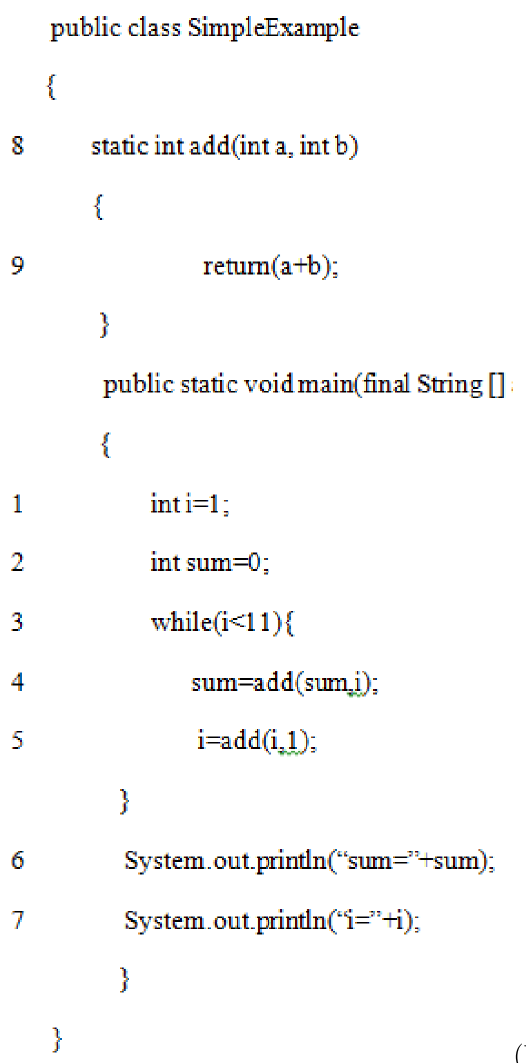
2.2 Program Slicing

Program slicing is a method of separating out the relevant parts of a program with respect to a particular computation. Thus, slice of a program is a set of statements of the program that affects the value of a variable at a particular point of interest. Program slicing was originally introduced by Mark Weiser [201] as a method for automatically decomposing programs by analyzing their data flow and control flow dependences starting from a subset of a program's behavior. Slicing reduces the program to a minimal form that still produces the same behavior. The input that the slicing algorithm takes, is usually an intermediate representation of the program under consideration [217]. Normally, the intermediate representation of the program under consideration is a graph. The first step in slicing a program involves specifying a point of interest, called the slicing criterion, which is expressed as (s, v), where s is the statement number and v is the variable that is being used or defined at s. Since last couple of decades, the area of program slicing has been enriched by contributions from several researchers. Since its inception by Mark Weiser [201] as a debugging aid, many new techniques have evolved to enhance the accuracy, preciseness, and speed up of the process of slicing and to make program slicing usable in different applications. The technique of program slicing has evolved to handle unstructured and multi-procedure programs, structured as well as object-oriented, aspect-oriented, and feature-oriented programs. Also, these slicing techniques have found application in diverse problem areas.

2.2.1 Types of Program Slices

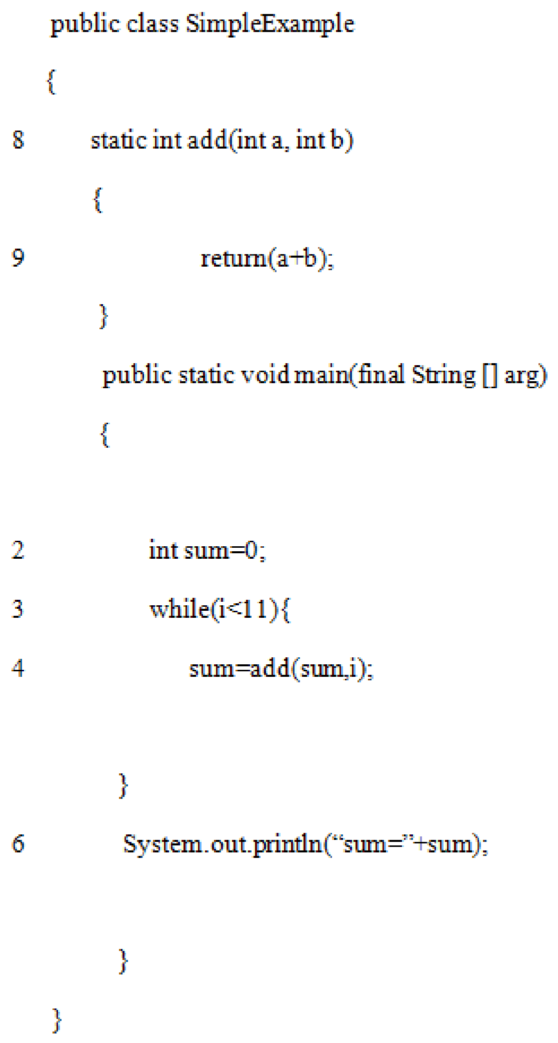
In this section, we discuss the different slicing approaches that researchers have used for different applications.

a. *Forward Slice*: It comprises of all those parts of a program that might be affected by the slicing criterion because of their dependence on the slicing criterion. Figure 2.3 shows a sample program in Figure 2.3a along with its corresponding forward slice shown in Figure 2.3b.



```
  public class SimpleExample
  {
8     static int add(int a, int b)
      {
9              return(a+b);
      }
      public static void main(final String []
      {
1          int i=1;
2          int sum=0;
3          while(i<11){
4              sum=add(sum,i);
5              i=add(i,1);
        }
6       System.out.println("sum="+sum);
7       System.out.println("i="+i);
        }
  }
```

(a) A sample program.



```
  public class SimpleExample
  {
8     static int add(int a, int b)
      {
9              return(a+b);
      }
      public static void main(final String [] arg)
      {

2         int sum=0;
3         while(i<11){
4             sum=add(sum,i);

          }
6         System.out.println("sum="+sum);

          }
  }
```

(b) Forward slice with respect to slicing criterion $< 2, sum >$.

Figure 2.3: Forward slice of a sample program.

b. *Backward Slice*: It comprises of all those parts of a program that might affect the slicing criterion because of the dependences of the slicing criterion on those parts. Figure 2.4 shows a sample program in Figure 2.4a along with its corresponding backward slice shown in Figure 2.4b.

```
    public class SimpleExample
    {
8       static int add(int a, int b)
        {
9               return(a+b);
        }
        public static void main(final String []
        {
1           int i=1;
2           int sum=0;
3           while(i<11){
4               sum=add(sum,i);
5               i=add(i,1);
            }
6           System.out.println("sum="+sum);
7           System.out.println("i="+i);
        }
    }
```

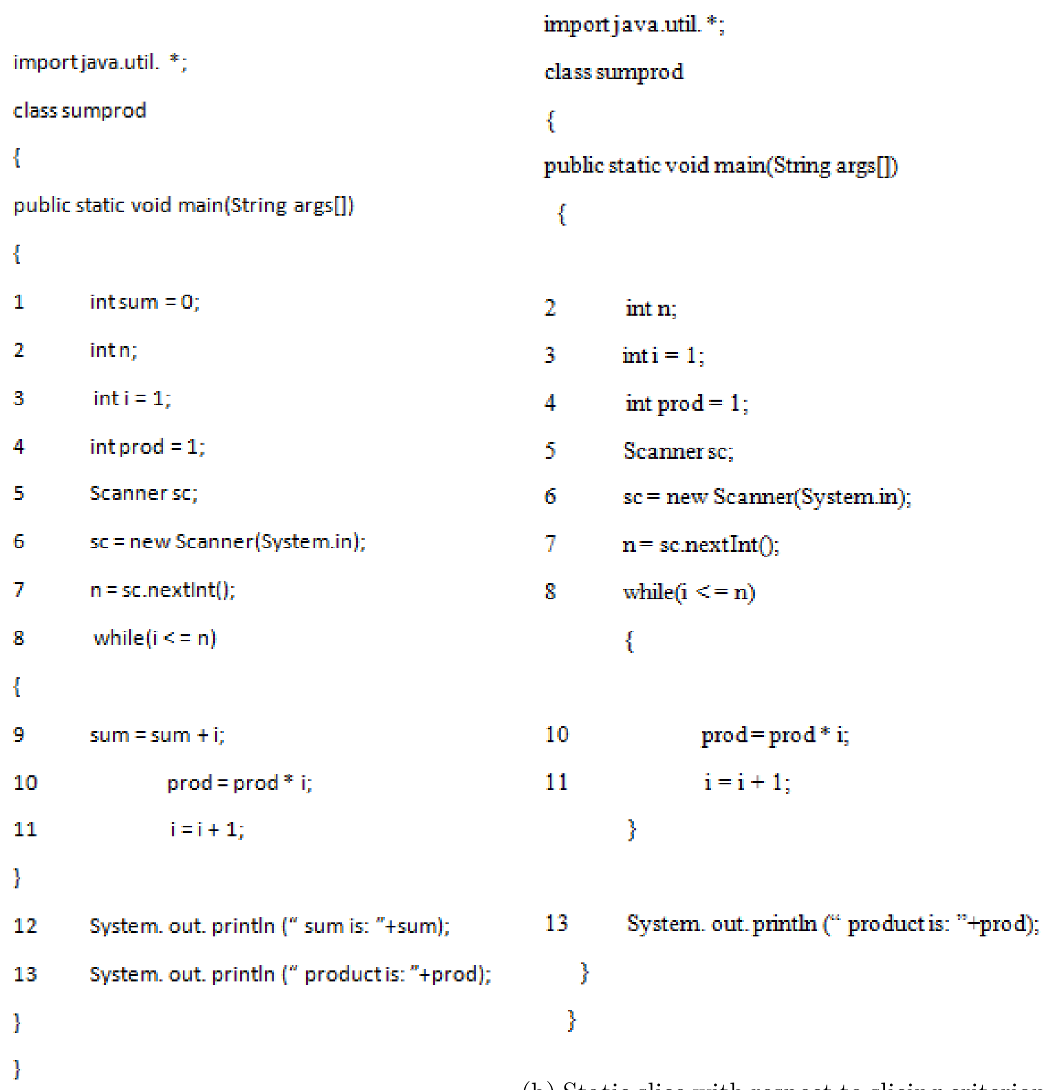
(a) A sample program.

```
    public class SimpleExample
    {
8       static int add(int a, int b)
        {
9               return(a+b);
        }
        public static void main(final String [] arg)
        {
1           int i=1;

3           while(i<11){

5               i=add(i,1);
            }

7           System.out.println("i="+i);
        }
    }
```

(b) Backward slice with respect to slicing criterion $< 7, i >$.

Figure 2.4: Backward slice of a sample program.

c. *Static Slice*: It comprises of those statements of a program that we get by statically analyzing the code, i.e. by examining some representation of the code without actually executing the program under consideration. Figure 2.5 shows a sample program in Figure 2.5a along with its corresponding static slice shown in Figure 2.5b.



```
import java.util. *;
class sumprod
{
public static void main(String args[])
{
1       int sum = 0;
2       int n;
3       int i = 1;
4       int prod = 1;
5       Scanner sc;
6       sc = new Scanner(System.in);
7       n = sc.nextInt();
8       while(i < = n)
{
9       sum = sum + i;
10              prod = prod * i;
11              i = i + 1;
}
12      System. out. println (" sum is: "+sum);
13      System. out. println (" product is: "+prod);
}
}
```

(a) A sample program.

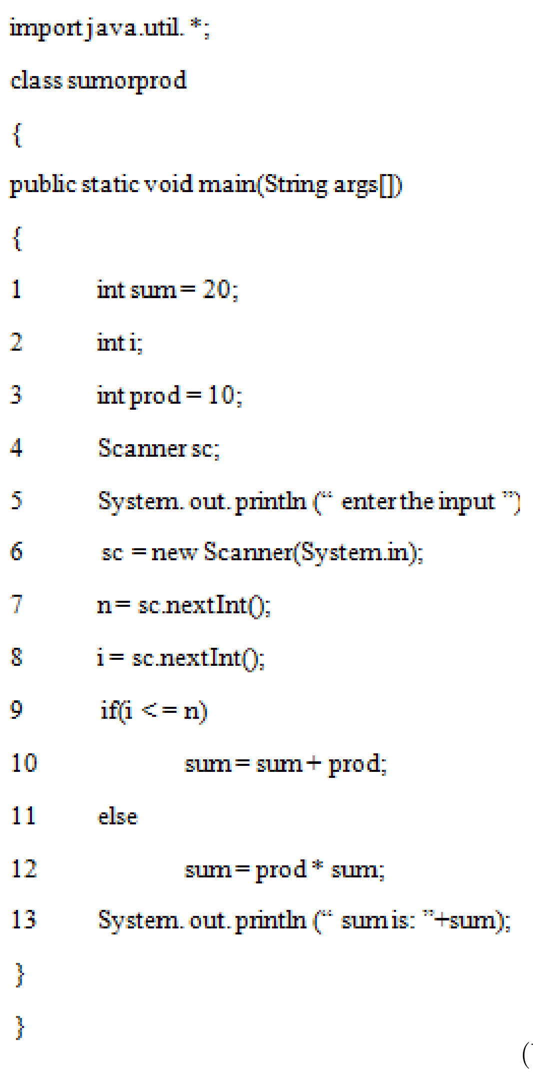
```
import java.util. *;
class sumprod
{
public static void main(String args[])
 {

2       int n;
3       int i = 1;
4       int prod = 1;
5       Scanner sc;
6       sc = new Scanner(System.in);
7       n= sc.nextInt();
8       while(i <= n)
        {

10              prod= prod * i;
11              i = i + 1;
        }

13      System. out. println (" product is: "+prod);
   }
  }
```

(b) Static slice with respect to slicing criterion $< 13, prod >$.

Figure 2.5: Static slice of a sample program.

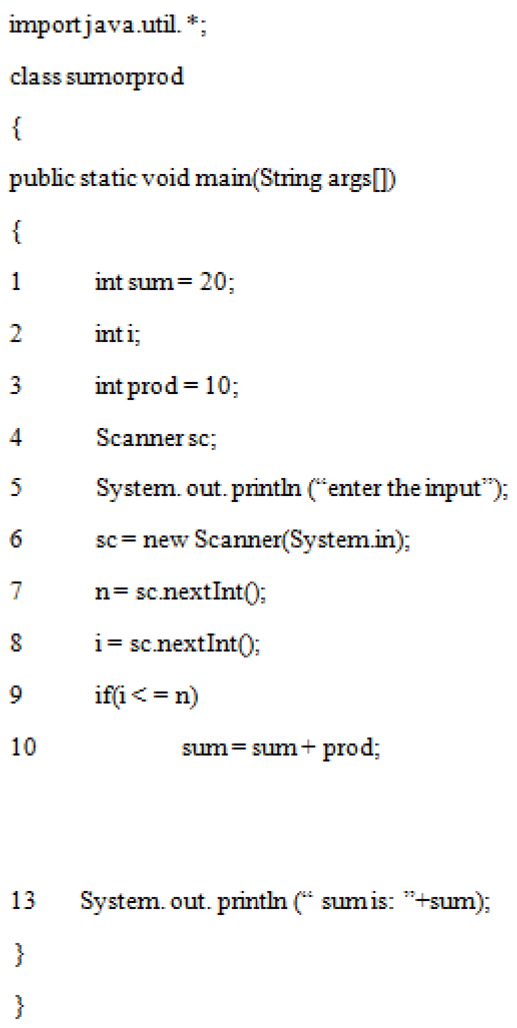
d. *Dynamic Slice*: It comprises of all those parts of the program that we obtain by actually executing the program with a specific input (included in the slicing criterion). Thus, a dynamic slice is only correct for a specific input whereas a static slice is correct for all inputs. Figure 2.6 shows a sample program in Figure 2.6a along with its corresponding dynamic slice shown in Figure 2.6b.

Over time many researchers have come up with many slicing techniques such as *Quasi-slicing* [195] and *conditional slicing* [39] that combine static and dynamic



```
import java.util. *;
class sumorprod
{
public static void main(String args[])
{
1       int sum = 20;
2       int i;
3       int prod = 10;
4       Scanner sc;
5       System. out. println (" enter the input ")
6        sc = new Scanner(System.in);
7       n= sc.nextInt();
8       i = sc.nextInt();
9        if(i < = n)
10              sum = sum + prod;
11      else
12              sum = prod * sum;
13      System. out. println (" sum is: "+sum);
}
}
```

(a) A sample program.



```
import java.util. *;
class sumorprod
{
public static void main(String args[])
{
1       int sum = 20;
2       int i;
3       int prod = 10;
4       Scanner sc;
5       System. out. println ("enter the input");
6       sc = new Scanner(System.in);
7       n= sc.nextInt();
8       i = sc.nextInt();
9       if(i < = n)
10              sum = sum + prod;



13    System. out. println (" sum is: "+sum);
}
}
```

(b) Dynamic slice with respect to slicing criterion $< 13, sum, i = 5 \& n = 9 >$.

Figure 2.6: Dynamic slice of a sample program.

slicing. Most of the slicing techniques proposed are syntax preserving, but if the slicer is allowed to make syntactic changes as long as the relevant semantics are preserved then this type of slicing is known as *amorphous slicing* [87]. Some of the recent applications and developments of program slicing can be found in [13, 136, 164, 199, 203, 209].

There are various aspects to be considered in slicing a program. They are listed as follows:

a. *Slicing variable*: Slicing variable may be based on the variables specified in the criteria (slicing point of interest) or it may be on all the variables.

b. *Slicing point*: Considering the slicing point, a programmer's interest may be in observing the impact before or after a particular statement [195].

c. *Scope*: The scope of the slice may be inter-procedural or intra-procedural [191]. But, many researchers [119, 123, 130, 152, 154, 182] have extended the scope to class level for OO programs.

d. *Slicing direction*: The expected slice of the program may be either in forward direction or backward direction.

e. *Abstraction level*: Abstraction level is either in statement or in procedure level. But considering the typical features of the OO programs, it needs to be extended to class or package level, taking into account the dependences induced by them.

f. *Type of information*: The information that we obtain from the slice can be either static or dynamic.

g. *Computational method*: Traditionally, the method of computing the slice was based on solving the data flow equations as a graph reachability problem [201]. But, over the years many researchers have proposed many other techniques such as marking and unmarking of the nodes [69, 153], marking and unmarking of the edges [152, 184], graph coloring [19], etc. for computing the slices.

h. *Output format*: The format obtained after slicing may be in either of the forms: code, dependence graph or execution tree.

2.2.2 Program Representation

Various types of program representation schemes exist which include high level source code, pseudo-code, a set of machine instructions in a computer's memory, a flow chart and others. The purpose of each of these representations depends upon the exact context of use. Different representations may be required to facilitate human readability, annotation for verifiability, and transformation for running a program on platforms such as multiprocessors and distributed computers, etc. In the context of program slicing, program representations are used to support efficient automation of slicing.

For a very simple program, a slice for any given slicing criterion can be determined manually. But for large sized and complex programs, automation of the slicing process is essential. The current automated slicing techniques require transforming the source code of the program into some mathematical representation during the slicing process. Various representation schemes have resulted from the search for ever more complete and efficient slicing techniques.

In the following, we present a few basic concepts associated with intermediate program representations. A feature shared by most of the slicing algorithms is that programs are represented by a directed graph that captures the data and control dependences.

It is important to note that there is no single correct way of constructing these intermediate graphs, nor there is a freezed set of exact information that must be available for slicing. Researchers have come up with different techniques and representations that best suits the problem at hand. However, each of these new techniques and representations is built upon its predecessor techniques. Therefore, a discussion on the existing representation schemes is not trivial, but these are not necessarily faithful to any single researcher's style.

Control Flow Graph

The *control flow graph* (CFG) is an intermediate representation for programs that is useful for data flow analysis and for many optimizing code transformations such as common sub-expression elimination, copy propagation, and loop invariant code motion [70, 100, 161].

Definition 2.7 (***Control Flow Graph***)**.** *Let the set N represent the set of statements of a program P. The* control flow graph *of program P is the flow graph $G = (N_1, E, Start, Stop)$ where $N_1 = N \cup \{Start, Stop\}$. An edge $(m, n) \in E$ indicates the possible flow of control from the node m to the node n.*

Note that the existence of an edge (x, y) in the control flow graph means that control *must* transfer from x to y during program execution. Fig. 2.7 represents the CFG of the example program given in Fig. 2.6a. The CFG of a program P models the branching structures of the program, and it can be built while parsing the source code using algorithms that have linear time complexity in the size of the program [18].

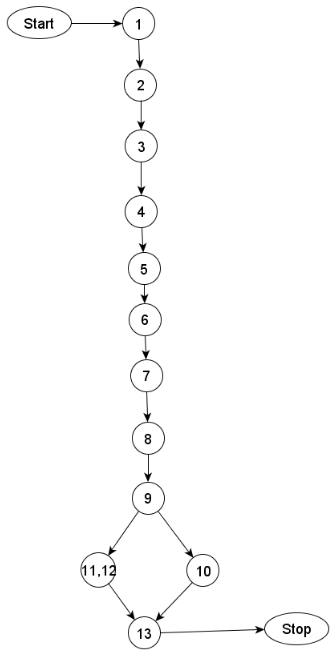


Figure 2.7: The CFG of the example program given in Figure 2.6a.

Data Dependence Graph

The CFG of a program represents the flow of control through the program. However, the concept that is often more useful in program analysis is the flow of data through a program. Data flow describes the flow of the values of variables from the points of their definitions to the points where their values are used.

Definition 2.8 (***Data Dependence***)**.** *Let G be the CFG of a program P. A node n is said to be* data dependent *on a node m if there exits a variable* var *of the program P such that the followings hold:*

(i) *node m defines* var*,*

(ii) *node n uses* var*, and*

(iii) *there exists a directed path from m to n along which there is no intervening definition of* var*.*

Consider the example program given in Fig. 2.6a and its CFG in Fig. 2.7. Node 9 has data dependence on each of the nodes 7, and 8. Similarly, node 12 has data dependence on node 1 and node 3.

The term *reaching definition* is used to mean that a value defined at a node may be used at another nodes [81, 101]. That is, node x is a reaching definition for a node y iff y is data dependent on x. A data dependence from node x to node y indicates that a value computed at x may be used at y under some path through the control flow graph. A dependence from x to y is a conservative approximation, which says that under some conditions a value computed at x may be used at y.

Definition 2.9 (***Data Dependence Graph***)**.** *The data dependence graph of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P and $(x, y) \in E$ iff x is data dependent on y.*

Control Dependence Graph

Ferrante et al. [70] introduced the notion of *control dependences* to represent the relations between program entities arising due to control flow.

Definition 2.10 (***Control Dependence***)**.** *Let G be the CFG of a program P. Let x and y be two nodes in G. Node y is* control dependent *on a node x if the followings hold:*

(i) there exists a directed path D from x to y,

(ii) y post-dominates every z in D (excluding x and y), and

(iii) y does not post-dominate x.

If x and y are two nodes in a flow graph then x dominates y iff every path from *Start* to y passes through x. y post-dominates x iff every path from x to *Stop* passes through y. Let x and y be nodes in a flow graph G. Node x is said to be immediate post-dominator of node y iff x is a post-dominator of y, $x \neq y$ and each post-dominator $z \neq x$ of y post-dominates x. The post-dominator tree of a flow graph G is the tree that consists of the nodes of G, has the root *Stop*, and has an edge (x, y) iff x is the immediate post-dominator of y.

Let x and y be two nodes in the CFG G of a program P. If y is *control dependent* on x, then x must have multiple successors in G. Conversely, if x has multiple successors, then at least one of its successors must be control dependent on it. Consider the example program given in Fig. 2.6a and its CFG in Fig. 2.7. Each of the nodes 10 and 12 is control dependent on node 9. Note that node 9 has two successor nodes 10 and 12.

Definition 2.11 (***Control Dependence Graph***)**.** *The control dependence graph of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P and $(x, y) \in E$ iff x is control dependent on y.*

Program Dependence Graph

Ferrante et al. [70] presented a new mechanism of program representation called *Program Dependence Graph* (PDG). Unlike the flow graphs, an important feature of PDG is that it explicitly represents both control and data dependences in a single program representation. A PDG models a program as a graph, where the nodes refer to the statements, and the edges refer to the inter-statement data or control dependences.

Definition 2.12 (***Program Dependence Graph (PDG)***)**.** *The program dependence graph G of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P. The graph contains two kinds of directed edges:* control dependence edges *and* data dependence edges. *A control (data) dependence edge (m, n) indicates that n is control (data) dependent on m.*

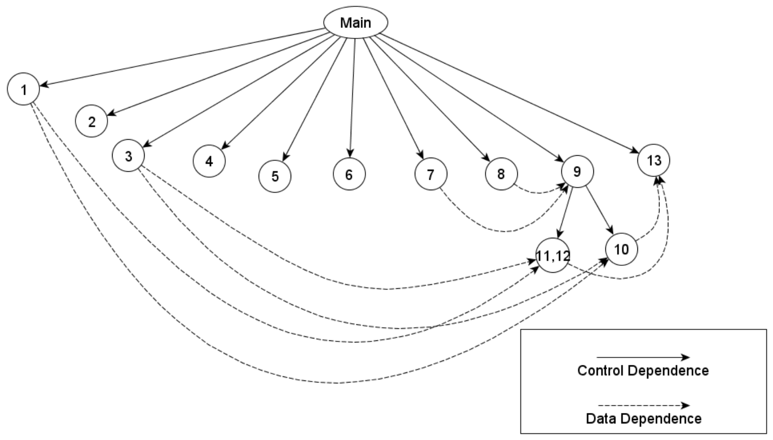


Figure 2.8: The PDG of the example program given in Figure 2.6a.

Note that the PDG of a program P is the union of a pair of graphs: the *data dependence graph* and the *control dependence graph* of P. Consider the program given in Fig. 2.6a. Its PDG is given in Fig. 2.8. In Fig. 2.8, the nodes of the graph represent the statements of the example program given in Fig. 2.6a. The *solid* edges represent the control dependences and the *dotted* edges represent the data dependences. The program dependence graph of a program P can be built from its control flow graph in $O(n^2)$ time, where n is the number of nodes in the control flow graph [70].

System Dependence Graph

The PDG of a program combines the control dependences and the data dependences into a common framework. The PDG has been found to be suitable for intra-procedural slicing. However, it cannot handle procedure calls. Horwitz et al. [102] enhanced the PDG representation to facilitate inter-procedural slicing. They introduced the *System Dependence Graph* (SDG) representation that models the main program together with all associated procedures. The *SDG* is an extension of the *PDG*, and models the programs with the following properties [102]:

- A complete program consists of a main program and a collection of auxiliary procedures.
- Procedures end with *return* statements. A *return* statement does not include a list of variables.
- Parameters are passed by value-results.

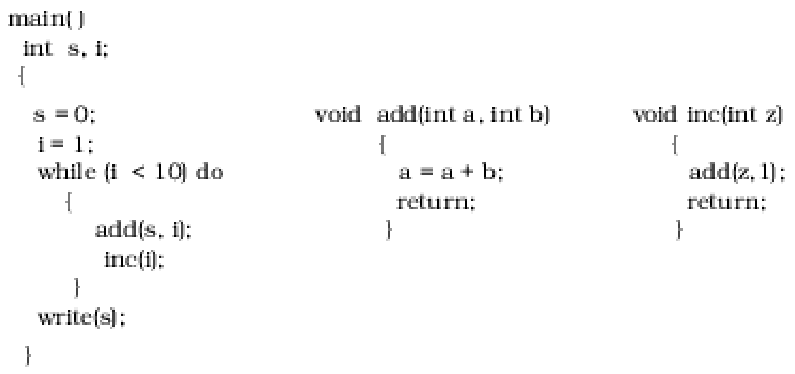
The *SDG* is very similar to the *PDG*. Indeed, a *PDG* of the *main* program is a subgraph of the *SDG*. In other words, for a program without procedure calls, the *PDG* and *SDG* are identical. The technique for constructing an *SDG* consists of first constructing a *PDG* for every procedure, including the *main* procedure, and then adding auxiliary dependence edges which link the various subgraphs together. This results in a program representation which includes the information necessary for slicing across procedure boundaries.

An *SDG* includes several types of nodes to model procedure calls and parameter passing:

- *Call-site nodes* represent the procedure call statements in a program.
- *Actual-in* and *actual-out nodes* represent the input and output parameters at call sites. They are control dependent on the call-site nodes.
- *Formal-in* and *Formal-out nodes* represent the input and output parameters at the called procedures. They are control dependent on the procedure's entry node.

Control dependence edges and data dependence edges are used to link the individual PDGs in an SDG. The additional edges used to link the PDGs together are as follows:

- *Call edges* link the call-site nodes with the procedure entry nodes.
- *Parameter-in edges* link the actual-in nodes with the formal-in nodes.
- *Parameter-out edges* link the formal-out nodes with the actual-out nodes.



```
main( )
 int  s, i;
{
  s = 0;                      void  add(int a, int b)      void inc(int z)
  i = 1;                            {                          {
  while (i  < 10) do                  a = a + b;                 add(z, 1);
     {                                return;                    return;
        add(s, i);                  }                          }
         inc(i);
     }
  write(s);
 }
```

Figure 2.9: An example program consisting of a main program and two procedures.

Finally, *summary edges* are added to represent the transitive dependences that arise due to procedure calls. A *summary edge* is added from an actual-in node A to an actual-out node B, if the value associated with the actual-in node A affects the value associated with the actual-out node B, due to transitive flow of dependence. The transitive flow of dependence may be caused by data dependences, control dependences or both. Fig. 2.10 represents the SDG of the example program given in Fig. 2.9.

The intermediate program representation is analyzed to compute a static slice. Horwitz et al. [102] developed *system dependence graph* (SDG) as an intermediate program representation and proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. The first pass of the inter-procedural slicing algorithm traverses backward along all the edges of the SDG except parameter-out edges, and marks those vertices reached. The second pass traverses backward from all vertices marked during the first pass along all edges except call and parameter-in edges, and marks the reached vertices. The slice is union of the vertices marked during pass one and pass two.

Larson and Harrold [124] enhanced the SDG [102] to represent object-oriented programs. Their SDG successfully represents object-oriented features such as method

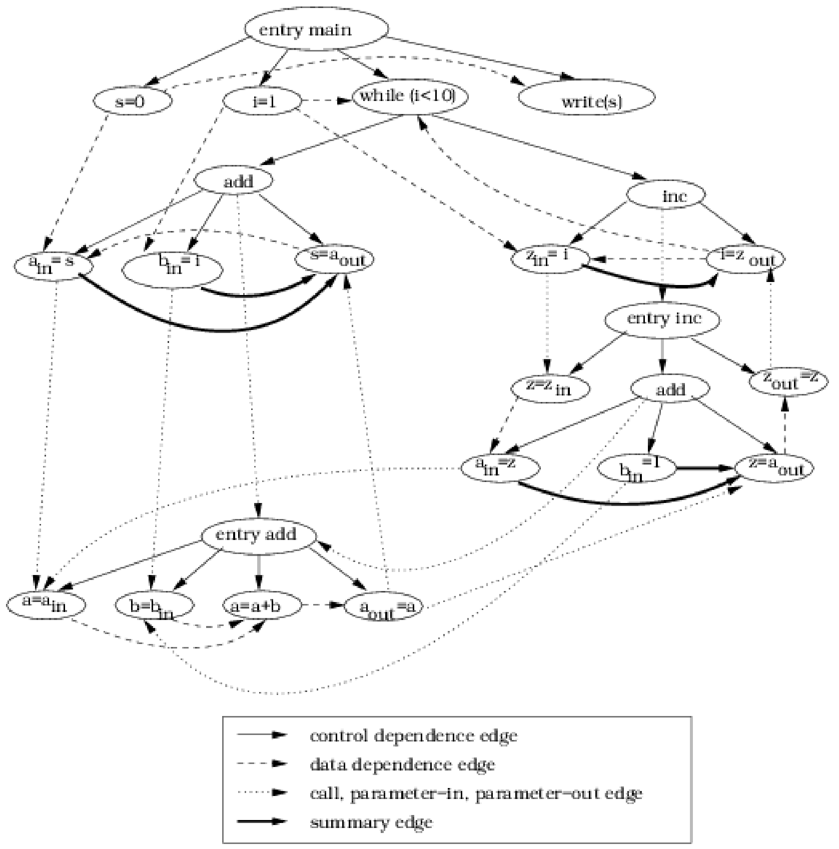


Figure 2.10: The SDG of the example program given in Figure 2.9.

calls, inheritance, and polymorphism. After constructing the SDG for a complete object-oriented program, they have used the two-pass graph reachability algorithm [102] for computing static slices. One limitation of this approach is that it fails to consider the fact that in different method invocations, the data members used by the methods might belong to different objects. So, the resulting data dependences become imprecise. A second limitation of the approach is that it does not handle cases in which an object is used as a parameter or as a data member of another object.

Object-Oriented Program Dependence Graphs

Krishnaswamy [119] proposed another dependence-based representation called the *object-oriented program dependence graph* (OPDG) to represent the object-oriented programs. The OPDG of an object-oriented program represents control flow, data dependences and control dependences. The OPDG representation of an object-oriented program is constructed in three layers, namely: *Class Hierarchy Subgraph* (CHS), *Control Dependence Subgraph* (CDS), and *Data Dependence Subgraph* (DDS). The CHS represents inheritance relationship between classes, and the composition of methods into a class. A CHS contains a single *class header node* and a *method header node* for each method that is defined in the class.

Inheritance relationships are represented by edges connecting class headers. Every method header is connected to the class header by a *membership edge*. Subclass representations do not repeat representations of methods that are already defined in the super classes. *Inheritance edges* of a CHS connect the class header node of a derived class to the class header nodes of its super classes. *Inherited membership edges* connect the class header node of the derived class to the method header nodes of the methods that it inherits.

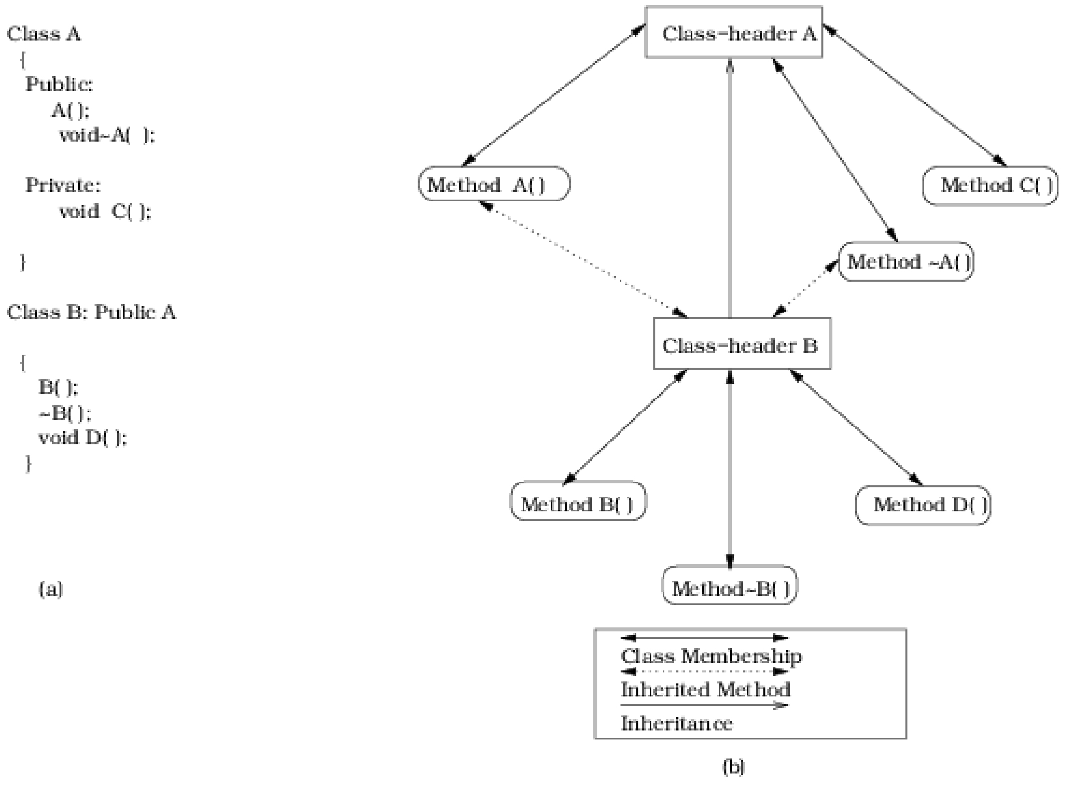
A CDS represents the static control dependence relationships that exists within and among the different methods of a class. The DDS represents the data dependence relationship among the statements and predicates of the program. The OPDG of an object-oriented program is the union of three subgraphs: CHS, CDS, and DDS. Slices can be computed using OPDG as a graph-reachability problem. He also computed the polymorphic slices of object-oriented programs based on the OPDG. The OPDG of an object-oriented program is constructed as the classes are compiled and hence it captures the complete class representations.

The main advantage of OPDG representation over other representations is that OPDG is generated only once during the entire life of the class. It does not need to be changed or regenerated as long as the class definition remains unchanged. Fig. 2.11 shows an example program and its CHS.

2.3 Precision and Correctness of a Slice

Let P be a program, and S be a static slice of P with respect to a slicing criterion C. In the original definition of Weiser [201], the reduced program S is required to be an executable program and its behavior with respect to the slicing criterion must be same as the original program P. A slice S of P with respect to a slicing criterion C is *statement-minimal* if no other slice of P with respect to the slicing criterion has fewer statements than S. Weiser [201] has shown that the problem of computing *statement-minimal* slices is undecidable.

Another common definition of a static slice is the following: a slice S of a program P with respect to a slicing criterion C is a subset of the statements of the



```
Class A
 {
  Public:
     A( );
       void~A(  );

  Private:
       void  C( );

 }

Class B: Public A

 {
    B( );
    ~B( );
    void D( );
  }
```

Figure 2.11: (a) An example program, and (b) its CHS

program that directly or indirectly affect the slicing criterion [70, 102, 161]. Note, that such a slice need not be executable. Unless specified otherwise, we shall follow this definition of a slice throughout the discussion in the thesis.

Let G_C be the control flow graph (CFG) of a program P. In all the existing program slicing frameworks, for each statement s in the program P, two sets are maintained. One set contains the variable names used at s and the other set contains the variable names defined at s. The inter-statement dependences in the program P are captured using the CFG G_C and the variable names in these two sets, for each statement s.

```
         integer m, i,
1.      read(m);
2.      i = 1;
3.      x = 4;
4.      z = m - m;
5.      write(z);
```

Figure 2.12: An example program

Note that Statement 4 of the example program given in Fig. 2.12 uses variable m. Though Statement 4 assigns the value zero ($m - m$) to the variable z, it has dependence on Statement 1 in the program slicing frameworks since Statement 1 is a reaching definition of the variable m for Statement 4.

It is therefore reasonable to define the precision of a slice in the existing program slicing frameworks as follows: *A slice is said to be precise if it contains only those statements that actually affect the slicing criterion.*

Note that a precise slice need not be a *statement-minimal* slice. Consider the example program given in Fig. 2.12. For any input value of the variable m, the *statement-minimal* slice with respect to the slicing criterion $< 4, z >$ should be empty set as z is always assigned the value $0 = m - m$. In the existing program slicing frameworks, the precise slice for the slicing criterion is certainly $\{1\}$ as Statement 1 is a reaching definition of the variable m for the Statement 4.

A slice is said to be *correct* if it contains all the statements that affect the slicing criterion. A slice is said to be *incorrect* if it fails to contain some statements that affect the slicing criterion. Note that the whole program is always a correct slice of any slicing criterion. A correct slice is *imprecise* if it contains at least one statement that does not affect the slicing criterion.

2.4 Applications of Program Slicing

This section describes the use of program slicing techniques in various applications. In trying to use the basic slicing concepts in diverse domains, several variations of the notions of program slicing as described in Section 2.2.1 are developed. The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging. From this modest beginning, the use of program slicing techniques has now ramified into a powerful set of tools for use in such diverse applications as program understanding, program verification, automated computation of several software engineering metrics, software maintenance and testing, functional cohesion, dead code elimination, reverse engineering, Parallelization of sequential programs, software portability, reusable component generation, compiler optimization, program integration, showing differences between programs, software quality assurance, etc. [73, 81, 88, 106, 190, 202, 217]. A comprehensive study on the applications of program slicing is made by Binkley and Gallagher [25] and Lucia [54]. In the following, we briefly discuss some of these applications of program slicing that are relevant to our work.

2.4.1 Testing

Software maintainers often carry out regression testing. Regression testing essentially implies retesting software after modification [24, 143, 188]. Even after the smallest change to a piece of code, extensive tests may be necessary which might involve running a large number of test cases to rule out any unwanted behavior arising due to the change. While decomposition slicing eliminates the need for regression testing on the complement, there may still be a substantial number of tests to be run on the dependent, independent and changed parts. Slicing can be used to reduce the number of these tests.

Suppose a program modification only changes the value of the variable x at program point p. If the forward slice with respect to x and p is disjoint from the coverage of regression test t, then the test t does not have to be rerun. Suppose a coverage tool reveals that a use of variable x at program point p has not been tested. What input data is required in order to cover p? The answer lies in the backward slice of x with respect to p. A lot of work has also been reported in order to test programs incrementally, to simplify testing, to apply program slicing to regression testing, to partition testing, and to test path selection [45, 81, 86, 88, 191].

2.4.2 Debugging

Finding bugs in a program is always challenging. The process of finding a bug usually involves running the program over and over, learning more and narrowing down the search each time, until the bug is finally located. Program slicing was originally proposed by observing the operation typically carried out by programmers while debugging a piece of code. Programmers mentally slice a code while debugging it. Even after several advancements to the basic slicing techniques, program debugging remains a main application area of slicing techniques. Debugging can be a difficult task when one is confronted with a large program, and little clues regarding the location of a bug. During debugging, a programmer usually has a test case in mind which causes the program to fail. Program slicing is useful for debugging, because it potentially allows one to ignore many statements in the process of localizing the bug [139, 202]. If a program computes an erroneous value for a variable x only the statements in the slice with respect to x have (possibly) contributed to the computation of that value; all statements which are not in the slice can safely be ignored.

Several variants of program slicing have been developed to further assist the programmer during debugging: *program dicing* [139] identifies statements those are likely to contain bugs by using information that some variables fail some tests while others pass all tests. Slices can be combined with each other in different ways: for example, the intersection of two slices contains all statements that lead to an error in both test cases; the intersection of slice a with the complement of slice b excludes from slice a all statements that do not lead to an error in the second test case. Another variant of program slicing is *program chopping* [170]. It identifies statements that lie between two points a and b in the program that are affected by a change made at a. This can be useful when a change at a causes an incorrect result at b. Debugging should be focused on the statements between a and b that transmit the change of a to b.

2.4.3 Software Maintenance

Software maintenance is a costly process because each modification to a program must take into account many complex dependence relationships in the existing software. The main challenges in effective software maintenance, are to understand various dependences in an existing software and to make changes to the existing software without introducing new bugs. One of the problems in software maintenance is that of the *ripple effect*, i.e., whether a code change in a program will affect the behavior of other codes of the program. To avoid this problem, it is necessary to know which variables in which statements will be affected by a modified variable, and which variables in which statements will affect a modified variable during software maintenance. The needs can be satisfied by *slicing* the program being maintained [73].

2.4.4 Change Impact Analysis

One of the important activity of the software maintenance phase is regression testing. In regression testing only those parts are tested that are affected by the changes made to the program. Thus, it is necessary to have some mechanism that identifies these affected program parts. Software change impact analysis is the mechanism of finding out the unpredicted and potential effect of the changes and the propagation of the impact to other parts of the program. Slicing has been one of the options for change impact analysis [186]. Program slicing identifies those statements in a program that are affected by some slicing criterion, making it useful for change impact analysis.

2.4.5 Software Quality Assurance

Software quality assurance auditors have to locate safety critical code and to ascertain its effect throughout the system. Program slicing can be used to locate all code that influences the values of variables that might be part of a safety critical component. But beforehand these critical components have to be determined by domain experts.

One possible way to assure high quality is to make the system redundant [25, 143]. If two output values are critical, then these output values should be computed independently. They should not depend on the same internal functions, since the same error might manifest in both output values in the same way, thereby hiding the error. One technique to defend against such errors is to use functional diversity, where multiple algorithms are used for the same purpose. Thus the critical output values depend on different internal functions. Program slicing can be used to determine the logical independence of the slices computed for the two output values.

2.4.6 Functional Cohesion

Cohesion measures the relatedness of the code of some component [46]. A highly cohesive software component is one that has only one function that is indivisible. Bieman and Ott [22] define *data slices* to consist of data tokens (instead of statements). Data tokens may be variables of constant definitions and references. Data slices are computed for each output of a procedure (e.g. output to a file, output parameter, assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices. They are called *glue*. The *glue* binds the slices together. The tokens that are in every data slice of a function are called *super-glue*. *Strong functional cohesion* can be expressed as the ratio of *super-glue* tokens to the total number of tokens in the slice, whereas *weak functional cohesion* may be seen as the ratio of *glue* tokens to the total number of tokens. The *adhesiveness* of a token is another measure expressing how many slices are glued together by that token. Many researchers have contributed to use of program slicing to compute functional cohesion [5, 14, 16, 79, 151, 218].

2.4.7 Functional Coupling

Coupling is defined as the degree of interdependence between two modules. In procedure-oriented programs, two modules are said to be coupled if they interchange data among them during function calls or if the interaction occurs through some shared data. In these circumstances, the modules are said to be tightly coupled. Coupling gives the complexity of a module. Slicing based approach [89, 150] is used to measure how one module affects another module in a traditional software system.

Henry and Kafura [97] computed coupling based on information flow and Harman et al. [89] used program slicing to compute coupling in traditional software systems. Harman et al. [89] proposed to measure the information flow between two functional modules through static slicing and then measured coupling through this information flow. An empirical study of measuring coupling using slicing techniques is given in [79, 150]. Coupling measures have been extensively used by researchers in diverse applications such as to predict fault proneness of the modules, maintainability, modularization drivers, impact analysis etc. [10, 33, 37, 58, 156].

2.4.8 Other Applications of Program Slicing

As mentioned earlier, program slicing methods have been used in several other applications such as compiler optimization, detecting dead code, software portability analysis, program understanding, program verification, measuring class cohesion, etc. These applications can be found in [20, 46, 122, 143, 166, 202].

2.5 Summary

In this chapter, we have discussed some definitions and concepts that will be used later in our thesis. We have discussed various aspects associated with software testing. We have also discussed program slicing and various intermediate program representations. We have also briefly dealt in *precision* and *correctness* issue of slices. Finally, we provided an overview of some important applications of program slicing.

Chapter 3

Review of Related Work

This chapter presents an overview of the basic program slicing and regression testing techniques and includes a brief history of their development. First, we discuss the work done by previous researchers on slicing of object-oriented programs in Section 3.1. We also discuss various slicing techniques available for object-oriented programs in Section 3.1.1 and for Java programs in Section 3.1.2. We discuss the work of previous researchers in the field of regression testing in Section 3.2. Some of the existing related work especially on test case selection is given in Section 3.2.1, test suite minimization in Section 3.2.2, and test case prioritization in Section 3.2.3. Then, we review the work reported on different change impact analysis (CIA) techniques in Section 3.3. As we could not find any reported work in literature on *change impact quantification* of the changes made to the object-oriented programs, we briefly discuss *the techniques available for change impact analysis.* Finally, we provide a brief summary of this chapter in Section 3.4.

3.1 Program Slicing

Many researchers [3, 101, 123, 161, 201] have proposed several methods for slicing of programs. The original work proposed by Weiser [201] focused on computing the slices from the control flow graph of the program. Weiser [201] defined slice with respect to a slicing criterion $< S, V >$, where S is a program point and V is the subset of variables at that point. The slices he computed are primarily executable programs and were obtained by removing zero or more statements from the original program. In his proposed algorithm he used data flow analysis of the control flow graph of the program to compute inter-procedural and intra-procedural slices. Ottestein and Ottenstein [161], for the first time defined slicing as a graph reachability problem. In their method, they used a program dependence graph to compute the static slices of a program. However, the limitation of this approach is that it works for a single method program. When a program comprises of more than one method then function calls and inter-method communication cannot be represented by a PDG.

The concept of *System Dependence Graphs (SDG)* to represent the inter-procedural programs was introduced by Horowitz et al. [101]. They came up with the two pass static backward slicing algorithm to find out the static backward slice of a statement in a program having multiple procedures. This algorithm is more precise than the previous one proposed by Ottenstein [161] because it uses the summary information at the call site nodes to account for the calling context of the procedure. In the first pass of the two pass graph reachability algorithm, he traversed along the summary edges to slice across the call vertices that have transitive dependences on actual-in vertices.

In 2006, Jehad Al Dallal [4] introduced a method for computing intra-procedural static forward slices by traversing the dependence graphs only once. The algorithm, named *ComputeAllForwardSlices*, invokes a function called *ComputeAFSlice*. An empty set is assigned to each node in the PDG before applying the algorithm to it. After the algorithm is applied, the set associated with a node n comprises of those statements that are included in the slice computed at n. It builds the set associated with each node in the PDG incrementally as the function *ComputeAFSlice* iterates recursively.

Alomari et al. [12] presented an efficient and lightweight forward static slicing approach. This technique does not compute slices based on the program/system dependence graph. It instead computes the dependence and control information on the fly while computing the slice on a variable. The resultant slice comprises a list of line numbers, dependent variables, aliases, and function calls concerning all variables of the system. The approach transforms the source code into an XML representation, and is implemented on a tool called *srcSlice*. The tool currently supports only C/C++ programs. This approach is highly scalable and with accuracy can generate the slices for all variables of the program.

Lisper et al. [136] proposed a light-weight inter-procedural algorithm for *backward static slicing*. In this approach, the data dependence analysis is done using the *Strongly Live Variables (SLV)* analysis to avoid the construction of the Data Dependence Graph. It allows to slice the program statements âĂIJon-the-flyâĂİ during the SLV analysis making it potentially faster for computing few slices. The use of an abstract *interpretation-based value analysis* allows slicing of low-level code. It dynamically calculates the addresses of data as a result the dependences are not relevant. The computed slice consists of the Control Flow Graph nodes and shows better accuracy over program dependence graph (PDG) based approaches [119, 161].

3.1.1 Slicing of Object-Oriented Programs

Slicing of object-oriented (O-O) programs throws special challenges than the traditional procedure-oriented programs. The presence of special features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism in O-O programs require special consideration as they introduce new dependences among the statements. Although these features are strengths of O-O programs but these may affect the correctness of the slices.

Larsen and Harrold [123] were the first to consider the O-O features in the intermediate representation of the program and overcome the challenges that they posse to program slicing. Larsen and Harrold [123] extended the concept of SDG [101] to incorporate the object-oriented features. In this method, each class is represented by a *Class Dependence Graph (ClDG)*. A ClDG represents both the control dependences and data dependences inside a class. A forward slice with respect to a slicing criterion $< s, v >$ is defined as the set of all statements which are affected by the variable v at the program point s. This approach faces with one limitation that the data dependences obtained using the approach for creating the individual procedure dependence graphs are imprecise. It treats the data members of the class as the global variable of the member methods. It fails to incorporate the fact that these data members belong to a different object each time the method is invoked by that object. Second, ClDG has no provision of representing the objects as parameters to the methods.

Tonella et al. [193] tried to overcome the limitations of [123]. To address the first limitation, they allowed a method's signature to include the data members of the class as formal parameters. This enabled an object to pass its data members as actual parameters into the method. This approach becomes unnecessarily expensive as it requires to represent each method call site with actual parameter vertices for all data members of the object. But, in actual conditions only a few of them are referenced by the method. They created an object as a single vertex when the object is used as a parameter. However, this representation may result in imprecise slices as it will cause the slice to include all the data members of the object even when not all of them would affect the slicing criterion.

Liang et al. [133] came up with an efficient intermediate representation to overcome the limitations of [193]. This new representation explicitly represents the data members of the object. The parameter object is represented as a tree. The object forms the root of the tree, where the data members are shown as the child nodes. The edges between the object and it's data members denote the data dependences. The child node is a subtree, if a data member of the object is another object. A new form of slicing based on the objects is also introduced called *object slicing*. This new slicing approach enables the user to analyze the effect of a particular object on the slicing criterion, and provides better support for program understanding for large scale programs. The shortcomings of their method are that: i. This approach could be expensive as to obtain the object slice, one has to obtain the full slice of the program. ii. It is also computationally expensive to maintain the context of method calls especially when an object's method invokes other methods or is invoked by other methods.

Krishnaswamy [119] augmented the SDG with some more dependences relevant to object-oriented programs. But, these dependences do not completely cover a true object-oriented program such as a Java program. In our proposed intermediate graph, we have added some new dependences applicable to Java programs, such as *package dependence*, *type dependence* and *read/write dependence*. This is a suitable representation for a true object-oriented program like Java.

Chen et al. [42] proposed two types of program slices based on the dependences of object-oriented features, *state* and *behavior* slices. A *state slice* for an object comprises of those set of messages and control statements that possibly affect the state of the object. A set of attributes and methods that are defined in related classes and concern the behavior of the object, forms the *behavior slice*.

A good survey of the slicing techniques is available in [154, 182]. Although, the slicing techniques in [42, 119, 123, 133, 154, 182, 193] represent many features of object-oriented programs, still there exists some limitations with these approaches. The first issue is with the scalability of the graphical representation of larger programs. This is because for a large program the SDG (to represent all the methods and attributes and other O-O features) will be too large to manage and comprehend. The second limitation is associated with the granularity of the existing techniques. These techniques only slice the statements within the methods of a class. Therefore slicing of statements is not sufficient to analyze and understand classes. So, there was a necessity to increase the efficiency of the slicing techniques.

3.1.2 Slicing of Java Programs

Many researchers [11, 44, 85, 116, 130, 200] have proposed different approaches to compute slices of Java programs. Some of the slicing mechanisms are based on the dependence graphs like PDG and SDG, while other approaches are based on the Java byte-code analysis. To overcome the limitations and increase the efficiency of O-O slicing techniques, Kovacs et al. [116] proposed a static inter-procedural slicing of Java programs. This approach focuses on representing special Java features for improving the efficiency of the slicing technique. The proposed slicing approach can handle static variables, multiple packages, and interfaces. They also enhanced the SDG of the program by incorporating the polymorphic calls without requiring extra nodes for the purpose.

Chen et al. [44] proposed a new approach for graphically representing the O-O Java software. The authors discussed different dependences possible in a Java program and proposed slicing of classes based on program dependence graph (PDG). In their method, the program dependence graph consists of a set of independent PDGs. In slicing of classes, the slicing criterion taken is $\langle s, v, class \rangle$, where s is the statement number, v is the variable and $class$ is the name of the class to be sliced. The slice is computed by traversing backward from s and marking all the statements and data members used in the class based on the PDG. Based on this new model of program representation Chen et al. [44] also introduced the concepts of *partial slicing*, *object slicing*, and *class slicing*.

Allen et al. [11] extended the work of Chen et al. [44] on program slicing by using SDG. In their work, they proposed slicing of programs in the presence of exceptions. The focus was mainly to determine the control and data dependences due to the presence of try, catch and throw blocks in the program. They have not considered other Java specific features (such as interface, super, polymorphic calls, and template classes) for slicing. But, in our approach, we have considered the O-O features like packages, super, method overriding, etc. for the purpose of slicing.

Wang et al. [200] proposed a technique for slicing of Java programs by using compressed byte-code traces. They represented the byte-code corresponding to an execution trace of a Java program. Then, through backward traversal of the execution trace, they determined the control and data dependences on the slicing criterion. This approach requires the trace table to be constructed for each method. If a program will have too many methods, then this approach will be disadvantageous to compute the slices. This is because of the increased execution overhead in maintaining the execution trace tables. This work is also silent regarding the execution trace of the methods that are nested, overloaded and/or overridden.

Similarly, Hammer et al. [85] proposed a method for slicing of Java programs in the presence of objects as parameters. The analysis of the dependences is based upon an *Intermediate Representation (IR)* generated from the byte-code of the program. A good point-to analysis is a prerequisite of this algorithm to compute more precise slices.

Li et al. [130] proposed a hierarchical slicing technique to slice Java programs. The slicing algorithm is implemented on a tool, named *JATO*. Hierarchical slicing is computed level wise, starting from the package level to statement level. Li et al. [130] simplified the program representation by introducing a level-wise graphical representation of object-oriented programs at different levels of program organization such as package level, class level, method level and statement level. Four different graphs are constructed one at each level, these are *Package Level Dependence Graph (PLDG)*, *Class Level Dependence Graph (CLDG)*, *Method Level Dependence Graph (MLDG)*, *System Level Dependence Graph (SLDG)*. This hierarchical slicing approach requires four different slicing criteria to be set, one at each hierarchical level. However, the traversal of the graph from package level to statement level resulted in imprecise slices.

Slicing of Java programs in all of the above work [11, 44, 85, 116, 200] was proposed by taking into consideration a specific feature or type of dependence present in a Java program. Whereas, the overall impact of the features on the dependences such as the dependence due to the presence of packages and other specific Java features are not considered. Our approach has made a decent effort in analyzing all the possible dependences in O-O programs and computing a more accurate slice. To be able to employ slicing for regression testing, we need to identify all those statements that affect the modified statement and those statements that may get affected by the modification. But, most of the existing approaches [11, 44, 85, 200] are based upon either forward traversing or backward traversing. This will only result in the partial identification of the affected statements due to the modification. But, our approach is better suited for regression testing due to the following reason: both forward and backward traversals of our approach are suitable for an efficient change impact analysis, as they correctly find those program parts that may be affected or may affect other program parts due to the change.

3.2 Regression Testing

An informal meaning to the word regress is to return back to an existing previous state. Regression testing is the process to ensure that a program has not regressed back to the faulty state after the changes are made to it. Regression testing can be either progressive or corrective according to Leung and White [127]. Progressive regression testing involves retesting of the major changes that are made to the specification of a program. Whereas corrective regression testing performs retesting only on minor modifications that do not affect the overall program structure. To ensure that the functionality in the new version of the program works correctly after its modification, Gupta et al. [81] proposed a program slicing based regression testing technique. This approach relies on solving the data flow equations to explicitly detect def-use associations that are affected by a program change. The algorithm first makes a backward traversal and then makes a forward traversal from the point of change to find the affected def-use associations. This slicing technique relies neither on the data flow history nor on re-computation of data flow for the entire program to detect the affected definition-use associations. The changes made to the program initiate the necessary partial data flow re-computation through slicing. This approach achieves the same testing coverage as achieved by retesting with all the test cases without maintaining a test suite. Thus, it eliminates the overhead of maintaining a test suite.

Leung and White [127] classified the initial test cases into different categories such as *reusable*, *retestable*, *obsolete*, and adding *new-structural* and *new-specification* test cases. Then they suggested to select the test cases from any one of the test cases or from all the categories. Harrold and Sofa [94] have provided a strategy for unit and integration regression testing by combining the data flow testing with the incremental data flow analysis. Harrold et al. [91] gave a methodology to select a minimal number of test cases that ensure the correctness of all the requirements of a module. A safe algorithm based on the dependence graph of a module is given by Rothermel et al. [173]. This approach selects those test cases for regression testing that results in a different output than the original output. Binkley [23] gave a semantic differencing based approach to reduce the cost of regression testing. An integer programming problem formulation was proposed by Fischer [71], which was extended in [95] and solved by using natural optimization in [145] for optimal retesting. Some researchers [145, 146] have used evolutionary algorithms like genetic algorithms to solve the problem of regression testing. Some software tools for regression testing based on the above mentioned approaches are available in [43, 205, 206]. According to Rothermel et al. [176], complete regression testing of a software of 20,000 lines of code require around seven weeks of continuous execution. This necessitates development of many techniques to enhance the efficiency of regression testing (selection, minimization, and prioritization). Some existing work on these regression testing techniques are discussed in the next subsequent sections.

3.2.1 Test Case Selection

Harrold et al. [90] proposed traversal algorithms to identify the dangerous edges for safe regression test selection. The dangerous edge is defined to be an edge e such that for each input i causing P to cover e, $P(i)$ and $P'(i)$ may behave differently due to differences between P and P', where P and P' are the programs under consideration and the modified program respectively. The dangerous edge is identified by traversing the proposed *Java Interclass Graph (JIG)*. This method compared two nodes of P and P' in the JIG to identify the execution path of a test case in P and P', so that it can be known whether any edge is dangerous or not. This technique ensures that any test case that does not cover the dangerous entity will behave in the same way in both P and P'. Thus, it cannot expose new faults in P'. So, it is safe to select only those test cases for which the dangerous entity is covered.

Li et al. [130], used hierarchical slicing for regression test case selection of object-oriented programs. They proposed a model consisting of three levels: syntax analysis, generation of dependence graphs, and computation of slices. They proposed different dependence graphs such as *package level dependence graph (PLDG)*, c*lass level dependence graph (CLDG)*, *method level dependence graph (MLDG)* and *statement level dependence graph (SLDG)* which were based on the slicing criteria. When any modification is done to a statement, the dependence of that statement with its method, class and package can be easily detected because of the different levels of graphs maintained. Identification of other packages, classes, methods and statements related to the modified statement can also be easily done. The overall performance had improved as the irrelevant packages, classes, methods and statements were discarded from the generated graph. But, the proposed method required all the different graphs (PLDG, CLDG, MLDG, SLDG) to be generated for each change done to the program and was not very advantageous in case of frequent changes. Thus, to avoid the above mentioned problem, the slicing criterion was fixed. Whereas, we have implemented the hierarchical slicing technique on the rEOSDGJ which is not constrained to any fixed change. It rather works for any number of changes done to any statement, without requiring us to maintain additional graphs. So the space requirement of our proposed approach is much less than that of Li et al. [130]. If the change made to the example program triggers some new changes to be made, then our approach is capable of handling it.

Tao et al. [188] applied hierarchical slicing for regression testing of object-oriented programs. In their approach, they had also proposed to maintain separate graphs for packages, classes, methods and statements even if they were not affected by the change. This again required more space requirement. This is because with the increase in the program complexity, there will be an increase in the number of packages, classes, methods and statements which are required to be represented as separate graphs. But, in our approach, we only maintain the graph EOOSDG. This does not impose any additional space requirement. In some work [66, 104, 175] only control dependence and data dependence are considered. But, we identified some more dependences such as *package membership dependence*, *type dependence* and *read/write dependence*, which represented various object relations so as to consider more features of Java programs and computed the slices more accurately. Therefore, appropriate test cases are selected more accurately for regression testing, in our approach.

Panigrahi and Mall [164] proposed a technique to reduce the size of the test suite by selecting the test cases using an improved precision slices. The approach requires the construction of the control flow graph model of an object-oriented program and detects the infeasible paths. Then, the authors computed the def-use pairs only for those paths that are feasible. A dependence model was then created using the computed information that helped ignore the dependences across infeasible paths and lead to computation of precise slices. The affected nodes were discovered by performing a forward slice on the dependence model. This approach selected those test cases that covered these affected nodes. The limitation of this approach is that it is not always feasible to compute the control flow graphs for large object-oriented programs in the presence of features like inheritance, polymorphism, dynamic dispatch, etc.

3.2.2 Test Suite Minimization

The work in [58] motivates the proposed work presented in Chapter 7 of this thesis to use integer linear programming for test suite minimization. Minimization techniques focus on selecting the minimum number of test cases that satisfy a given criterion.

Li et al. [58] have focused on minimizing the in situ test suite based on their level of energy consumption. It selects those test cases in less than 1 second that consume 95% less energy and maintains the coverage of testing requirements. However, this time for minimization does not include the time required to pre-compute the test data (such as energy consumption and coverage information).

Yoo and Harman [210] provided an elaborate, recent study on the available techniques for test suite minimization, selection, and prioritization. Like our proposed approach, the techniques discussed in [210] are designed for regression testing.

Rothermel et al. [174] and Wong et al. [208] carried out an empirical study to investigate the limitations of single criterion minimization techniques. They performed experiments to minimize the test suite on the basis of fault detection capability. The results concluded that single criterion-based minimization techniques detected fewer faults as compared to the original test suite considered.

Jeffrey and Gupta [105] addressed the limitations of single-criterion minimization techniques by considering multiple sets of testing requirements (e.g. coverage of different entities). The results in [105] had shown an improvement over the existing techniques. In our approach, we also considered similar coverage criterion.

Black et al. [28] proposed a two-criteria variant of test suite minimization and computed optimal result using an integer linear programming solver. The authors focused on minimization based on definition-use association coverage and the ability of test cases to reveal errors. The results have shown that the error revealing ability of the test cases measured with respect to a collection of program faults helped in revealing other program faults.

3.2.3 Test Case Prioritization

Software maintenance being the most important and expensive activity in the process of Software Development Life Cycle (SDLC), many researchers have proposed several approaches for ordering the test cases of procedural programs [104, 132, 142, 175]. All the existing techniques on prioritization focus on procedural programs. Some slicing based techniques [104, 175] also exist to prioritize the test cases for procedural programs.

Rothermel [142, 175] and Elbaum [66] have considered different types of program coverage criteria such as total statement coverage, additional statement coverage, total function coverage etc. for prioritizing the test cases. Jeffrey and Gupta [104], proposed a method for prioritizing the test cases for regression testing based on the coverage of relevant slice of the output of a test case. They assigned some weights to the test cases to determine their priority. They determined the weight by summing up the number of statements present in the relevant slice and number of statements exercised by the test case.

Korel et al. [113, 114, 115] prioritized the regression test suite by considering the state model of the system. Whenever, the source code was modified, the corresponding change in its state model was identified. These modified transitions along with the run-time information were used to prioritize the test cases. Their initial approach was called selective prioritization, which was strongly connected to regression test selection (RTS) [115]. Test cases were classified into a high priority set, *TSH*, and a low priority set, *TSL*. They defined and compared different definitions of high and low priority test cases, but essentially a test case is assigned high priority if it is relevant to the modification made to the model. The initial selective prioritization process consists of the random prioritization of *TSH* followed by the random prioritization of *TSL*. Korel et al. [113, 114] developed more sophisticated heuristics for prioritization based on the dependence analysis of the models.

The use of mutation score for test case prioritization has been analyzed by Rothermel et al. [176, 177] along with other structural coverage criteria. Hou et al. [103] considered interface contract mutation for the regression testing of component-based software and evaluated with the additional prioritization techniques.

However, the available techniques were of little help when they were applied to regression testing of *object-oriented programs*. The detail survey conducted by different researchers on available coverage based prioritization techniques [60, 210, 214] reveals that, these techniques are not suitable for object-oriented programs. This is because of the presence of different other types of dependences that are inherent to object-oriented programs, other than data and control dependences. Panigrahi and Mall proposed a version specific prioritization technique [162] to prioritize the test cases for an object-oriented program. Their technique prioritizes the selected regression test cases. The test cases are prioritized based on the coverage of affected nodes of an intermediate graph model of the program under consideration. The affected nodes are determined based on the dependences arising on account of the object relations in addition to the data and control dependences. The effectiveness of their approach is shown in the form of improved APFD measure achieved for the test cases.

In another work, Panigrahi et al. [163] have improved their earlier work [162] by achieving a better APFD value. In this technique, the affected nodes are initially assigned a weight of 1. The weight is decreased by 0.5, whenever the affected node is covered by previous execution of the test cases. The limitation in both the approaches [162, 163] is that they have assumed that all the test cases have equal cost, and all faults have same severity. They have also assumed that all the affected nodes have a uniform distribution of faults. As a result, a test case executing more number of affected nodes will detect more faults and therefore, has a higher priority.

3.3 Change Impact Analysis (CIA)

To the best of our knowledge, no work has been done on quantification of the impact of crosscutting changes. In the absence of any work that can be directly compared with our work, we discuss some of the existing work on change impact analysis that closely relate to our work. A detail survey of the different change impact analysis techniques is available in [126, 129]. The approach of CIA by Sun et al. [186] is based on identifying a hierarchical set of changes at different granularity levels. The impact of the change was computed using hierarchical slicing proposed in [130]. One of the major limitation of this approach is that the hierarchical slicing approach used for change impact analysis does not yield in precise slices. Kung et al. [121] automated the change identification process and assessed the impact of these changes in object-oriented programs. This approach categorizes the identified potential changes into different granularity levels such as data, method, class, and class library changes. Identification of the changes is only one aspect, but this paper lacks inclusion of metrics to facilitate and quantify the maintenance work.

Rajlich [168] proposed a change propagation model for software maintenance activity. This propagation model was based on graph rewriting that analyzed the dependences between the changes. A prototype tool, named *Ripples 2*, implemented two basic processes of change propagation through *change-and-fix* and *top-down* propagation. The major limitation of this approach is that the dependence analysis assumed that there were no incoming inconsistencies after a change was made. Therefore, the analysis was made only for outgoing dependences.

Briand et al. [33] investigated the impact of the change in a commercial C++ system based on the coupling dimensions of the classes. They found a significant correlation between the coupling dimensions of the classes with the ripple effects of the changes. The authors used this coupling dimension to rank the classes according to their probability of containing the ripple effects. The proposed coupling measure does not work well for Java programs with their added complex features. Coupling factor alone does not signify well the level of error scattering and tangling in the modules.

The change impact analysis by Ryder et al. [178] determined the affected test drivers. Thus, it identified the test cases that failed or passed due to the set of changes. This paper does not analyze the dependence among the changes. It lays equal focus on every change made to a program. A performance analysis of the proposed approach is not included in the work.

Tonella [192] carried out impact analysis using the concept lattice of decomposition slices. The decomposition slice graph represents the dependences that exist between the computations performed on different variables. The concept lattice groups the computations that share the common variables and arranges the groups into a hierarchy of concepts. The main contribution in [192] is the graphical representation, called lattice of decomposition slices, to support software maintenance. The graph provides the relevant information regarding the computations and a data structure to conduct impact analysis. The major drawback of this approach is that it only works at intra-procedural level.

Badri et al. [17] proposed a call graph based predictive change impact analysis. It generated the different control flow paths in a program that were then used to identify the components affected by the change. The reported technique supported the prediction of impact sets and regression testing. This technique also lacked any provision for quantifying the impact of changes and focused on testing all the changes with same priority.

Ren et al. [169] identified the causes of failure of Java programs through CIA. The reported approach used the results from a CIA tool, named Chianti, to build a compilable intermediate version of the program. This intermediate version of the program is re-executed with the tests for specific changes to locate the exact reasons for failure. This paper only focuses on using CIA for debugging. This paper is also silent about the scalability and overhead of generating the intermediate version of the program.

The CIA proposed by Sheriff et al. [180] was based on singular value decomposition. The proposed approach was based on the collection of historical change records that may not be correct. And if these records are not available then the approach will fail to locate the effects. Hattori et al. [96] have proposed an approach to measure the precision and accuracy of the impact analysis techniques. The authors have defined the concept of false positives and false negatives in the context of fault analysis. This paper only focuses on precision and recall attributes of the studied techniques and is silent on the scattering and tangling of the impact of the changes.

German et al. [76] proposed a *change impact graph (CIG)* to visualize the impacts of changes. The unaffected nodes were removed from the graph. This work heavily depends on the quality of the graph constructed from the original program. The more accurate is the graph in representing the dependences among the program statement, the better will be the change impact analysis. This approach is not yet extended to work for object-oriented programs. The approach by Gethers et al. [77] estimated the impact set by analyzing the change request, source code, and semantic indexing. But, this approach suffers from a limitation that if the change request is inaccurate and inefficient, it may result in erroneous omission of some methods during filtration. Some more work on change impact analysis is available in [1].

3.4 Summary

In this chapter, we briefly reviewed some work on program slicing, slicing of object-oriented programs, and slicing of Java programs that are relevant to our research. We discussed the work on regression testing techniques such as test case selection, test suite minimization, and test case prioritization. In the absence of any literature on change impact quantification, we discussed the relevant work on change impact analysis of object-oriented programs. Our literature survey observation shows that the existing slicing algorithms have exponential or unbounded space complexity and time complexity with respect to the number of statements in the program. As slicing algorithms are used in interactive applications such as maintenance and testing, it is essential to increase its efficiency and search for new avenues of its application. To meet the goals of this thesis, we attempt to develop in the subsequent chapters a suitable intermediate representation for Java programs and an efficient slicing algorithm for the purpose of change impact analysis and efficient regression testing.

Chapter 4

Regression Test Case Selection using Slicing

The change in the user requirements and growing expectations of the customers have forced the software to evolve at regular intervals of time. As the complexity of the software increases, the cost and effort to maintain such complex software also increases. After making changes to the software, regression testing should be carried out in order to assure the validity of the modified part and to ensure that the changes do not affect other parts of the program. Therefore, regression testing has become an integral part of the software maintenance process [109]. It is indispensable to make changes and modifications to an already tested program. Thus, the role of regression testing has become apparent in retesting the program. Retesting is based on the modifications done without compromising with the time and cost of retesting, while maintaining the same testing coverage [109, 157].

Program slicing is an effective and efficient technique to debug, test, analyze, understand and maintain software [54, 73, 182, 203]. However, while applying the same techniques to OO programs, we fail because of the presence of many other dependences originating from the OO features. Although OO features have improved program understandability and readability, but have complicated the maintenance activities [179]. The dependences that arise due to the class and object concepts are inheritance dependence, message dependence, data dependence, type dependence, reference dependence, concurrency dependence, etc.

The existing slicing techniques based on system dependence graphs in [119, 123, 154] have considered C++ programs which are *partially* OO [34] in nature. That's why we are motivated to consider Java programs for our work which is a widely used true OO programming language. The existing slicing techniques cannot be applied to Java programs because of the presence of many new features. The presence of the features like *packages*, *super*, *dynamic method dispatch*, *interface*, *exception handling*, *multi-threading*, etc, in Java add to the list of dependences and thus make the maintenance even more difficult. Their effects on the maintenance of the programs need to be considered separately. In Java, all the classes and their methods are grouped into packages. Suppose a method M1 of class C1 belonging to a package P1 wants to invoke a method M2 of class C2 that belongs to another package P2. This can be achieved by importing the package P2 in package P1 and by instantiating the class C2 in C1. This will create a dependence among the packages P1 and P2, classes C1 and C2, methods M1 and M2 and among the statements in both the methods. Apart from this, there are many methods which are dependent on the type of data they are operating upon. For each type of data, there is a different method.

Therefore, owing to the presence of such new dependences, separate analysis of their impact on regression testing is essential. As a result, existing techniques of slicing based on the System Dependence Graph (SDG) of Java programs, does not seem to be feasible for regression testing. Incremental regression testing is a probable solution which is based on the following simple observations: (1) if a statement is not executed under a test case, it cannot affect the program output for that test case. (2) not all statements in the program are executed under all test cases. (3) even if a statement is executed under a test case, it does not necessarily affect the program output for that test case. (4) every statement does not necessarily affect every part of the program output. We can apply the above assumptions to Java programs at different levels such as packages, classes, methods and statements.

Keeping in view the above motivations and to overcome the challenges, we fix our goal in this chapter as follows:

1. To construct a suitable intermediate graph for representing different dependences in Java programs arising due to the different object-oriented features. This intermediate graph is formally defined in Section 4.2.1.

2. To remove the redundant dependences, if any, in EOOSDG to improve the scalability of the intermediate graph and save slice computation time. The detail process of removing the redundant edges is discussed in Section 4.2.2.

3. To develop and implement a hierarchical graph reachability slicing algorithm that identifies different program parts affected by the changes made to the

program. This slicing algorithm is discussed in Section 4.2.3.

4. To map the coverage information of the test cases with the affected program parts to select the test cases that are relevant for regression testing of the program under consideration. The different coverage information maintained for the test cases are discussed in Section 4.3.

The rest of the chapter is organized as follows: Section 4.1 gives a background of hierarchical program slicing and other related issues. In Section 4.2, we discuss our proposed work on hierarchical regression test selection which is based on the Extended Object-Oriented System Dependence Graph. In this section, we present our hierarchical decomposition slicing approach for finding the program parts affected by the changes made to programs. We give the correctness proof of our proposed algorithms and compute their space and time complexities. along with a working example. In Section 4.3, we discuss the implementation of our work. We state the sample programs that are taken for our experiments and describe our experimental setting. Further, snapshots of our implementation are taken for a better analysis of the result. In Section 4.4, we have compared our work with some of the related work. Section 4.5 summarizes the chapter.

4.1 Background

In this section, we discuss hierarchical slicing, which is required for understanding our approach on regression test case selection.

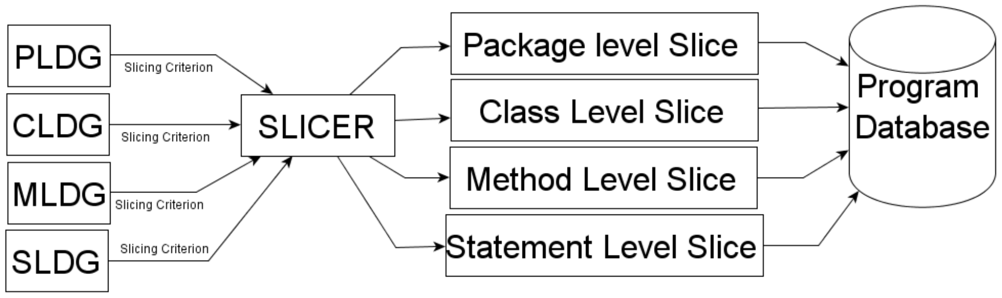


Figure 4.1: Model for Hierarchical Slicing.

Hierarchical Slicing

Instead of analyzing the data flow and control flow for an OO program as a whole, it is useful to employ the hierarchical structure of the OO programs (e.g. Java programs), to detect the impact of the changes made to the program. A Java program P, is composed of a set of packages, classes, methods and statements, organized in a hierarchical manner. Therefore, in *hierarchical slicing*, we first try to slice out the packages that might have been affected by the change. From the set of affected packages, we then slice out the affected classes. Then the affected methods and the statements inside those methods are sliced out for retesting. The above concept of *hierarchical slicing* [130] can be explained by considering a slicing criterion (i.e. the point of modification) $< s, v >$, where s is the statement containing variable v. Let $S(P)$ be the set of packages, classes, methods and statements of a program P. The model for hierarchical slicing is shown in Figure 4.1. The steps of hierarchical slicing are as follows:

Step-1 First, we detect the package p containing s and v and all other packages, based on their direct or indirect dependences on p caused due to import statements. All those packages which are not related to the package p are removed. By following this process, the package level slice obtained is marked as $S_1(P)$.

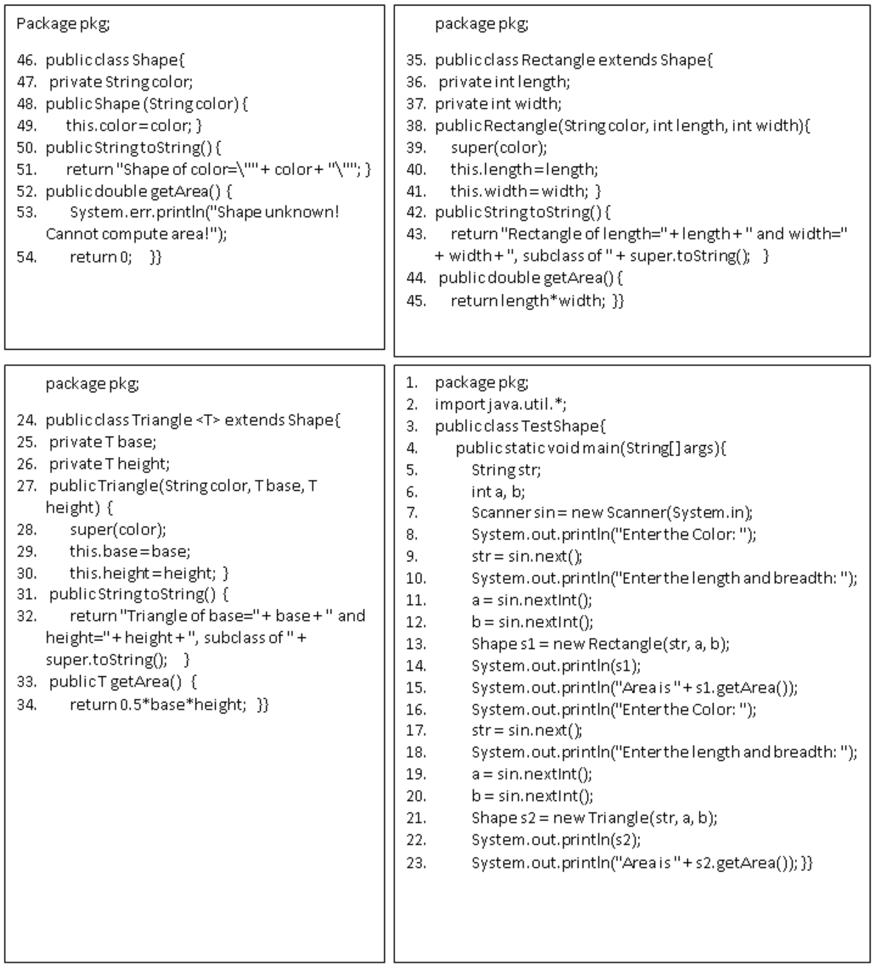
Step-2 Then, we analyze $S(P)$, to find out all those classes that are related to the class containing s and v. All other irrelevant classes are removed to get the class level slice. The class level slice is marked as $S_2(P)$.

Step-3 Next, we analyze $S(P)$ and delete all the member methods and variables that are not related to the method containing s and v. This results in the method level slice, which is marked as $S_3(P)$.

Step-4 Finally, to find out the statement level slice, we analyze $S(P)$ and delete all the statements and predicates that are not related to statement s containing variable v. The slice thus obtained is marked as $S_4(P)$.

This step wise extraction of the slices is known as *hierarchical slicing*. The test cases needed at each level can be related as $T(S_4(P)) \subseteq T(S_3(P))$, $T(S_3(P)) \subseteq T(S_2(P))$, $T(S_2(P)) \subseteq T(S_1(P))$. At each level, we obtain more accuracy in minimizing the required number of test cases from a higher level to a lower level by discarding the test cases that are not relevant to the affected program parts. The concept of hierarchical slicing is available in [130] and used for test case selection

[188] and change impact analysis [186] of OO Programs. We use this concept of hierarchical slicing for selecting our regression test cases. However, the technique of computing the hierarchical slice as given in [130] is modified in this work.



```
Package pkg;

46. public class Shape{
47.  private String color;
48. public Shape (String color) {
49.     this.color=color; }
50. public String toString() {
51.     return "Shape of color=\"" + color + "\""; }
52. public double getArea() {
53.     System.err.println("Shape unknown!
    Cannot compute area!");
54.     return 0;   }}
```

```
    package pkg;

35. public class Rectangle extends Shape{
36.  private int length;
37. private int width;
38. public Rectangle(String color, int length, int width){
39.    super(color);
40.    this.length=length;
41.    this.width=width; }
42. public String toString() {
43.    return "Rectangle of length=" + length + " and width="
    + width + ", subclass of " + super.toString();  }
44.  public double getArea() {
45.    return length*width; }}
```

```
    package pkg;

24. public class Triangle <T> extends Shape{
25.  private T base;
26.  private T height;
27.  public Triangle(String color, T base, T
    height) {
28.     super(color);
29.     this.base=base;
30.     this.height=height; }
31.  public String toString() {
32.     return "Triangle of base=" + base + " and
    height=" + height + ", subclass of " +
    super.toString();   }
33.  public T getArea()  {
34.     return 0.5*base*height;  }}
```

```
1.  package pkg;
2.  import java.util.*;
3.  public class TestShape{
4.      public static void main(String[] args){
5.          String str;
6.          int a, b;
7.          Scanner sin = new Scanner(System.in);
8.          System.out.println("Enter the Color: ");
9.          str = sin.next();
10.         System.out.println("Enter the length and breadth: ");
11.         a = sin.nextInt();
12.         b = sin.nextInt();
13.         Shape s1 = new Rectangle(str, a, b);
14.         System.out.println(s1);
15.         System.out.println("Area is " + s1.getArea());
16.         System.out.println("Enter the Color: ");
17.         str = sin.next();
18.         System.out.println("Enter the length and breadth: ");
19.         a = sin.nextInt();
20.         b = sin.nextInt();
21.         Shape s2 = new Triangle(str, a, b);
22.         System.out.println(s2);
23.         System.out.println("Area is " + s2.getArea()); }}
```

Figure 4.2: An example Java program

4.2 Hierarchical Regression Test Selection

In this section, we discuss our proposed work on hierarchical regression test selection. The discussion on the proposed approach comprises of three phases: i) construction of the intermediate program representation, ii) computation of the slices, iii) selection of the regression test cases. These three phases are broadly divided into the following subsections: Section 4.2.1 gives an idea of the different dependences considered to construct the intermediate representation of the input program. In

Section 4.2.2, we discuss an algorithm to remove the redundant edges present in the proposed intermediate graph, to improve the scalability of this approach. In Section 4.2.3, we discuss the proposed algorithm for computing the slice and use the resultant slice for hierarchically selecting the test cases for regression testing. Our regression test case selection approach is given in Section 4.2.4. We discuss the working of the algorithm in Section 4.2.5. Section 4.2.6 shows the complexity analysis of the proposed algorithm.

Java being the most popular OO language, we are encouraged to consider Java programs. An example Java program is shown in Figure 4.2. We use this example program as our running example for explaining the working of our proposed approaches and for implementing the proposed approaches in the rest of this thesis. This program defines a class named *Shape* which is inherited by the classes *Rectangle* and *Triangle*. The class *TestShape* contains the *main* method and computes the area of a *rectangle* and *triangle*, based on the user inputs given through the console and displays the result. We propose an intermediate representation suitable for OO programs called *Extended Object-Oriented System Dependence Graph (EOOSDG)*. However, an investigation of intermediate graph representations of other OO programming languages (such as C#) is equally important and has been left as future work.

4.2.1 Proposed Intermediate Graph Representation:EOOSDG

An example Java program shown in Figure 4.2 is taken as a case study to discuss the various dependences identified in the EOOSDG. EOOSDG for the example Java program in Figure 4.2 is shown in Figure 4.3. In the construction of EOOSDG, we consider some additional dependences in Java, in addition to the dependences defined by Krishnaswamy [119] for OO programs. The proposed EOOSDG is a directed graph $G(N, E)$, where N is the set of nodes that correspond to different program parts such as statements, methods, parameters, classes, packages etc. E is the set of edges that represent the dependences present among different program parts. The semantic of the edge $u \rightarrow v \in E$ represents the dependence of node v on node u. Some of these dependences (edges) are identified and defined in [119]. We classify these dependences based on their role in representing some Object-Oriented features at different hierarchical levels.

Package level dependences

Package level dependence specifies the dependence of a package with its con-

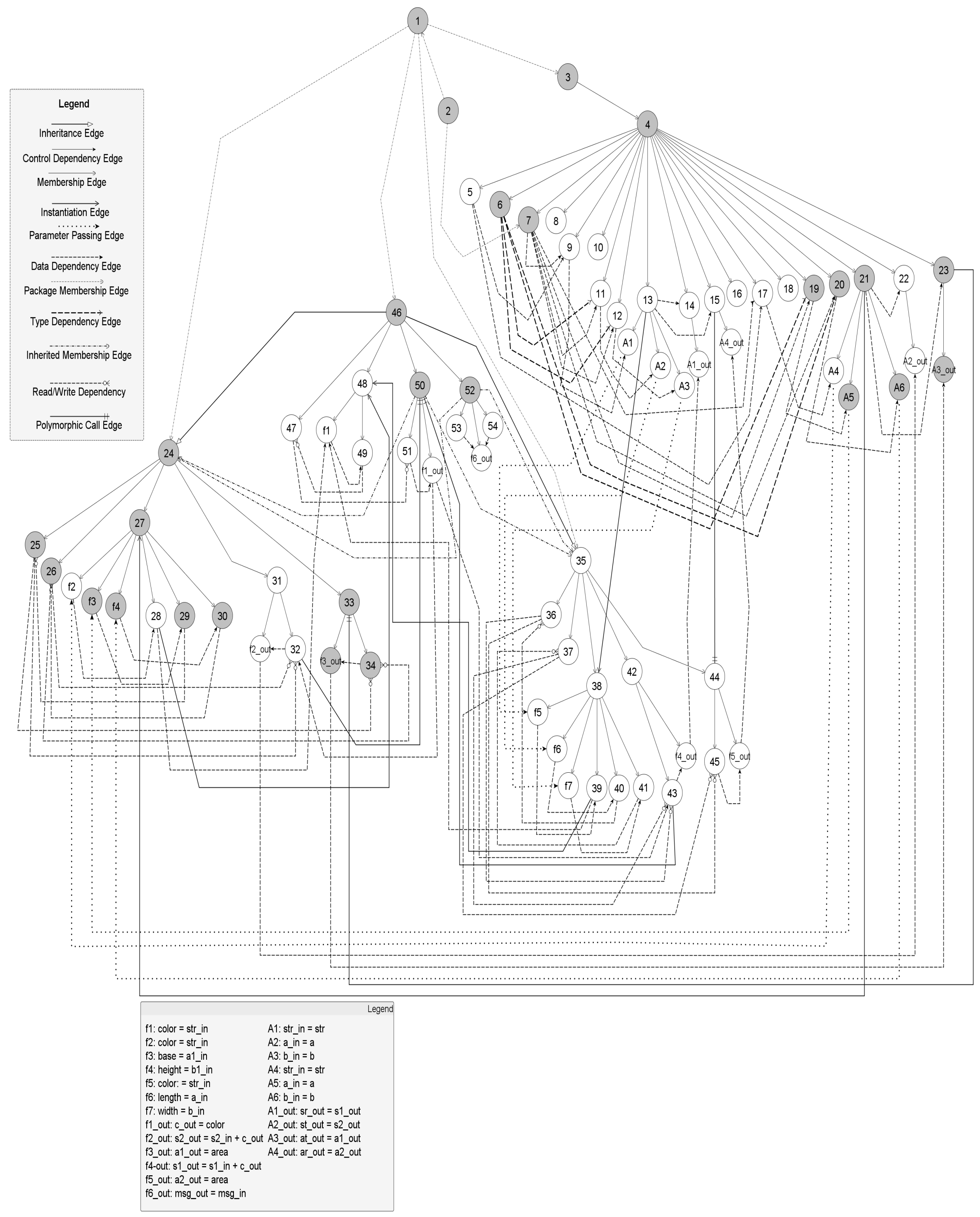


Figure 4.3: EOOSDG of the example program in Figure 4.2.

stituent classes and sub-packages through package membership edges.

i. *Package membership edge*: In Java, all the library classes and user defined classes belong to some package. We have considered the packages as separate nodes in our proposed intermediate representation EOOSDG. The package dependence arises when one package imports some other packages into it so that some or all the classes in the imported package can be made accessible by instantiating those classes. This creates a dependence between the packages. Thus, an edge from the header of the importing package to the header of the imported package depicts this dependence. Some of the package dependence edges in Figure 4.3 are $(1, 24)$, $(1, 46)$, $(1, 3)$ etc.

Class level dependences
When a class uses the features of another class, then various types of dependences are formed between different parts of the participating classes. Here, we discuss some of the edges used to model such dependences that results from inheritance of classes.

ii. *Inheritance/Implementation edge*: Inheritance is an important feature of the object-oriented paradigm. It establishes the association between the base class and derived class, in the direction of hierarchy. This relationship between the classes is shown by Inheritance/Implementation edge. When a class implements an interface then also Inheritance/ Implementation edge connects the interface entry vertex with the class entry vertex to mark the implementation of the interface. Some of the Inheritance/Implementation edges in Figure 4.3 are $(46, 24)$, $(46, 35)$, etc.

iii. *Has edge*: If a class declares an instance of another class, it establishes a has a relationship between both the classes. Some *has edges* in Figure 4.3 are $(3, 24)$ and $(3, 35)$.

iv. *Membership edge*: Every method and attribute in the object-oriented paradigm are the members of a class and are accessible through the object of that class only. Thus, a membership edge connects the method/ attribute header and the class header of the class in which the method is defined [119]. Similarly, we extend this definition of membership edge to represent the relationship between method header and different parts of the method such as formal parameter-in and parameter-out, statements and function calls. The relationship between function call node and its actual parameter-in and parameter-out nodes is also represented by membership edge. Some of the membership edges in Figure 4.3 are $(4, 21)$, $(21, A4)$, $(21, A5)$, $(21, A6)$, $(24, 25)$, $(46, 48)$, etc.

v. *Inherited membership edge*: Each method and the data members of a class are inheritable if they are accessible by the instance of the derived class. Thus, an inherited method or a data member can be considered as an implied member of the derived class. The inherited membership edge connects the header of the derived class with the header of the method or data member. Some of the inherited membership edges in Figure 4.3 are $(50, 24)$, $(50, 35)$, etc.

Method level dependences

Message passing is an important feature in object-oriented programs realized through method invocation by the objects. When one method invokes another method, it passes messages in the form of parameters, giving rise to various dependences such as:

vi. *Call edge*: When a method invokes another method, this relationship between the methods is represented by the call edge. Thus, a call edge connects the call site statement with the method entry vertex of the callee method.

vii. *Parameter passing edge*: The parameter passing edge represents the data exchange taking place between the actual parameter and formal parameter vertices whenever a method is invoked. Thus, a parameter_in edge connects the actual parameter of the calling method with the formal parameter of the called method. Parameter_out edges represent the data flow of the return value between called and the caller method. Some of the parameter passing edges in Figure 4.3 are $(A4, f2)$, $(A5, f3)$, $(A6, f4)$, etc.

viii. *Instantiation edge*: Instantiation means creating the instance of a class by invoking the constructor of the class which initializes the object. Instantiation edge connects the instantiation statement with the constructor method entry vertex. Thus, it marks a special method invocation to initialize the objects. Some of the instantiation edges in Figure 4.3 are $(21, 27)$, $(13, 38)$ etc.

ix. *Method overridden edge*: Method overriding is an attribute resulting from inheritance feature. A method in the base class is said to be overridden if it is redefined in the derived class. An interfaces and abstract classes specify the methods that are overridden by the implementing classes. This relationship of the methods with its overridden methods are represented through method

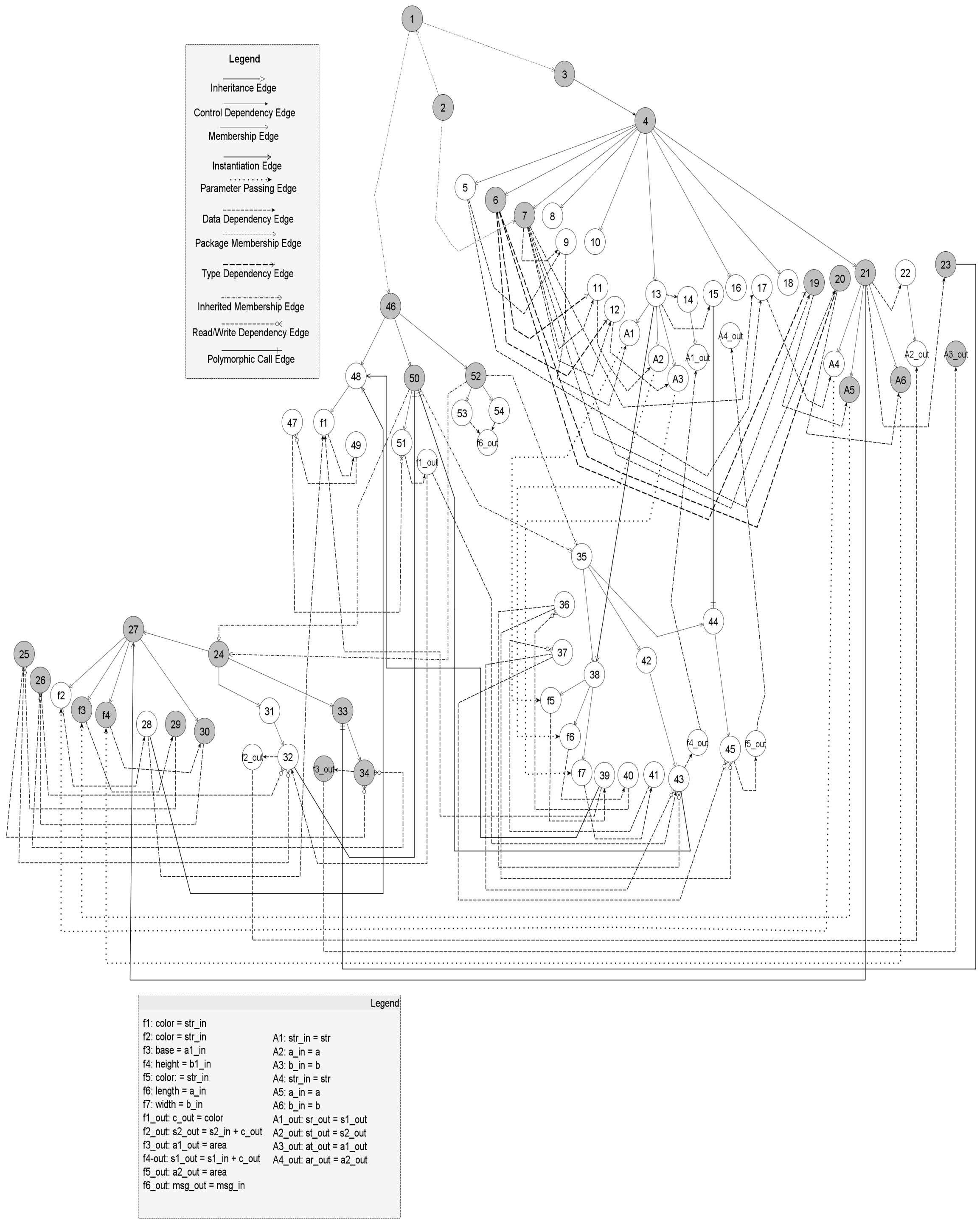


Figure 4.4: Reduced EOOSDG (rEOOSDG) of the example program in Figure 4.2.

overridden edges. Thus, the method overridden edge connects the header of the method in base class (interface) with the header of the method in the derived (implementing) class. Some of the method overridden edges in Figure 4.3 are $(52, 33)$, $(50, 31)$, etc.

Statement level dependences

Apart from the usual control and data dependences between statements in a program, we identify and discuss the representation of two other essential dependences present in object-oriented programs.

x. *Data dependence edge*: When data computed at one statement is used at another statement, an edge is marked to represent the flow of data from the site of computation to the site of usage. Some of the data dependence edges in Figure 4.3 are $(6, 19)$, $(7, 19)$ etc.

xi. *Control dependence edge*: When the execution of one statement is dependent on the execution of another statement, then the former is said to be control dependent on the later. The edge from one vertex to another depicts the control dependence between the vertices in the representation. As the example Java program in Figure 4.2 does not have any *if ... else* statements or *loop* statements, so its corresponding EOOSDG in Figure 4.3 does not contain any control dependence edge.

xii. *Type dependence edge*: In Java, there are several methods that depend upon the type of data. If the type of data is changed then the method also changes accordingly. Therefore, an edge from the data declaration statement to the statement containing a call to such a method, is essential to depict the type dependence. We name this edge as type dependence. Some of the type dependence edges in Figure 4.3 are $(6, 11)$, $(6, 12)$ etc.

xiii. *Read/Write dependence edge*: In an object-oriented language such as Java, encapsulation refers to the fact that a member method always reads/writes data from/to the data member of an object that invokes the member method. We represent such a relationship between member method and data member by read/write dependence edge. Some of the read/write dependence edges in Figure 4.3 are $(29, 25)$, $(30, 26)$, $(25, 34)$ etc.

Special dependences

There are certain situations that necessitates the representation of some other dependences among the program parts. For example, some method calls can only be resolved during execution of the program. In such a scenario, the static graphical representation should show all the possible methods that respond to a method call. Thus, such polymorphic dependences may result in context insensitive program comprehension, if not addressed. These dependences are as follows:

xiv. *Polymorphic call edge*: A polymorphic edge connects the call statement with the methods that are possible to be executed by the call after the binding is resolved at run-time. Thus, a polymorphic call edge connects the call site with both the possible executions. In case of dynamic polymorphism, Java interpreter dynamically resolves the choice of execution. Some of the polymorphic call edges in Figure 4.3 are $(15, 44)$, $(15, 52)$, $(23, 33)$, $(23, 52)$ etc.

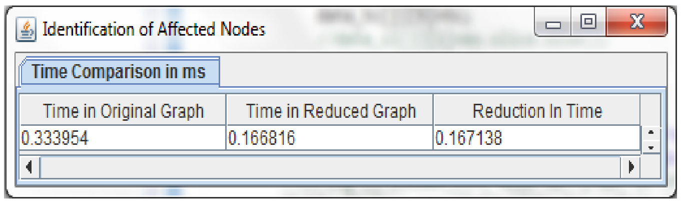
xv. *Summary edge*: Summary edge is added to represent the transitive dependence that exist between actual_in and actual_out vertices of the caller method. If there exists an inter-procedural path from the actual_in vertex to the actual_out vertex, then as summary edge is added to mark the transitive dependence between both the vertices. The summary edge protects the context sensitivity of the method call during the backward slicing of the program by restricting the entry into called method during backward traversal. Some of the summary edges in Figure 4.3 are $(A5, A21\_1\_out)$, $(A6, A21\_2\_out)$, etc.

Interfaces, Abstract Classes and Templates

Java interfaces are defined in the same way as classes, but with the keyword *interface*. An interface specifies the methods that are overridden by the implementing classes. A class implements an interface through the keyword *implements*. A derived class automatically implements the interface that its base class implements. One interface can also inherit from another interface through *extends* keyword in the same way as classes inherit. Thus, representation of interfaces and their dependences with other interfaces and classes in the intermediate graph of a program is essential. This aspect has been considered in the graphs proposed by Kovacs et al. [69] and Zhao [184]. EOOSDG treats interfaces and abstract classes as special types of class definitions, thus has same structural representation. However, unlike interfaces, abstract classes contain method implementation. Therefore, abstract methods are represented with method entry vertex. We add parameter_in and parameter_out vertices to correctly represent the method signature and return value, respectively.

An interface is represented with an interface entry vertex (same notation as class entry vertex). However, the abstract methods of the interface are connected with the interface enrty vertex by *abstract membership edges* (same notation as membership edges). The parameter_in and parameter_out vertices are connected through membership edges. The method entry vertex connects with the method entry vertex of the method implementing it by a *method overridden edge*. Similarly, the inheritance and implements relationships among interface and/or class vertices are shown by *inheritance/implementation edge*. If a class implementing an interface is inherited by another class, then the derived class also automatically implements the same interface as that of its base class. Such a relationship is implicit and is not explicitly represented in EOOSDG.

Templates are represented through generic edges. A generic edge corresponds to the dependence that arises due to the presence of a generic method in the program. The data type of the formal parameters in a generic method is known only at runtime when the actual arguments are passed to the generic method call. A generic edge depicts the data type information of the passed parameter. Thus, a generic edge connects the actual parameters of a method call and the formal parameters of the executed method. In case the return type of the method is of generic type, then a generic edge connects the formal_out node of the called method with the actual_out node of the caller method. If the data member of a class is of generic type then a generic edge connects the node accessing the data member with the data member itself. Thus, a generic edge is added between two nodes in addition to the preexisting dependences to mark the flow of generic information. Some of the generic edges in Figure 4.3 are $(f3\_out, A3\_out)$, $(f3, 25)$, $(f4, 26)$, etc.


Identification of Affected Nodes

Time Comparison in ms

Time in Original Graph	Time in Reduced Graph	Reduction In Time
0.333954	0.166816	0.167138

Figure 4.5: Time based comparison between EOOSDG and rEOOSDG for identifying the affected nodes with respect to some modification (slicing criterion).

4.2.2 Removal of Redundant Edges

In this section, we discuss an algorithm to remove the redundant edges present in our proposed intermediate graph EOOSDG. The proposed slicing technique (discussed in Section 4.2.3) is based on the reachability of nodes to identify the different affected program parts with respect to some modification made to the program. The reachability of a node v_1 in a given graph is defined as the set of nodes that can be reached by traversing (backward/forward) through the edges of v_1. Therefore, in a graph $G(N, E)$, where N is set of nodes and E is the set of edges, if $v_1,\ w,\ v_2 \in N$ and $(v_1, w),\ (w, v_2),\ (v_1, v_2) \in E$, then (v_1, v_2) is said to be redundant. This is because a path from v_1 to v_2 already exists through w, so another edge (v_1, v_2) is not needed. We observed the presence of such redundant edges in the EOOSDG shown in Figure 4.3. As each type of edge $e \in E$ exhibits a different kind of dependence among the program parts, it is likely that (v_1, v_2) is semantically different from (v_1, w) and (w, v_2). Since, the edge between a pair of nodes is used only for traversing, so removing a semantically different redundant edge does not affect the graph reachability slicing process. Algorithm 1, named *Redundant Edge Removal (RER)* algorithm, gives the pseudocode to remove the redundant edges (dependences) in EOOSDG. RER algorithm checks the redundancy of each edge of the input graph. If the edge is found redundant, then it is removed from the set.

Algorithm 1 *Algorithm RER*

```
Input: EOOSDG .
Output: rEOOSDG containing a non-redundant set of edges $E_r$.
 1: $E_r := E$;                                      ▷ Initialize $E_r$.
 2: $\forall\ u \to v\ \in\ E_r$ do
 3:     $G := E - \{u \to v\}$;
 4:     $S := \{u\}$;                                ▷ $S$ is a temporary set.
 5:     $\forall\ x \to y\ \in\ G$ do
 6:         If $x \subseteq S$ then
 7:             $S := S \cup \{y\}$;
 8:         End If
 9:     End For
10:     If $v \subseteq S$ then
11:         $E := E - (u, v)$;
12:     End If
13: End For
14: $E_r := E$;                                      ▷ $E_r$ is the set of redundant free edges.
15: End
```

The graph obtained after removing the redundant edges (dependences) is named *rEOOSDG*. The rEOOSDG of the graph given in Figure 4.3 is shown in Figure 4.4. After the removal of redundant edges, the representation of the graph becomes much simpler, and traversal becomes considerably faster. The time required to identify the affected nodes (with respect to the point of modification taken as the slicing criterion) is reduced approximately to 50% in rEOOSDG. This is shown in Figure

4.5. Figure 4.5 shows the reduction in time required to detect the affected nodes in the original graph (Figure 4.3) and in the reduced graph (Figure 4.4). Removal of the redundant edges (dependences) also makes the graph based slicing approach comparatively scalable without any loss of slicing information. This is because, we obtained the same slice in case of both EOOSDG and rEOOSDG. The computed slices for both EOOSDG and rEOOSDG are shown as shaded nodes in Figure 4.3 and Figure 4.4, respectively. However, in some applications this redundant edge removal process may result in loss of information and requires more investigation. The authors have deferred this investigation as a future work.

Correctness of RER Algorithm

Theorem 4.1. *RER algorithm works correctly and removes all the redundant edges from a given graph.*

Proof. Let graph G contains the following set of edges, E = { $a \rightarrow c$, $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow e$, $d \rightarrow e$, $b \rightarrow d$}. If edge $a \rightarrow c$ can be logically inferred from the set $E - \{a \rightarrow c\}$, i.e. $G \models a \rightarrow c, where\ G = E - \{a \rightarrow c\}$, then edge $a \rightarrow c$ is redundant, and hence removed from E. Let set S be initialized to a, i.e. $S = \{a\}$. Now the node a of edge $a \rightarrow c$ is a subset of S, i.e. $a \subset S$, so b is added to S, i.e. $S = \{a, b\}$. Repeating this for all the edges in E, we get $S = \{a, b, c, e, d\}$. Since, $c \subset S$ implies $S \models a \rightarrow c$, therefore $a \rightarrow c$ is redundant and hence removed from G. The *for* loop continues till all the edges in E have been checked for redundancy. Since, the number of edges in a graph is always finite, the execution of *for* loops in the algorithm exits after a finite number of iterations. Hence, the algorithm terminates and we get a graph with non-redundant set of edges. □

4.2.3 Proposed Hierarchical Decomposition (HD) Slicing Algorithm

In this section, we discuss our proposed algorithm named *Hierarchical Decomposition (HD) Slicing*, for finding those program parts that are affected by the change. We represent the input program in the form of EOOSDG as discussed in Section 4.2.1. The node that corresponds to the statement of modification is taken as the slicing criterion to compute the slices.
In this proposed work, we maintain the following sets of information:
P = { $p_1, p_2, \ldots, p_n$ } is the set of all the packages that are used in the given program.

Algorithm 2 *HDslice(G, n)*

Input: EOOSDG/rEOOSDG $G(N, E)$, total number of nodes $n = \mid N \mid$.

Output: Set of affected program parts.

```
Step 1 Initialize
 1:   $worklist\ Q_1 = \phi$
 2:   $worklist\ Q_2 = \phi$
 3:   $worklist\ Q_3 = \phi$
 4:   $worklist\ sp\_worklist = \phi$
 5:   $worklist\ Q = \phi$
 6:   $type_1 = \{polymorphic\ call\ edge, inherited\ membership\ edge, parameter\_out\ edge, generic\_out\ edge$
    $\}$
 7:   $type_2 = \{parameter\_in\ edge, generic\_in\ edge\}$
Step 2 Forward Traversal
 8: $enque(Q_1, current\_node)$        ▷ $current\_node$ contains some node of EOOSDG/rEOOSDG that
    corresponds to some modification done at statement $s$ in program $P$ taken as slicing criterion.
 9: while $Q_1 \neq \phi$
10:    $v = deque(Q_1)$
11:    $if\ v\ is\ not\ marked$
12:       $mark\ v$
13:    $enque(Q_2, v)$
14:    $for\ each\ unmarked\ w \in adj(v)$
15:       $mark\ w$
16:       $enque(Q_1, w)$
Step 3 Backward Traversal: *Pass 1*
17: while $Q_2 \neq \phi$
18:    $v = deque(Q_2)$
19:    $for\ each\ unmarked\ w \in adj(v)$
20:       $if\ (w, v) \notin type_1$
21:          $mark\ w$
22:          $type_2 = type_2 \cup \{(w, v)\}$
23:          $enque(sp\_worklist, w)$
Step 4 Backward Traversal: *Pass 2*
24: while $sp\_worklist \neq \phi$
25:    $v = deque(sp\_worklist)$
26:    $if\ v\ is\ not\ marked$
27:       $mark\ v$
28:    $for\ each\ unmarked\ w \in adj(v)$
29:       $if\ (w, v) \notin type_2$
30:          $mark\ w$
31:          $enque(Q_3, w)$
Step 5 Compute Final Slice
32:    $Q = Q_1 \cup Q_2 \cup Q_3$
Step 6 Compute Hierarchical Decomposition (HD) Slice
33: Find  $P_1 = P \cap Q$   ▷ where $P$ is the set of packages in the program and $P_1$ is the set of affected
    packages.
34: Update $Q = Q - P_1$                    ▷ now $Q$ contains classes, methods and statements.
35: Find $C_1 = C \cap Q$       ▷ where $C$ is the set of classes in the program and $C_1$ is the affected classes.
36: Update $Q = Q - C_1$                    ▷ now $Q$ contains only the methods and statements.
37: Find $M_1 = M \cap Q$  ▷ where $M$ is the set of methods in the program and $M_1$ is the affected methods.
38: Update $Q = Q - M_1$                          ▷ now $Q$ contains only affected statements.
39: Set $S_1 = Q$                                 ▷ where $S_1$ is the set of affected statements.
40: Exit
```

Table 4.1: Test case coverage distribution for the example program in Figure 4.2.

$TestCases$	$Packages$	$Classes$	$Methods$	$Statements\,(node\ nos.)$
$T1-T5$	$node1 : pkg$	$node24 : Triangle$	$node31 : toString\,()$	32
			$node27 : Triangle\,()$	$25, 26, 28, 29, 30$
		$node46 : Shape$	$node50 : toString\,()$	51
			$node48 : Shape\,()$	$47, 49$
		$node3 : TestShape$	$node4 : Main\,()$	$5, 6, 7, 16, 18, 19, 20, 21, 22$
$T6-T10$	$node1 : pkg$	$node24 : Triangle$	$node33 : getArea\,()$	34
			$node27 : Triangle\,()$	$25, 26, 28, 29, 30$
		$node46 : Shape$	$node52 : getArea\,()$	$53, 54$
			$node48 : Shape\,()$	$47, 49$
		$node3 : TestShape$	$node4 : Main\,()$	$5, 6, 7, 16, 18, 19, 20, 21, 23$
$T11-T15$	$node1 : pkg$	$node35 : Rectangle$	$node42 : toString\,()$	43
			$node38 : Rectangle\,()$	$36, 37, 39, 40, 41$
		$node46 : Shape$	$node50 : toString\,()$	51
			$node48 : Shape\,()$	$47, 49$
		$node3 : TestShape$	$node4 : Main\,()$	$5, 6, 7, 8, 9, 10, 11, 12, 13, 14$
$T16-T20$	$node1 : pkg$	$node35 : Rectangle$	$node44 : getArea\,()$	45
			$node38 : Rectangle\,()$	$36, 37, 39, 40, 41$
		$node46 : Shape$	$node52 : getArea\,()$	$53, 54$
			$node48 : Shape\,()$	$47, 49$
		$node3 : TestShape$	$node4 : Main\,()$	$5, 6, 7, 8, 9, 10, 11, 12, 13, 15$

C = $\{ c_1, c_2, \ldots, c_n \}$ is the set of all the classes defined in the program.
M = $\{ m_1, m_2, \ldots, m_n \}$ is the set of all the methods defined in the program.
S = $\{ s_1, s_2, \ldots, s_n \}$ is the set of all the statements in the program.

The pseudo code of our slicing algorithm is given in Algorithm 2. HD slice takes the intermediate graph representation (rEOOSDG) of the program under consideration as its input. Though the algorithm can work for both EOOSDG and rEOOSDG, we have taken rEOOSDG as the input. The reason being that the time for computing the slice is less in rEOOSDG. This is shown in Figure 4.5. The proposed algorithm computes a forward slice with respect to the point of modification taken as the slicing criterion. The algorithm then traverses backward in two passes from each node obtained in the forward slice to determine a set of affected program parts. The slice is then hierarchically decomposed into sets of packages, classes, methods and statements. This gives the impact of change at different programming levels.
The notations used in the algorithm are:

i. P_1 - The set of packages extracted from rEOOSDG that are affected by the modification.

ii. C_l - The set of classes extracted from rEOOSDG that are affected by the modification.

iii. M_1 - The set of methods extracted from rEOOSDG that are affected by the modification.

iv. S_1 - The set of statements extracted from rEOOSDG that are affected by the modification.

v. E - The set of edges in rEOOSDG.

vi. N - The set of nodes in rEOOSDG.

vii. $type_1$, $type_2$ - The set of some specific edges along which backward traversal is restricted.

4.2.4 Proposed Hierarchical Regression Test Case Selection (HRTS) Algorithm

In this section, we discuss our proposed algorithm named *Hierarchical Regression Test Selection (HRTS)*, for generating selective regression test cases. We maintain the test case coverage distribution for our example program (Figure 4.2) as shown in Table 4.1. Our proposed HRTS algorithm takes the decomposed slices and the test case coverage information as input. The algorithm selects those test cases that affect at package level, class level, method level and statement level. The outcome of the algorithm is a set of change-based hierarchically selected test cases suitable for regression testing. Algorithm 3 gives the steps of our proposed hierarchical regression test case selection approach in pseudocode form.

Table 4.2: Summary of test case selection for the example program in Figure 4.2

Level	Selected Test Cases	Number of Selected Test Cases
Package	$T1 - T20$	20
Class	$T1 - T10$	10
Method	$T6 - T10$	5
Statement	$T6 - T10$	5

4.2.5 Working of the Algorithms

In this section, we explain the working of our proposed HD slice algorithm and HRTS algorithm. We have taken an example Java program shown in Figure 4.2 as our case

Algorithm 3 *HRTS*

```
Input: Decomposed Slices $\{P_1, C_1, M_1, S_1\}$ , Test coverage information.
Output: A set of hierarchically selected change-based test cases.
Step 1: Construct the EOOSDG for the program.
Step 2: Invoke RER(EOOSDG)
Step 3: Invoke HDslice(rEOOSDG)
Step 4: Select the test cases step by step from the package level to the statement level.
  i. Let there be $n$ number of test cases in the test suite $T$, where $T = \{t_1, t_2, \ldots, t_n\}$. The set of packages
     covered by each test case $t_i$, i = 1, 2, ..., n, is represented by $P_{t_i}$.
  1: Determine the set of test cases selected at the package level for retesting the program.
  2:    $T^{'} = \{\}$ // $T^{'}$ is initialized.
  3:    For each $t_i \in T$, do the followings:
  4:       $P_t' = Pk \cap P_{t_i}$
  5:       If $P_t'$ is non-empty, then
  6:          $T' = T' \cup \{t_i\}$, where $T'$ is the set of selected test cases at package level.
  7:    End For
 ii. Determine the set of test cases selected at the class level
  8:    $T^{''} = \{\}$ // $T^{''}$ is initialized.
  9:    For each $t_i \in T'$, do the followings:
 10:       $C_t' = Cl \cap C_{t_i}$, where $C_{t_i}$ is the set of classes covered by test case $t_i$.
 11:       If $C_t'$ is non-empty, then
 12:          $T'' = T'' \cup \{t_i\}$, where $T'' \subseteq T'$ is the set of selected test cases at class level.
 13:    End For
iii. Determine the set of test cases selected at the method level
 14:    $T^{'''} = \{\}$ // $T^{'''}$ is initialized.
 15:    For each $t_i \in T''$, do the followings:
 16:       $M_t' = Mt \cap M_{t_i}$, where $M_{t_i}$ is the set of methods covered by test case $t_i$.
 17:       If $M_t'$ is non-empty, then
 18:          $T''' = T''' \cup \{t_i\}$, where $T''' \subseteq T''$ is the set of selected test cases at method level.
 19:    End For
 iv. Finally, determine the statement level slice and the set of test cases selected at the statement level.
 20:    $T^f = \{\}$ // $T^f$ is initialized.
 21:    For each $t_i \in T'''$, do the followings:
 22:       $S_t' = St \cap S_{t_i}$, where $S_{t_i}$ is the set of statements covered by test case $t_i$.
 23:       If $S_t'$ is non-empty, then
 24:          $T^f = T^f \cup \{t_i\}$, where $T^f \subseteq T'''$ is the set of selected test cases at statement level.
 25:    End For
```

study. The corresponding rEOOSDG of the program that is given as an input to Algorithm 2, is shown in Figure 4.4. Suppose, the object *s2* at line 23 of the example program is changed to *s1*. As a result, the method of class *Rectangle* at line 44 is invoked instead of the method of class *Triangle* at line 33. Thus, the change at Line 23 (corresponding to node 23 of rEOOSDG) results in erroneous output. Computing the slice of the program with the modification point as the slicing criterion, helps in detecting the other program parts affected by this change. To compute the slice, the algorithm first traverses in the forward direction from the point of modification to determine all those nodes/program parts that may be affected by the change.

The nodes that are added to worklist Q_1 in the forward traversal of $node23$ are $23, 52, 33, A3\_out, 53, 54, f6\_out, 34, f3\_out$. Then, from each of the nodes reached in the forward traversal, the algorithm traverses in the backward direction to find all those nodes that might have affected the reached nodes. This backward traversal is done in two passes. In the first pass, the algorithm traverses backward along all the edges except *polymorphic call edge*, *inherited membership edge*, *parameter_out edg*e, and *generic_out* edge. In Pass-2, the algorithm traverses backward from all the nodes marked in Pass-1 along all the edges except *parameter_in edge*, *generic_in edge* and any edge traversed in pass-1. For example, the nodes added to worklist Q_2 for node 23 in Pass-1 are $\{4, A21\_1\_out, A21\_2\_out, 3, 1, 2, 21, A5, A6, 19, 20, 7, 6\}$. The nodes added to worklist $Q3$ in Pass-2 are {f27_1_out, f27_2_out, 27, 29, 30, 21, 24, f3, f4, 46}. Similarly, the HD slicing algorithm finds the affecting nodes for other nodes in Q_1. Thus, the final slice Q is given as the union of Q_1, Q_2, and Q_3 that comprises of all the affected and affecting nodes. The nodes included in the final slice are shown as shaded nodes in Figure 4.4. The slice is then hierarchically decomposed into packages, classes, methods and statements.

Then Algorithm 3 takes these decomposed slices along with the test case coverage information as input. The two package nodes that are present in the slice are $node1$ and $node2$. So from Table 4.1, we select the test cases $T1 - T20$ that cover these two package nodes. In the second level, the class nodes that are sliced are $node3, node46$ and $node24$ that correspond to *TestShape*, *Shape* and *Triangle* classes of the program, respectively. We select those test cases that cover these sliced classes. Therefore, we select $T1 - T10$ out of the 20 test cases selected in the first level. Similarly, in the third level i.e. the method level, we select $T6 - T10$ as these test cases cover the affected methods. We also select $T6 - T10$ in the fourth level (statement level slicing), as all these 5 test cases also cover the sliced statement nodes. Finally, we get the set of five test cases ($T6 - T10$) that can be used to retest the program. This is shown in Table 4.2. In Figure 4.7, we show the implementation result of hierarchical test case selection for the input $node23$.

Correctness of our Algorithms

In this section, we present the proof of correctness of our proposed algorithms.

Theorem 4.2. *Algorithm 2 always finds the correct slice with respect to a given slicing criterion (modification point).*

Proof. We use mathematical induction to prove the above theorem. Let the nodes of the input intermediate graph G, correspond to the packages, classes, methods and statements of an OO program P. Let statement s be the single node in G. Then, HD slice algorithm correctly computes the slice $\psi(s)$ with respect to some modification at s, such that $\psi(s) = \{s\}$, where s is itself the slicing criterion. Let $\{s, s_1\} \in G$ be the only two nodes. If there exists some dependence (edge) between s and s_1, then $\psi(s) = \{s, s_1\}$. Using this argument, we claim that Algorithm 2 correctly computes the affected nodes (slice) in the presence of two nodes. Let there be n nodes in G including s. In Step 2 of the algorithm, we traverse in the forward direction to detect all those nodes that are dependent on the modified statement s. It is obvious that in Step 2 of the algorithm, we have a set of nodes which have a direct as well as transitive dependence on the modified statement. If a node is not affected then it will not be included in the worklist, Q_1. Further, Steps 3 and Step 4 ensures that we mark all those nodes on which the selected nodes in Q_1 depend upon and could have affected the modification at s. The empty conditions at Lines 9, 17, and 24 ensures the termination of the algorithm after a finite number of iterations. Step 5 computes the resultant slice of affected nodes. Step 6 of the algorithm decomposes the affected nodes in $\psi(s)$ into respective sets of packages, classes, methods and statements. As $\mid \psi(s) \mid$ is finite, this step terminates in finite time. Hence, HRTS algorithm correctly computes the slice. $\square$

Theorem 4.3. *Algorithm 3 always hierarchically selects the change-based affected test cases for regression test cases.*

Proof. Let the test suite T contains k number of test cases to test P, where k is a finite number. Let the coverage information of each test case $t_i \in T, i = 1, 2, \ldots, k$ is available, before any modification is done to P at s. Let $t_1 \in T$, then Algorithm 3 correctly selects t_1, if t_1 covers the affected nodes. Using this argument, let $\{t_1, t_2\} \in T$, then Algorithm 3 correctly selects the test case $t_i, i = 1, 2$, that covers the affected nodes. For $t_1, t_2, \ldots, t_k \in T$, Step 4(i) of the algorithm selects those test cases that cover the affected packages and discards the rest. Similarly, in the subsequent steps 4(ii), 4(iii), and 4(iv), the algorithm filters the test cases on the basis of their coverage of affected classes, methods and statements, respectively. The number of test cases is finite and with each hierarchical selection the number reduces further. Further, Step 4(iv) of the algorithm guarantees that the algorithm stops after the test cases are selected at the statement level coverage. This establishes the correctness of the algorithm. $\square$

4.2.6 Complexity Analysis of HRTS Algorithm

The Complexity analysis of HRTS algorithm is as follows:
Space Complexity:

Let the program under consideration contains N statements. Each node in rEOOSDG represents a single statement of the program. However, some extra nodes are required to represent the actual and formal arguments of the method invocation and method definition. For such statements in the program, the number of extra nodes required is equal to the number of actual and formal arguments present in the program. Let us assume that too many parameters in a method definition are not allowed. Let the number of parameters present in the program be k, where k is some bounded small positive number. If each statement of the program is represented in rEOOSDG by k number of extra nodes (assuming each statement has actual and formal arguments), then it can be stated that the space requirement of rEOOSDG is $O\left(kN^2\right)$. Since, k is a small bounded positive integer, we can conclude that the space requirement of rEOOSDG is $O\left(N^2\right)$. Apart from this, some additional space is required by the algorithm in maintaining the packages, classes, methods, statements and the coverage information for each test case. The additional space requirement is as follows:

i. We have assumed that the total number of lines of code in our program is N. Therefore, the number of packages, classes, methods and statements present in the program will be less than equal to N. So, we can say that the space required to maintain this additional information about packages, classes, methods and statements present in the program will be $O(N)$.

ii. Let the number of test cases used to test the original program be m, where m is a bounded positive integer. Each test case will maintain the coverage information of packages, classes, methods and statements. Assuming that each test case covers all the packages, classes, methods and statements in the program, the total space requirement would be $O(mN)$. As m is a bounded positive integer, so the space requirement is $O(N)$.

Therefore, the total space requirement for our HRTS algorithm is $O\left(N^2+N+N\right) \equiv O\left(N^2\right)$.
Time Complexity:

Let N be the set of vertices and E be set of edges in rEOOSDG. Since, each node in the graph is visited (using DFS Algorithm) only once, so the time complexity is in the order of $(N+E)$. If the time spent in each recursive call is ignored, then each vertex u can be processed in $O\left(1+d_G^+(u)\right)$ time, where $d_G^+(u)$ is the out-degree of node u. So the total time required for our algorithm is given by

$$TotalTime = N + \sum_{u \epsilon N}\left(1+d_G^+(u)\right) = N + \sum_{u \epsilon N} d_G^+(u) + N = 2N + E \approx \Theta(N+E)$$

The operations involved in the algorithm for hierarchical decomposition slicing and selection of test cases are intersection and union which require two sets as operands. Assuming that each set contains N elements, the worst case run time of each of the above operations will be $O(N^2)$. Therefore, the worst-case run time of our algorithm is $O(N^2)$.

4.3 Implementation

In this section, we briefly describe the implementation of our approach.

Table 4.3: Summary of change types in Java programs.

Sl. No.	Change Types	Sl. No.	Change Types
1	Adding a package	6	Adding a local variable
2	Deleting a package	7	Deleting a local variable
3	Changing the identifiers of a class	8	Changing the type of the local variable
4	Changing the return type of a method in a class	9	Changing the operator of a condition
5	Changing the type and number of parameters of method in a class	10	Changing wrong variable

4.3.1 The sample programs

We conducted the experiments on fifteen medium-sized programs of different specifications as shown in Table 4.4. Out of these fifteen programs, ten benchmark programs (Stack, Sorting, BST, CrC, DLL, Elevator_spl, Email_spl, GPL_spl, Jtopas, Nanoxml) are taken from Software-artifact Infrastructure Repository (SIR) [59] and other five programs are developed as academic assignments. SIR [59] is a repository of programs and tools to support controlled experimentation with testing and regression testing techniques. These smaller programs are chosen to ascertain the correctness and accuracy of the approach, keeping in mind that they represent a variety of Java features and applications, the test cases are available or can be easily developed, and coverage information can be computed. The modifications that

Table 4.4: Result obtained for regression testing of different programs.

Sl.No.	Programs	Lines of Code	Total # Test Cases	# Selected Test Cases for regression testing
1	Example Program	54	20	5
2	Calculator	75	15	7
3	Elevator	90	25	10
4	Stack	114	22	9
5	Sorting	130	16	5
6	BST	130	20	12
7	CrC	261	18	6
8	DLL	277	24	6
9	Notepad	300	17	8
10	ATM Application	900	33	12
11	elevator_spl	1046	15	10
12	email_spl	1233	18	11
13	GPL_spl	1713	22	14
14	jtopas	5400	16	9
15	nanoxml	7646	14	7

are made to the above mentioned programs include modification to the data types of member variables, modification of expressions in a method, modification of the object relation, addition and deletion of a new member variable, etc. A summary of the different change types considered for the experimental programs are listed in Table 4.3. The third column of Table 4.4 gives the size of each of the program in terms of *Lines of code* (*LOC*). The smallest program has 54 LOC, and the largest program has 7646 LOC. The total LOC for all the fifteen programs is 19369 and the average LOC per program are 1291. The fifteen EOOSDGs are constructed using our prototype tool. The smallest EOOSDG has 91 nodes, and the largest has 26451 nodes. The total number of nodes for all the fifteen EOOSDGs is 88511, and the average number of nodes per EOOSDG is 5901.

The experimental programs were given to five post graduate students to develop the required test cases. Each of the students developed a set of test cases for each of the programs using JUnit eclipse plugin [1]. These test cases were then executed in JaBUTi [196] to find their coverage percentage. We considered those test cases that were having more than ninety percent of code coverage for a particular program for our regression test selection process. The fourth column of Table 4.4 gives the total number of test cases initially taken into consideration for each program. The total number of test cases considered for all the programs is 295 with a mean of 20 test

[1]http://www.tutorialspoint.com/junit/junit_plug_with_eclipse.htm

cases per program.

4.3.2 Experimental settings

All the algorithms related to slicing and hierarchical regression test case selection are implemented using Java and Eclipse v3.4 IDE [2] on a standard Windows 7 desktop. The proposed approaches are completely based on the intermediate graph *EOOSDG* of the modified program. The identification of the dependences to construct the intermediate graph, follows a build on build approach, i.e. we use the existing APIs and tools to build the graph instead of developing the source code parser from scratch. Source code instrumentation and generation of the intermediate graph are implemented by using XPath parser on srcML (SouRce Code Markup Language) representation of the input Java program. Thus, srcML is the XML [48] (eXtended Markup Language) representation of the input Java program. The input program is converted to srcML using src2srcml tool. This srcML representation is then used to extract the dependences between program parts by using the XPath parser. The details of the program transformation [26] and fact extraction process can be referred in [12, 50, 51, 137]. Many other APIs and tools (such as Document Object Model (DOM) and Simple API for XML (SAX)) can be used to extract facts from the srcML representation.

In this thesis, the fact and dependence extraction are done using XPath. XPath [125] is a language support used by XSLT (extensible stylesheet language) parser to address specific part(s) of the entire XML document. The choice of using XPath is because of its simplicity and easy extraction by directly tracing to the location of the information. This also works on both visioXML and srcML formats of XML. The source code is first instrumented and then dependences in the program are identified and extracted into the program dictionary to construct intermediate graphs. The modified statement (instrumented number) is taken as input along with the intermediate graph, to slice the affected nodes. Most of the dependences at package level, class level and method level, are extracted from the Imagix4D XML data. Imagix4D [3] is a static analysis tool that gives the graphical representation of most of these dependences. The statement level dependences such as control dependence, data dependence, etc. [87] are extracted from the srcML representation of the program. This process of graph construction is followed through in the rest of the thesis.

[2]https://eclipse.org/

[3]http://www.imagix.com/products/source-code-analysis.html

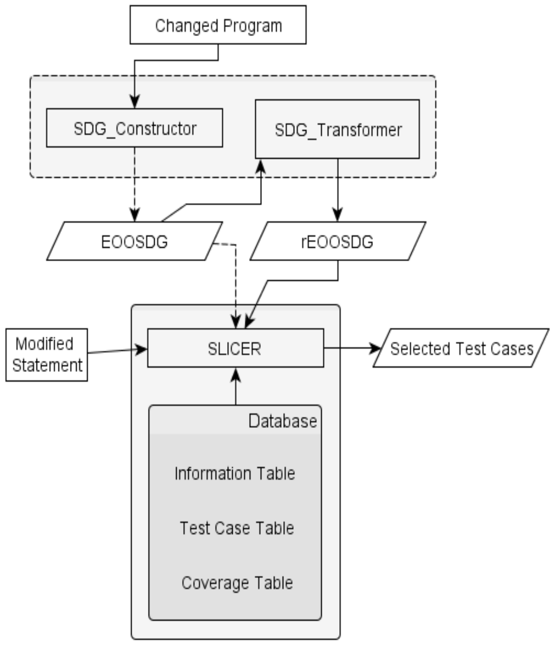


Figure 4.6: Architectural model of the hierarchical regression test selection method

4.3.3 Architectural Model of Regression Test Case Selection

The functional components of the prototype tool developed for implementing the proposed approach are shown in Figure 4.6. It consists of the following components:

- SDG_Constructor
- SDG_Transformer
- Slicer
- Database

The solid (dashed) arrows show the actual (optional) flow of information between the functional components of the tool. The rectangular blocks represent the functional components and the parallelogram blocks represent the outputs from the corresponding functional components. We developed the prototype tool using Java and Eclipse. The changed program under test is given as input to the SDG_Constructor that generates the required intermediate graph named EOOSDG as explained in Section 4.2.1. The SDG_Constructor is basically a parser that fist instruments the code before anlayzing the input program for finding the possible dependences. This SDG_Constructor parser is reified as discussed in the previous section. The details of the internal architecture of the parser are not discussed here for space constraints. In the first two passes of the parser, the code is instrumented and in another two passes the EOOSDG is generated based on the dependence analysis information. To make this intermediate graph (EOOSDG) scalable, SDG_Transformer removes some of the redundant edges as described in Section 4.2.2 to get a reduced graph rEOOSDG. The information about the program elements (such as packages, classes, methods and statements) along with the coverage of the test cases are stored using MySQL [4] database. The data in the database are organized in three kinds of tables: Information table, Test Case table and Coverage table:

i. *Information Table*: The information about the program elements such as packages, classes, methods and statements are maintained in these tables.

ii. *Test Case Table*: Information such as test case id, inputs and the expected output are maintained in these tables. The test cases that executed the original program are the desired candidates maintained here in these tables.

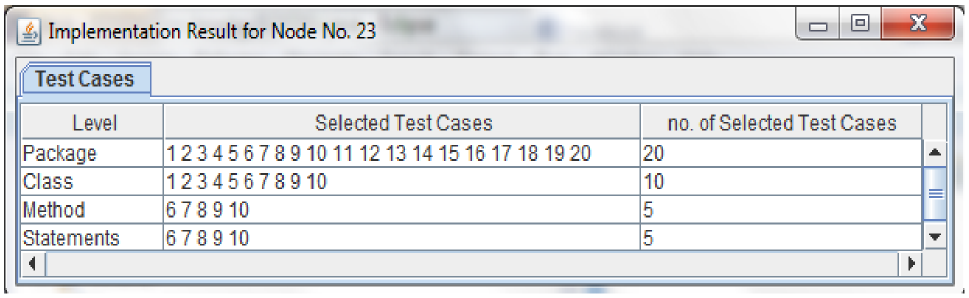
iii. *Coverage Table*: These tables keep the information about the coverage information of each test case. The packages, classes, methods and the statements covered by each test case are maintained in these tables.

Our slicer component is not fully automated to detect the changes in the program from the intermediate graph (EOOSDG/rEOOSDG). Therefore, it manually takes the modified statement (instrumented number) as input along with the intermediate graph EOOSDG and/or rEOOSDG. Taking the point of modification as the slicing criterion, the slicer computes the slice consisting of the affected parts with respect to the modification made to the input program. The affected program parts in the slice are then decomposed into packages, classes, methods and statements by using the information contained in *Information Table* of the database. The slicer component requires additional information contained in *Test Case Table* and *Coverage Table* of the database for selecting the required test cases. The slicer maps these decomposed program elements with the test case coverage information and selects the required test cases that are suitable for regression testing.

4.3.4 Result Analysis

Initially, we consider the example program in Figure 4.2 as input to our algorithm. Figure 4.8 shows the total number of test cases that get selected from Table 4.1, for

[4]http://www.mysql.com/

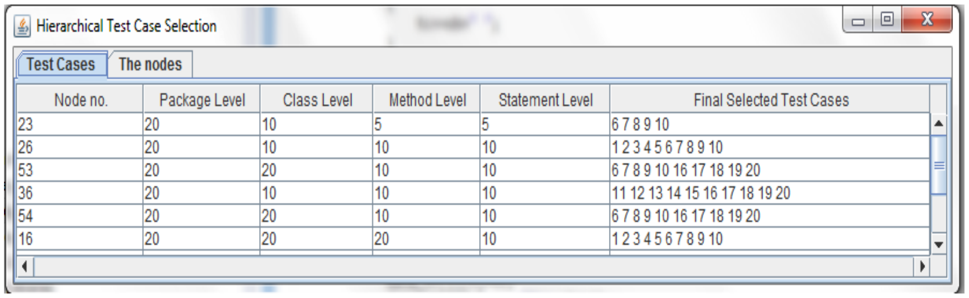

Implementation Result for Node No. 23

Test Cases

Level	Selected Test Cases	no. of Selected Test Cases
Package	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	20
Class	1 2 3 4 5 6 7 8 9 10	10
Method	6 7 8 9 10	5
Statements	6 7 8 9 10	5

Figure 4.7: Summary of hierarchical test case selection for node 23 of rEOOSDG in Figure 4.4.

each of the corresponding change made at different nodes. The selected test cases are then used for regression testing. Figure 4.9 shows the time based comparison between EOOSDG and rEOOSDG of different programs to detect their affected program parts. The first ten programs given in Table 4.4 are taken for computing the reduction in time achieved in computing the slices from their corresponding EOOSDG and rEOOSDG. The slice for rest five programs are computed directly from their corresponding EOOSDG. Thus, the average reduction in time to detect the affected program parts of different programs with respect to a given point of modification is 28.1% approximately. In the fifth column of Table 4.4, we show the test cases finally selected for regression testing by using our proposed approach.


Hierarchical Test Case Selection

Test Cases | The nodes

Node no.	Package Level	Class Level	Method Level	Statement Level	Final Selected Test Cases
23	20	10	5	5	6 7 8 9 10
26	20	10	10	10	1 2 3 4 5 6 7 8 9 10
53	20	20	10	10	6 7 8 9 10 16 17 18 19 20
36	20	10	10	10	11 12 13 14 15 16 17 18 19 20
54	20	20	10	10	6 7 8 9 10 16 17 18 19 20
16	20	20	20	10	1 2 3 4 5 6 7 8 9 10

Figure 4.8: Hierarchical test case selection for different input nodes

In Table 4.5, we validate the efficiency of our proposed HD slicing technique by comparing our work with an existing approach by Li et al. [130]. Table 4.5 clearly shows that our approach finds the affected program parts in the form of computed slices in less time. The total time taken by our approach to compute the slices is 324.9 ms, while the total time taken by Li et al. approach is 343.7 ms. Our approach discovered 1812 total affected nodes for all the fifteen programs, while Li et al. approach identified 2154 nodes out of which many were not relevant

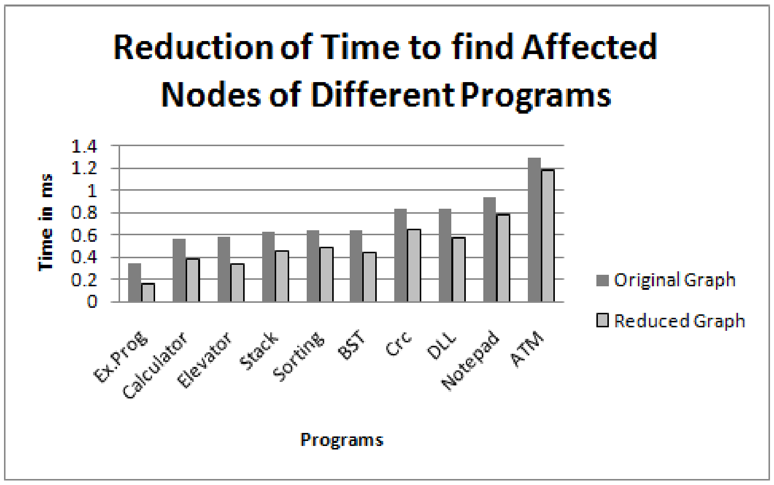


Figure 4.9: Time based comparison between EOOSDG and rEOOSDG of different programs to detect their affected parts

Table 4.5: Comparison of Hierarchical Slicing versus HD slicing.

Sl. No.	Program	Hierarchical Slicing (Li et al. [130])		HD Slicing (Our Approach)	
		# Selected Nodes	Time (ms)	# Selected Nodes	Time (ms)
1	Expt Prog.	35	16.6	33	15.52
2	Calculator	63	15.9	51	14.6
3	Elevator	61	15.6	54	14.71
4	Stack	86	19.87	72	18.56
5	Sorting	97	19.78	86	18.77
6	BST	78	19.98	74	18.79
7	Crc	123	23.37	94	21.83
8	DLL	142	23.96	83	22.16
9	Notepad	73	25.1	68	23.8
10	ATM	152	24.81	97	24.08
11	Elevator_spl	133	26.2	105	24.9
12	Email_spl	107	27.31	98	25.83
13	GPL_spl	118	26.2	112	25.6
14	Jtopas	283	27.8	241	26.63
15	Nanoxml	603	31.3	544	29.12

to the slicing criterion. This justifies that our approach computes precise slices. Figure 4.10 shows a comparison of the percentage of test cases selected for regression testing. By using our proposed approach, there is an average reduction of 56.3% (approximately) in the number of test cases selected for regression testing. While Tao et al. [188] approach could achieve only 43.3% reduction in the selection of the affected test cases. This is mainly due to the selection of some irrelevant nodes

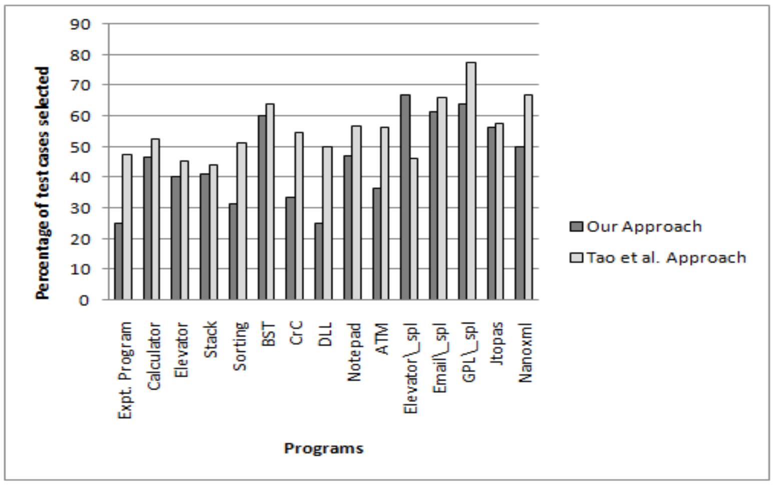


Figure 4.10: A comparison of the percentage of test cases selected for regression testing.

(as shown in Table 4.5) by the slicing algorithm used in [188]. The selected test cases are also efficient to find the critical errors early during regression testing of the programs.

4.3.5 Threats to Validity

Like many other novel approaches, this work also suffers from some threats to its validity as given below:

- Although we used a diverse set of programs for our experiments that are of moderate size, but these do not represent the larger industrial applications. Therefore, not all the features and complexities of real-time industrial applications are considered in this approach.

- Our approach is based on the concept of HD slicing performed on an intermediate graph representation of the program under consideration. The addition of extra nodes such as parameter_in, parameter_out, actual_in, actual_out, and package nodes, to the graph to represent the program parts accurately, increases the size of the graph. This is a possible threat on the scalability of the approach.

- To overcome this limitation, we propose to identify and remove the redundant edges present in the intermediate graph representation. This reduces the size of the graph and results in faster computation of the slices. Even though the redundant edge removal process works fine with the graph reachability algorithm, for some other applications it may be essential to investigate the semantic effect of the edges. This forms another threat to our approach.

- The next threat to this approach is that it selects the test cases based on the affected program parts that it covers. However, the fault proneness of the affected program parts and the criticality of the faults discovered by the selected test cases need more investigation.

4.4 Comparison with Related Work

In this section, first we discuss the work related to our work and then compare some of these work with our approach. First, we discuss the available related work on program slicing. Then, we discuss the existing related work on regression testing. Many researchers [3, 101, 123, 161, 201] have proposed several techniques for slicing of programs. The original work proposed by Weiser [201] focused on computing the slices from the control flow graph of the program. Ottestein and Ottenstein [161] for the first time defined slicing as a graph reachability problem. In their method, they used a program dependence graph to compute the static slices of a program.

The concept of *System Dependence Graphs (SDG)* to represent the inter-procedural programs was introduced by Horowitz et al. [101]. Later, Larsen and Harrold [123] extended the concept of SDG to incorporate the OO features. In this method, each class entry is represented by a *Class Dependence Graph (ClDG)*. A ClDG represented both the control dependences and data dependences inside a class.

Krishnaswamy [119] identified some more dependences relevant to OO programs in addition to the control and data dependences. But, these dependences do not completely cover a true OO program such as a Java program. In our proposed EOOSDG, we have added some new dependences applicable to Java programs, such as *package dependence*, *type dependence* and *read/write dependence*. This is a suitable representation for a true OO program like Java. Harrold et al. [90] proposed an algorithm to identify the presence of dangerous edges in the intermediate graph for safe regression test selection. This method compared two nodes in the proposed *Java Interclass Graph (JIG)* of a program P and that of the modified version P' to identify the execution path of a test case in P and P', so that it can be known whether any edge is dangerous or not. To make the comparison between the nodes, they have used the lexicography equivalence of the text labeled on each node. For example, if a class Y in package *pkg* extends a class X in the same package, and X

implements interface I in package *abc*, then the text associated with the node for class Y will be $Java.lang.Object : abc.I : pkg.X : pkg.Y$. As the level of inheritance will be deeper, the text will become more lengthy and comparison will incur more run-time overhead. In our approach, we do not perform any such comparisons. So, we save time by avoiding this computational overhead. We select only those test cases that cover the affected program parts with respect to the modification done to the program.

Many researchers have proposed different approaches to compute slices of Java programs. Some of the slicing mechanisms are based on the dependence graphs like PDG and SDG, while other approaches are based on the Java byte-code analysis. Chen et al. [44] discussed different dependences possible in a Java program and proposed slicing of classes based on PDG. In their method, the program dependence graph consists of a set of independent PDGs. In slicing of classes, the slicing criterion taken is $\langle s, v, class \rangle$, where s is the statement number, v is the variable and *class* is the name of the class to be sliced. The slice is computed by traversing backward from s and marking all the statements and data members used in the class based on the PDG. Allen et al. [11] extended the work of Chen et al. [44] on program slicing by using SDG. In their work, they proposed slicing of programs in the presence of exceptions. The focus was mainly to determine the control and data dependence due to the presence of try, catch and throw blocks in the program. They have not considered other Java specific features for slicing. But, in our approach, we have considered the OO features like packages, super, method overriding, etc. for the purpose of slicing.

Wang et al. [200] proposed slicing of Java programs by using compressed byte-code traces. They represented the byte-code corresponding to an execution trace of a Java program. Then, through backward traversal of the execution trace, they determined the control and data dependences on the slicing criterion. This approach requires the trace table to be constructed for each method. If a program will have too many methods, then this approach will be disadvantageous to compute the slices. This is because of the increased execution overhead in maintaining the execution trace tables. This work is also silent regarding the execution trace of the methods that are nested, overloaded and/or overridden. Similarly, Hammer et al. [85] have proposed slicing of a Java program in the presence of objects as parameters. The analysis of the dependences is based upon an *Intermediate Representation (IR)* generated from the byte-code of the program. A good point-to analysis is a prerequisite of this algorithm to compute more precise slices.

Slicing of Java programs in all of the above work [11, 44, 85, 200] was proposed by taking into consideration a specific feature or type of dependence present in a Java program. Whereas, the overall impact of the features on the dependences such as the dependence due to the presence of packages and other specific Java features are not considered. Our approach has made a decent effort in analyzing all the possible dependences in OO programs and computing a more accurate slice. To be able to employ slicing for regression testing, we need to identify all those statements that affect the modified statement and those statements that may get affected by the modification. But, most of the existing approaches [11, 44, 85, 200] are based upon either forward traversing or backward traversing. This will only result in the partial identification of the affected statements due to the modification. But, our approach gives a better result for regression testing due to the following reason: both forward and backward traversals of our approach correctly find all the program parts that get affected and that may affect other program parts due to the change.

Software maintenance being the most important and expensive activity in the process of Software Development Life Cycle (SDLC), many researchers have proposed approaches for ordering the test cases of procedural programs. Rothermel [142, 175] and Elbaum [66] have considered different types of program coverage criteria such as total statement coverage, additional statement coverage, total function coverage etc. Jeffrey and Gupta [104], proposed a method for prioritizing the test cases for regression testing based on the coverage of relevant slice of the output of a test case. They assigned weights to the test cases to determine their priority. They determined the test case weight by summing up the number of statements present in the relevant slice and number of statements exercised by the test case. Korel et al. [113] prioritized the regression test suite by considering the state model of the system. Whenever, the source code was modified, the corresponding change in its state model was identified. These modified transitions along with the runtime information were used to prioritize the test cases. However, the available techniques were of little help when they were applied to regression testing of *OO programs*.

Li et al. [130] used hierarchical slicing for regression test case selection of OO programs. Their proposed model consisted of three levels: syntax analysis, generation of dependence graphs, and computation of slices. They proposed different dependence graphs such as *package level dependence graph (PLDG)*, *class level dependence graph (CLDG)*, *method level dependence graph (MLDG)* and *statement level dependence graph (SLDG)* that were based on the slicing criteria. When any modification is done to a statement, the dependence of that statement with its method, class and package can be easily detected because of the maintenance of different levels of graphs. Identification of other packages, classes, methods and statements related to the modified statement can also be easily done. The overall performance had improved as the irrelevant packages, classes, methods and statements were discarded from the generated graph. But, the proposed method required all the different graphs (PLDG, CLDG, MLDG, SLDG) to be generated for each change made to the program and was not very advantageous in case of frequent changes. Thus, to avoid the above mentioned problem, the slicing criterion was fixed. Whereas, we have implemented the hierarchical slicing technique on the rEOOSDG which is not constrained to any fixed change. It rather works for any number of changes made to any statement, without requiring us to maintain additional graphs. So the space requirement of our approach is much less than that of Li et al. [130]. If the change made to the example program triggers some new changes to be made, then our approach is capable to handle it.

Tao et al. [188] applied hierarchical slicing for regression testing of OO programs. They have constructed separate graphs for packages, classes, methods and statements even if they were not affected by the change. This again required more space requirement. This is because with the increase in the program complexity, there will be an increase in the number of packages, classes, methods and statements which are required to be represented as separate graphs. But, in our approach, we only maintain the graph rEOOSDG. This does not impose any additional space requirement of constructing different graphs. Tao et al. [188] have used the same hierarchical slicing technique as given in [130]. In Table 4.5, we show the relative advantage of our approach over the hierarchical slicing technique in [130]. Therefore, it justifies that our regression test case selection approach will compute the slices efficiently and select the test cases in less time compared to the approach of [188].

In some work [66, 104, 175], only control dependence and data dependence are considered for program analysis. But, we identified some more dependences such as *package membership dependence*, *type dependence* and *read/write dependence*, that represented various object relations so as to consider more features of OO programs and computed the slices more accurately. Therefore, in our approach, appropriate test cases are selected more accurately for the purpose of regression testing.

4.5 Summary

We proposed an application of slicing to regression test selection based on the Extended Object-Oriented System dependence Graph (EOOSDG). We considered some new dependences in addition to control and data dependences, that play a crucial role in the regression test selection. It would be interesting to assess such effects. In this approach, we proposed a method to reduce the space requirement of the intermediate graph by removing the redundant edges from EOOSDG and hence addressed the scalability issue of the intermediate representation to some extent. The affected program parts are also detected in less time due to the removal of redundant edges. The average reduction in the time required for identifying the affected program parts with respect to some modification made to the programs is approximately 28.1%. The selected test cases are also found to be very efficient in detecting the regression errors. The average reduction in the selected test cases is approximately 56.3%, for all the programs under consideration. However, theses change-based selected test suite may still be enormous for very large programs. Even it may not always be possible to exhaust all the test cases within a selected test suite during shortage of time and budget. Therefore, it is a dire necessity to minimize even the selected test suite. We address this issue of test suite minimization in the next chapter.

Chapter 5

Regression Test Suite Minimization

Test suite minimization techniques aim to identify a reduced test suite that can still assure software quality. The size of the reduced test suite should therefore be much smaller than the original test suite.

The rest of the chapter is organized as follows: Section 5.1 presents a motivating example for our proposes test suite minimization approach. Section 5.2 introduces the proposed minimization framework and discusses the pre-computations required for formulating the minimization problem. We present the experimental setup in section 5.3 and answer some research questions. We also list some of the threats to the validity of this approach. Then, we compare our proposed work with some related work in Section 5.4. We summarize the chapter in Section 5.5.

5.1 Motivating scenario

In this section, we introduce the motivation and necessity for test suite minimization with an example. Consider a program P and a given set of selected test cases $ST = \{t_i\}$. ST for larger and complex programs can be very large for the tester to handle. Supposing the tester wants to minimize ST to MT, such that $MT \subseteq ST$. The testing process is always guided by a set of requirements $R = r_1, r_2, \ldots, r_n$ expressed in terms of code coverage [157], MC/DC coverage, fault-prone nodes coverage, rate of fault detection, etc. The matter of fact is that the minimized test suite should satisfy R. In this chapter, we consider rate of fault detection as our requirement. The computation of rate of fault detection requires prior knowledge of the number of faults present, which is not possible. Therefore, we assume that if the test case covers all the affected fault-prone nodes (statements), then it has a high probability of discovering the faults. We identify the error-proneness by computing the cohesion values of the nodes. Through empirical studies many researchers [8, 10, 34, 65, 117] have validated that modules having low cohesion and high coupling values are more prone to errors. Thus, test cases executing such nodes have a high chance of detecting faults. Therefore, our minimization problem focuses on maintaining the same coverage, by minimizing the cohesion values of the affected nodes. Table 5.1 shows the pre-computed data of the selected test cases for

Table 5.1: Test related data for the example program given in Figure 4.2.

Sl. No.	Test Case ID	Statements Covered	# Statements	Cohesion Wt.
1	$T6$	$1, 2, 3, 4, 6, 7$	6	0.65234375
2	$T7$	$1, 2, 21, 46, 27, 29, 30, 33, 34, 24, 25, 26$	12	9.7807664
3	$T8$	$1, 2, 3, 4, 6, 7, 21, 46, 27, 29, 30, 19, 20, 25, 26$	15	6.8609747
4	$T9$	$1, 2, 3, 4, 6, 7, 21, 46, 52, 27, 33, 30, 34, 24$	14	6.6401414
5	$T10$	$1, 2, 3, 4, 21, 23, 46, 34, 33, 24$	10	8.7971354

the example program given in Figure 4.2. The second column shows the test cases selected for regression testing. The third column shows the statements that each selected test case covers. These statements are affected by some change made to the experimental program. These affected statements are obtained by performing change impact analysis using program slicing. The fifth column shows the sum of cohesion values of the statements covered by the selected test cases. The intention of the testers is to find the smallest number of test cases that covers all the statements and expose maximum number of faults i.e. the sum of the cohesion values is minimum. If the sum of the cohesion values of the statements covered by a test case is minimum, it implies that the test case executes all those statements that have low cohesion values. This can be expressed as a binary integer linear programming (ILP) problem. Modeling the minimization problem in ILP has resulted in obtaining superior results in minimizing the test cases in terms of coverage and minimization constraints [58]. The problem formulation depends on the fact that test suite minimization is represented as an ILP problem. The primary advantage of using ILP is that as long as the problem is solvable, the solution to the minimization problem is guaranteed. The formulation of the minimization problem depends on three things. First, representing the minimized test suite, MT, as an array of binary values, $T = \{t_1, t_2, \ldots, t_n\}$. The value 1 for t_i indicates inclusion of i^{th} test case in MT, and vice-versa for 0 value. Second, identifying the objective function that satisfies the minimization constraints. And finally, to encode linear relationships among the elements of T. The minimization criteria is stated as follows:

- Criterion #1: To maintain the statement coverage.
- Criterion #2: To minimize the sum of cohesion values of the statements covered by the test cases.

Thus, this chapter proposes a test-suite minimization framework that concerns the minimization of cohesion values for maximizing the fault detection, while maintaining the coverage of the statements.

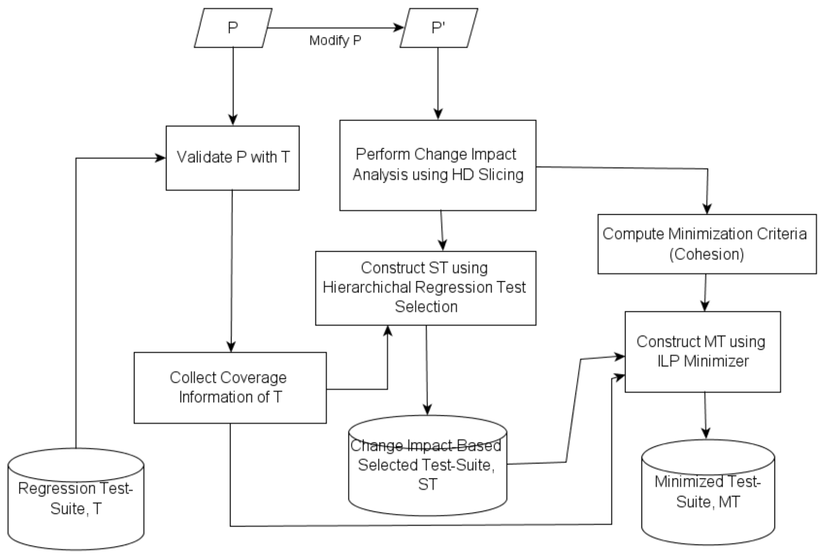

P
Modify P
P'
Validate P with T
Perform Change Impact Analysis using HD Slicing
Compute Minimization Criteria (Cohesion)
Construct ST using Hierarchichal Regression Test Selection
Construct MT using ILP Minimizer
Collect Coverage Information of T
Change Impact Based Selected Test-Suite, ST
Minimized Test-Suite, MT
Regression Test-Suite, T

Figure 5.1: Framework to minimize change-impact-based selected test-suite.

5.2 Proposed Approach for Test Suite Minimization

In this section, we discuss the techniques adopted in reifying the proposed framework for minimization of the selected regression test cases. Based on the above motivations, we propose a framework to minimize a given test suite of an object-oriented program using the cohesion values of the affected program parts covered by the test cases.

5.2.1 Minimization framework

In Figure 5.1, we show the overall framework of our proposed approach. P' is the resultant program after the changes are made to the program P, as part of the maintenance. To validate P, it is executed with all the test cases of T. The collection of coverage information for the test-suite T with respect to P can include any data such as various coverage data, cost data, energy data, time of execution data, etc. The choice of the type of test data collected depends upon the objective of the tester. The change impact analysis includes constructing a system dependence graph (SDG) for P', and performing *hierarchical decomposition slicing (HD Slicing)* [156] to identify the affected program parts due to the changes. Then, based on the coverage information of T and change impact analysis, we hierarchically select the test cases to get the *selected test suite (ST)*. For experimentation, we have taken a sample Java program shown in Figure 4.2. A total of twenty test cases (T1-T20) were taken along with their node (statement) coverage information. All those test cases that covered the affected nodes (with respect to a modification point) are selected hierarchically. The minimization criteria can be any set of data of concern to the testers. However, regardless of the factors considered, the minimization criteria has two aspects. First, to specify a constraint or goal for minimization (e.g. minimizing time or maximizing the rate of fault detection). And second, to specify ways of combining these constraints to find an optimal minimal test-suite. We consider the cohesion values of the covered affected nodes as the minimization criteria. The ILP minimizer uses the coverage information of T, selected test suite ST, and the minimization criteria to compute the minimized test suite, MT.

5.2.2 Regression Test Case Selection

The test case selection process for regression testing of the given program is carried as given in Section 4.2. The steps of the test case selection process are given below:

- First, construct the system dependence graph (SDG) of the program under consideration.
- Second, perform a change impact analysis with respect to the changes made to the program by using the decomposition slicing method given in [156].
- Third, decompose the slice into impacted packages, classes, methods, and statements.

- And finally, select those test cases hierarchically that cover these impacted program parts. The selection starts from a coarse granularity of impacted packages and proceeds to a finer granularity of impacted statements.

Finally, the test suite ST contains five test cases ($T6 - T10$) that are selected for regression testing of our example program given in Figure 4.2 (test cases with their coverage information are shown in Table 5.1).

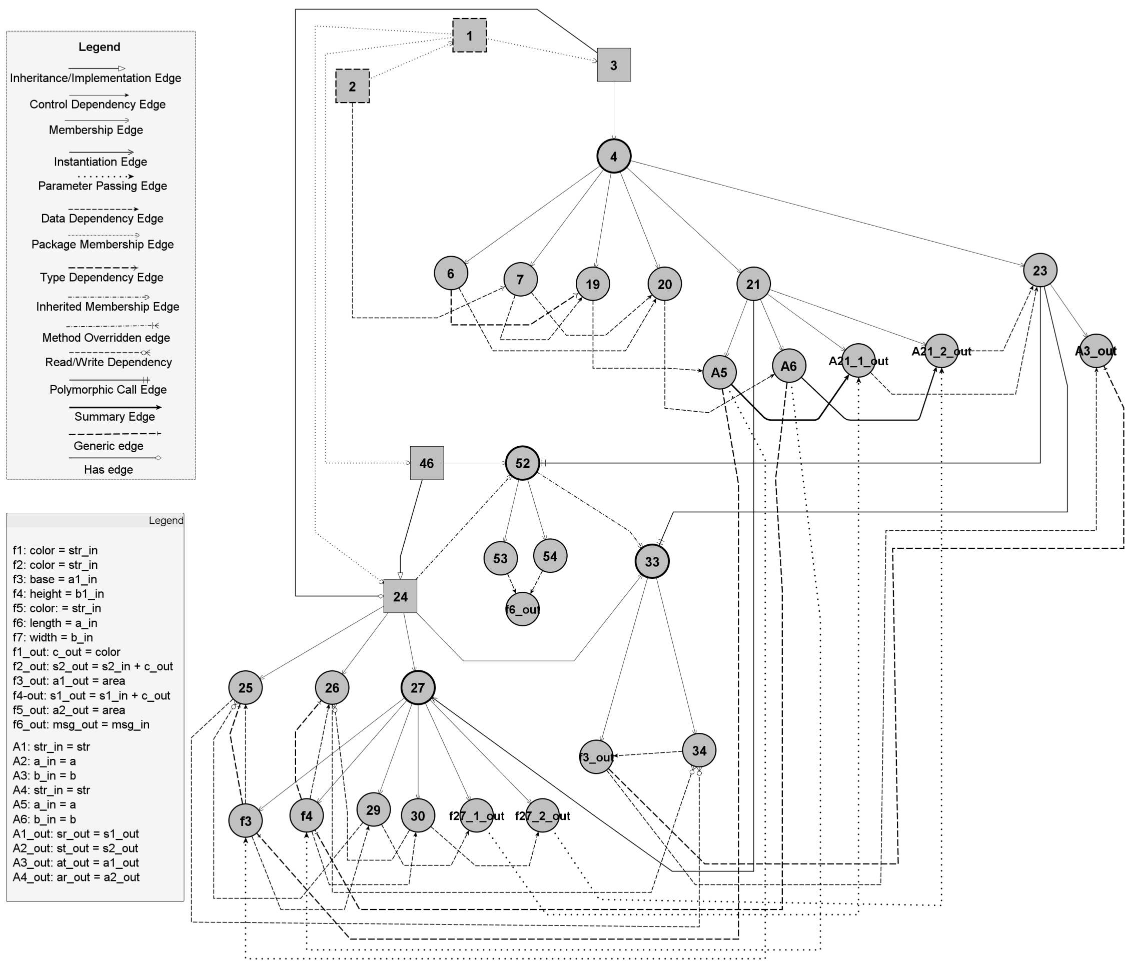


Figure 5.2: Affected Slice Graph (ASG) of the example Java program given in Figure 4.2.

5.2.3 Affected Slice Graph (ASG) Construction using HD Slicing

ASG is the graphical representation of the slice that is computed with respect to some change made to the program. The steps to hierarchical decomposition

(HD) slicing to compute the slices are discussed in Section 4.2.3. ASG shown in Figure 5.2 is a directed graph $G_a = (N_a, E_a)$ that is obtained by performing hierarchical slicing on the intermediate graph given in Figure 4.3. The set of nodes N_a represents the affected program parts such as packages, classes, methods and statements. We identify these affected program parts during hierarchical slicing of the program under consideration. These program parts either affect or get affected by some modification made to the program. The program parts that are affected by some modification made at statement 23 of the example Java program in Figure 4.2 are shown as shaded nodes in Figure 4.3. The set of edges E_a represents the relationship that exists between any two affected nodes $n_1, n_2 \in N_a$ in the ASG. This set of affected nodes and their dependences are then modeled graphically to form the *Affected Slice Graph (ASG).*

5.2.4 Computation of Affected Component Cohesion (ACCo)

The slice obtained as a result of the change impact analysis is represented in the form of a graph named *affected slice graph (ASG).* The ASG for the slice obtained for the example program in Figure 4.2 is shown in Figure 5.2. Each node in ASG corresponds to the statement affected by the change and each edge corresponds to the dependence between them. We define our proposed cohesion measure based on the ASG. We compute the cohesion value of each node of ASG and then update it for the method, class and package nodes.

Cohesion is defined as the tightness with which different elements of a module or a modular system are grouped together. The four major approaches that exist to measure cohesion are as follows:

i. Measuring the cohesion by counting the number of attributes accessed by the member methods of a class, i.e. attribute-method interaction.

ii. Measuring the cohesion based on the count of number of cohesive method pairs, i.e. method-method interaction [21].

iii. Measuring the cohesion based on the degree of similarity between each pair of methods. The degree of similarity is computed by the number of commonly accessed attributes.

iv. Measuring cohesion based on the degree of connectivity between attributes and methods of the class, i.e. attribute-attribute, attribute-method, method-method interactions.

The following discussion summarizes some limitations of the existing approaches [32, 40, 46, 47, 218, 219] that we propose to overcome in our approach. These are as follows:

i. The existing techniques do not address the impact of inheritance on the cohesion measure.

ii. The hierarchical organization of the object-oriented program and the impact of different program parts (statements, methods, classes and packages) on the cohesion measurement have not been studied in the existing techniques.

iii. The degree of inter-relatedness among the different parts of a sliced component, i.e. cohesiveness of a sliced component has not been proposed in the existing literature.

Many work exist [7, 16, 40, 46, 82, 118, 218, 219] that focus on cohesion measurement of packages, classes, methods or statements. However, these studies have not focused on the hierarchical organization of the program parts and the direct and indirect impact of their inter-relatedness on the maintainability. Cohesion also refers to the degree of relatedness of the members in a component that comprises of packages, classes, methods and statements. All these program parts together formulate the objective of the component as a logical function. Therefore, splitting the elements of a cohesive component is difficult in an object-oriented paradigm. Thus, it is essential to consider the degree of relatedness of these packages, classes, methods and statements to measure cohesion. The stronger is the relatedness between these program parts, the more maintainable [10, 211] is the system. Therefore, cohesion metric is sensitive to the changes made to a program as a part of maintenance activity and is often used to predict fault-proneness of the components [8].

In this chapter, our hypothesis is that the components having low cohesion are more prone to errors and require more attention of the testers. We named our proposed cohesion measure *affected component cohesion (ACCo)*. Algorithm 4 gives the pseudocode to compute the proposed change-based cohesion metric. Line 3 and 4 of the algorithm computes the predecessor and successor nodes by traversing in the backward and forward direction respectively, from a given node. Then, we compute the cohesion measure of these affected nodes at Line 6. In Lines 8-17, we update the cohesion values of the method, class, and package nodes. We define our cohesion measure as given below:

Algorithm 4 *findACCo(G_a, n)*

Input: Affected Slice Graph (ASG) $G_a = (N_a, E_a)$,
N_a is set of affected nodes in ASG G_a,
E_a is the set of edges connecting the affected nodes in ASG G_a,
$n = |N_a|$.
Output: Affected Component Cohesion (ACCo) of each node

```
1: $for\ x := V_1, V_2, V_3, \ldots, V_n$    ▷ $set\ x\ not\ visited$
2:     $Set\ status_x = FALSE$
3:   $inflow = call\ BTraverse(G_a, n, x)$
4:   $outflow = call\ FTraverse(G_a, n, x)$
5:   $Dep(n) = inflow(n) \cup outflow(n)$
6:   $ACCo\,(x) = \frac{|Dep(x) \cap N_x|}{n-1}$
7: $End\ for$    ▷ $To\ update\ the\ cohesion\ value\ of\ all\ the\ method,\ class\ and\ package\ nodes$
8: $for\ u := M_1, M_2, M_3, \ldots, M_m$    ▷ $Where\ m\ is\ the\ number\ of\ method\ nodes\ in\ the\ graph.$
9:   $ACCo[u] := \frac{(ACCo[u] + \sum_{i=1}^{j} ACCo[n_i])}{(j+1)}$    ▷ $n_i\ is\ the\ statement/parameter\ node\ of\ method\ M_i,\ j\ is\ the\ total\ number\ of\ statement/parameter\ nodes\ of\ each\ M_i.$
10: $End\ for$
11: $for\ u := C_1, C_2, C_3, \ldots, C_c$    ▷ $Where\ c\ is\ the\ total\ number\ of\ class\ nodes.$
12:   $ACCo[u] := \frac{(ACCo[u] + \sum_{i=1}^{k} ACCo[n_i])}{(k+1)}$    ▷ $n_i\ is\ the\ attribute/method\ node\ of\ class\ C_i,\ k\ is\ the\ total\ number\ of\ attribute/method\ nodes\ of\ each\ C_i.$
13: $End\ for$
14: $for\ u := P_1, P_2, P_3, \ldots, P_p$    ▷ $Where\ p\ is\ the\ total\ number\ of\ package\ nodes.$
15:   $ACCo[u] := \frac{(ACCo[u] + \sum_{i=1}^{l} ACCo[n_i])}{(l+1)}$    ▷ $n_i\ is\ the\ subpackage/class\ node\ of\ package\ P_i,\ l\ is\ the\ total\ number\ of\ subpackage/class\ nodes\ of\ each\ P_i.$
16: $End\ for$
17: $ACCo(S) = \frac{\sum_{i=1}^{|N_a|} ACCo(n_i)}{|N_a|}$    ▷ $ACCo(S)\ represents\ the\ cohesion\ of\ slice\ S$
18: Exit
```

Definition 5.1. *Cohesion of a node n is defined as the tightness of n among other nodes in ASG $G_a = (N_a, E_a)$. To measure this cohesion, we define a set Dep(n) that comprises of all those nodes on which n depends. For any node n in ASG,*

$$Inflow(n) = \{n_1, n_2, \ldots, n_k \mid \langle n_1, n_2 \rangle, \langle n_2, n_3 \rangle, \ldots, \langle n_k, n \rangle \in E_a \ \wedge \ n_1, n_2, \ldots, n_k, n \in N_a \ \wedge \ 1 \leq k \leq |N_a| - 1\}$$

The outflow of n in ASG is defined as the set comprising of all those nodes that depends on n.

$$Outflow(n) = \{n_1, n_2, \ldots, n_l \mid \langle n, n_1 \rangle, \langle n_1, n_2 \rangle, \ldots, \langle n_{l-1}, n_l \rangle \in E_a \wedge \ n_1, n_2, \ldots, n_l, n \in N_a \ \wedge \ 1 \leq l \leq |N_a| - 1\}$$

Thus, the dependence set Dep(n) of each node is defined as the union of $Inflow(n)$ and $Outflow(n)$.

$$Dep(n) = Inflow(n) \ \cup \ Outflow(n)$$

For a node n_i where $\{n_i \in N_a \mid (N_a = n_1, n_2, \ldots, n_n) \wedge (N_a, n_i) \in E_a, E_a = \{membership\ edge,\ package\ membership\ edge,\ inherited\ membership\ edge\}\}$, the cohesion value of node n_i is defined as

$$ACCo\,(n_i) = \frac{|Dep\,(n_i) \cap N_{n_i}|}{|N_{n_i}| - 1},$$

where $n_i \in N_{n_i}$ and $N_{n_i} = \{N_{n_i}, n_1, n_2, \ldots, n_k\} \subset N_a$, $1 \leq k \leq |N_a|$ and $(N_{n_i}, n_j) \in \{\overset{mem}{\rightarrow}, \overset{pkg-mem}{\rightarrow}\}$, $1 \leq j \leq k$. Thus, every program part $N_{n_i} \in N_a$ is defined as the set of nodes connected by either membership or package membership edge.

Definition 5.2. *The updated cohesion of a method node M in ASG $G_a = (N_a, E_a)$ is defined as the average of the cohesion values of all its elements (i.e. parameters and statements) along with its own cohesion. Let a method node M has j number of elements i.e. $n_1, n_2, \ldots, n_j$. Thus, cohesion of the method node M is given as*

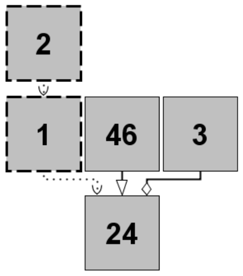
$$ACCo\,(M) = \frac{ACCo\,(M) + \sum_{i=1}^{j} ACCo\,(n_i)}{j + 1}$$

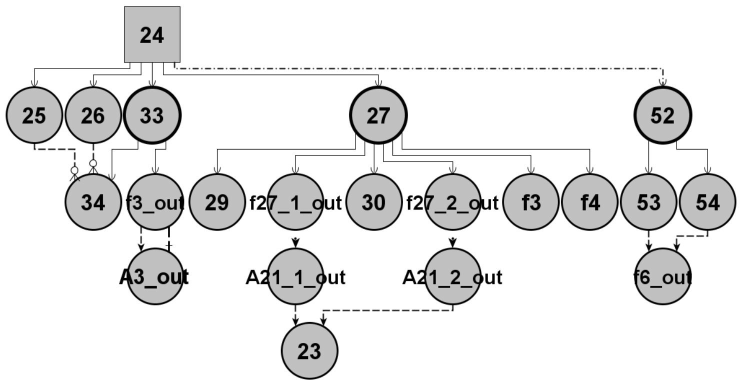
Definition 5.3. *The updated cohesion of a class node C in ASG $G_a = (N_a, E_a)$ is defined as the average of the cohesion values of all its elements(i.e. attributes, methods, and inherited members) along with its own cohesion. Let a class node C has k number of elements i.e. $n_1, n_2, \ldots, n_k$. Thus, cohesion of the class node C is given as*

$$ACCo\,(C) = \frac{ACCo\,(C) + \sum_{i=1}^{k} ACCo\,(n_i)}{k + 1}$$

Definition 5.4. *The updated cohesion of a package node P in ASG $G_a = (N_a, E_a)$ is defined as the average of the cohesion values of all its elements (i.e. classes and sub-packages) along with its own cohesion. Let a package node P has l number of elements i.e. $n_1, n_2, \ldots, n_l$. Thus, cohesion of the package node P is given as*

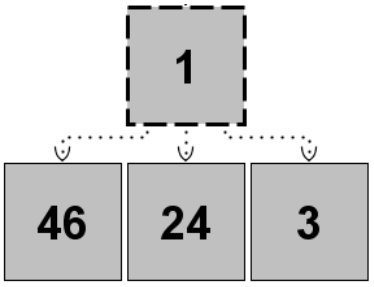
$$ACCo\,(P) = \frac{ACCo\,(P) + \sum_{i=1}^{l} ACCo\,(n_i)}{l+1}$$

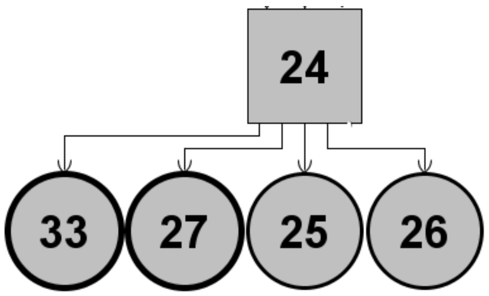




(b) Outflow set for node 24.

(a) Inflow set for node 24.





(c) The set to which node 24 is a member.

(d) The members of node 24.

Figure 5.3: ACCo computation of nodes of ASG in Figure 5.2.

Working of findACCo Algorithm

Algorithm 4 uses the formula given at Line 6 to compute the ACCo value of each node in the ASG. For example, we show the ACCo calculation for the class *Triangle* represented as *node 24* in Figure 5.2. Initially, ACCo value of *node*24 is given by

$$ACCo(24) = \frac{|Dep\,(24) \cap N_{24}|}{|N_{24}| - 1} = \frac{3}{3} = 1,$$

where $Dep(24) = inflow(24) \cup outflow(24)$ and *node* $24 \in N_{24}$, i.e. $N_{24} = \{1, 3, 24, 46\}$.

Figure 5.3 shows the sets associated with the computation of ACCo of node 24. The inflow set for node 24 is shown in Figure 5.3a and outflow set is shown in Figure 5.3b. Figure 5.3c shows the set to which node 24 is a member. The union of inflow(24) and outflow(24) is intersected with the set shown in Figure 5.3c to find the degree of relatedness of node 24. Now, to compute the ACCo of any member node of node 24, the dependence set of that node is intersected with the set shown in Figure 5.3d to find its degree of relatedness.

Similarly, the ACCo values of all the associated nodes (25, 26, 27, f3, f4, 29, 30, f27_1_out, f27_2_out, 33, f3_out, 34) of node 24 shown in Figure 5.2 are computed as follows:

$ACCo(25) = \frac{|Dep(25) \cap N_{25}|}{|N_{25}|-1} = \frac{2}{4} = 0.5$

$ACCo(26) = \frac{|Dep(26) \cap N_{26}|}{|N_{26}|-1} = \frac{2}{4} = 0.5$

$ACCo(27) = \frac{|Dep(27) \cap N_{27}|}{|N_{27}|-1} = \frac{4}{4} = 1$

$ACCo(f3) = \frac{|Dep(f3) \cap N_{f3}|}{|N_{f3}|-1} = \frac{3}{6} = 0.5$

$ACCo(f4) = \frac{|Dep(f4) \cap N_{f4}|}{|N_{f4}|-1} = \frac{3}{6} = 0.5$

$ACCo(29) = \frac{|Dep(29) \cap N_{29}|}{|N_{29}|-1} = \frac{3}{6} = 0.5$

$ACCo(30) = \frac{|Dep(30) \cap N_{30}|}{|N_{30}|-1} = \frac{3}{6} = 0.5$

$ACCo(f27\_1\_out) = \frac{|Dep(f27\_1\_out) \cap N_{f27\_1\_out}|}{|N_{f27\_1\_out}|-1} = \frac{3}{6} = 0.5$

$ACCo(f27\_2\_out) = \frac{|Dep(f27\_2\_out) \cap N_{f27\_2\_out}|}{|N_{f27\_2\_out}|-1} = \frac{3}{6} = 0.5$

$ACCo(33) = \frac{|Dep(33) \cap N_{33}|}{|N_{33}|-1} = \frac{2}{3} = 0.67$

$ACCo(f3\_out) = \frac{|Dep(f3\_out) \cap N_{f3\_out}|}{|N_{f3\_out}|-1} = \frac{2}{3} = 0.67$

$ACCo(34) = \frac{|Dep(34) \cap N_{34}|}{|N_{34}|-1} = \frac{2}{3} = 0.67$

Then, Algorithm 4 updates the ACCo value of some nodes of ASG. The reason behind this updation is that, for any node that represents a method, the statements contained inside that method also contribute to the ACCo of the method. Even if a method does not have any statement inside it, still it will have some ACCo value as some other method may be overriding it. Therefore, we have taken the average of all the ACCo values of all the statements and the ACCo value of the method under consideration, to compute the updated ACCo value of the method. For example, the ACCo values of nodes $\{24, 27, 33\}$ are updated. The

average ACCo value of $node27$ along with the ACCo values of all its member nodes $\{f3, f4, 29, 30, f27\_1\_out, f27\_2\_out\}$ are computed and assigned to $node27$, i.e.
$ACCo(27) = \frac{ACCo(27)+ACCo(f3)+ACCo(f4)+ACCo(29)+ACCo(30)+ACCo(f27\_1\_out)+ACCo(f27\_2\_out)}{7}$
$= \frac{1+0.5+0.5+0.5+0.5+0.5+0.5}{7} = 0.571$
Similarly, ACCo values of $Node33$ and $Node24$ are updated as follows:
$ACCo(33) = \frac{ACCo(33)+ACCo(f3\_out)+ACCo(34)}{3} = \frac{0.67+0.67+0.67}{3} = 0.67$
$ACCo(24) = \frac{ACCo(24)+ACCo(25)+ACCo(26)+ACCo(27)+ACCo(33)}{5} = \frac{1+0.5+0.5+0.571+0.67}{5}$
$= 0.6482$
Therefore, ACCo value of *class Triangle* in Figure 4.2 represented as $node24$ in Figure 5.2 is 0.6482. Similar procedure is followed to update the ACCo values of all the nodes representing the classes and packages within ASG.

Theoretical Validation

In this section, we provide the theoretical soundness of the proposed measure, i.e. this approach satisfies the four basic properties for cohesion measure as suggested by Briand et al. [31]. Even though these four basic properties are not sufficient to characterize the proposed cohesion measure in a rigorous manner, but these properties are necessary to prove the correctness [6] of any cohesion measurement approach. There exists many more validation frameworks [204] in the literature, but we validate our approach with Briand's framework and have left other frameworks for future study.
The cohesion properties of Briand's framework are as follows:
Property 1: *Non-Negativity and Normalization*
This property states that the value of cohesion given by the proposed measure should be non-negative and normalized, i.e. $0 \leq ACCo\,(n) \leq 1$, for any node n in G_a.
Proof. For any node n in G_a, $Dep(n) \cap N_n \subseteq N_a \Rightarrow 0 \leq \frac{|Dep(n) \cap N_n|}{|N_n|-1} \leq 1$. That is if $Dep(n) \cap N_n = \phi \Rightarrow |Dep\,(n) \cap N_n| = 0 \Rightarrow ACCo(n) = 0$. And if $Dep(n) \cap N_n = N_n - \{n\} \Rightarrow |Dep\,(n) \cap N_n| = |N_n| - 1 \Rightarrow ACCo(n) = 1$. Therefore, $0 \leq ACCo\,(n) \leq 1$ will always hold true.

Property 2: *Null Value and Maximum Value*
This property states that for any node $n \in G_a$, if $|N_a| > 1$, then $E_a = \phi \Rightarrow ACCo\,(n) = 0$, and E_a is maximal $\Rightarrow ACCo\,(n) = 1$.
Proof. For any node n in G_a, if $E_a = \phi$ then $Dep(n) = \phi \Rightarrow ACCo(n) = 0$. If $Dep(n) \cap N_n = N_n$, then $Dep(n)$ contains all the nodes in N_n. Hence, $ACCo(n) = 1$.

Property 3: *Monotonicity*

This property states that addition of a new relationship (edge) to the ASG must not decrease the cohesion value of any node, i.e. if $G_{a2} = (N_{a2}, E_{a2})$ is obtained by adding an edge $< n1, n2 >$ to $G_{a1} = (N_{a1}, E_{a1})$, then $ACCo_1(n) \le ACCo_2(n)$, where $ACCo_1(n)$ and $ACCo_2(n)$ are the cohesion values of node n in G_{a1} and G_{a2}, respectively.

Proof. We use $Dep_1(n)$, $Dep_2(n)$ to represent the dependence set of any node n in G_{a1} and G_{a2}, respectively.

Case 1: If $n_1 \in Dep_1(n) \wedge n_2 \in Dep_1(n) \wedge n_1, n_2 \in N_n$, then adding $< n_1, n_2 >$ will not change $Dep_2(n) \cap N_n$. Hence, $Dep_1(n) \cap N_n = Dep_2(n) \cap N_n$. Therefore, $ACCo_1(n) = ACCo_2(n)$.

Case 2: If $n_1 \in Dep_1(n) \wedge n_2 \notin Dep_1(n) \wedge n_1, n_2 \in N_n$, then adding $< n_1, n_2 >$ will result in $|Dep_1(n) \cap N_n| + 1 = |Dep_2(n) \cap N_n{}'|$ and $|N_n| = |N_n{}'|$. Hence, $\frac{|Dep_1(n) \cap N_n|}{|N_n|-1} < \frac{|Dep_2(n) \cap N_n{}'|}{|N_n{}'|-1} \Rightarrow ACCo_1(n) < ACCo_2(n)$.

Case 3: If both $n_1, n_2 \notin N_n$ or any one of $n_1, n_2 \notin N_n$, then adding $< n_1, n_2 >$ will not change $Dep_2(n) \cap N_n$. Hence, $Dep_1(n) \cap N_n = Dep_2(n) \cap N_n$. Therefore, $ACCo_1(n) = ACCo_2(n)$.

Therefore, adding an edge does not increase the cohesion of any node in G_a.

Property 4: *Merging of Unconnected Modules*

This property states that when two sets of unconnected nodes N_1 and N_2 are merged to form a single set of nodes N such that $N = N_1 \cup N_2$ in G_a, then this should not increase the cohesion value of any node n, i.e. $max\{ACCo_1(n), ACCo_2(n)\} \ge ACCo(n)$, where $ACCo_1(n)$, $ACCo_2(n)$ and $ACCo(n)$ are the cohesion values of a node n in N_1, N_2 and N, respectively.

Proof. Let N_1 and N_2 be two sets of nodes such that $N_1 \cap N_2 = \phi$ and $n \in N_1$. For any node n, $ACCo_1(n)$ and $ACCo_2(n)$ may represent the cohesion values before and after merging of nodes, respectively. Let $Dep_1(n)$, $Dep_2(n)$ be the dependence sets before and after merging of N_1 and N_2, then $Dep_1(n) = Dep_2(n)$ as $N_1 \cap N_2 = \phi$. Supposing, $N' = N_1 \cup N_2$. Then, $\frac{|Dep_1(n) \cap N_1|}{|N_1|-1} > \frac{\left|Dep_2(n) \cap N'\right|}{\left|N'\right|-1}$ because $\left|N'\right| > |N_1|$ $\implies ACCo_2(n) < ACCo_1(n)$.

Therefore, it proves that merging two independent sets of nodes will not increase the cohesion value.

5.2.5 Modeling test suite minimization as binary ILP problem

Our technique for getting efficient minimized test-suite is based on the idea of enhancing the fault-detection capability of the test-suite while maintaining the same coverage of statements as the original test-suite under consideration. We consider the coverage of the affected nodes instead of the statements as our approach is based on the intermediate graphical representation of the impacted program parts, ASG. Here, the constraint is to maintain the original coverage of the affected nodes and the objective function is to minimize the total ACCo value of the nodes covered by the test cases. The reason for minimizing the sum of ACCo values is that any test case for which the sum is minimum denotes that test case executes those nodes that have a small cohesion value. The hypothesis is that nodes with smaller cohesion value are more error-prone [8, 10, 167, 211], and the test case that executes more number of such nodes has the maximum chance of finding errors.

The test-suite minimization problem may be viewed as expressing it as an ILP problem in terms of the objective function and a set of constraints. Given that these constraints are identified and the objective function is formulated, ILP guarantees an optimal solution to a mathematical problem. We refer to Table 5.1 for representing the test suite minimization problem as an ILP problem. In the formulation of the problem, we focus to maintain the coverage of all the affected nodes and to minimize the sum of the cohesion values to maximize the fault detection with minimum test cases. Below, we define the terminologies required for formulating the proposed minimization problem:

- **The Set N:** N represents the set of all the affected nodes n_i with respect to the change.

- **The set S:** For each test case $t_j \in ST$, there is some $s_j \in S$ that corresponds to the set of affected nodes covered by t_j (refer third column of Table 5.1).

- **Decision variables:** Each test case $t_j \in ST$ is represented by a decision variable b. The value of b is equal to 1 if test case t_j is included in the cover, and is 0 otherwise.

- **Objective function:** It is used to minimize the number of test cases needed to cover all the affected nodes and to minimize the sum of cohesion values of the nodes covered by the minimized test cases. c_j refers to the values shown in the last column of Table 5.1.

- **Constraints:** Each constraint refers to each node of ASG, i.e. it ensures that each affected node is covered by a test case in MT.

- **Constraint coefficient matrix:** The constraint coefficient matrix, X, is a matrix with N rows, one for each affected node, and S columns, one for each test case. Each element $x_{i,j}$ of this matrix is 1 if test case t_j covers affected node n_i, and is 0 otherwise.

Considering all these above concerns, the complete test suite minimization problem is represented in ILP as follows:

$$Minimize : \sum_{j=1}^{|S|} c_j b_j$$

Subject to:

$$\sum_{j=1}^{|S|} x_{i,j} b_j \geqslant 1, \ i = 1, \ldots, |N|, \ where \ b_j \ binary \ value \ for \ j = 1, \ldots, |S|.$$

Thus, the ILP encoding of our motivating example is given in Figure 5.4. Line 1 shows the five binary variables corresponding to the five test cases in Table 5.1. Line 2 defines the objective function for minimization which is weighted by the sum of cohesion values (refer fourth column of Table 5.1). Lines 3 - 33 define the constraints to achieve 100% affected-nodes coverage. Thus, Line 33 clearly shows that the value of $b5$ has to be 1 to achieve 100% coverage by the minimized test suite.

Once the encoding part is complete, any standard solver can be used to solve the given ILP problem. We solved our stated ILP problem using CPLEX, a software package for linear, network, and integer programming. Though CPLEX ia a commercial platform, it is available for free with restricted features for academic and research. The *minimized test-suite (MT)* given by this process for the test data in Table 5.1 is $MT = \{T8, T10\}$. Figure 5.5 shows the percentage of faults detected by ST and MT. The rectangles in Figure 5.5 show the percentage of faults detected by the test cases of MT and the diamonds represent the percentage of faults revealed by ST. The number of rectangles or diamonds in the figure denotes the number of test cases in the corresponding test suite. It is evident that even after 60% minimization of ST, the percentage of faults detected by MT is comparable with that of ST. The graph implies that within constrained budget and time, the testers can still ensure good quality of the software with MT considering 100% fault detection is never possible in a real scenario.

```
1.   bin: b1, b2, b3, b4, b5;
2.   min: 0.65b1 + 9.78b2 + 6.86b3 + 6.64b4 + 8.79b5;
3.   1:       b1  + b2  + b3  + b4  + b5  >=  1;
4.   2:       b1  + b2  + b3  + b4  + b5  >=  1;
5.   3:       b1  +         b3  + b4  + b5  >=  1;
6.   4:       b1  +         b3  + b4  + b5  >=  1;
7.   6:       b1  +         b3  + b4          >=  1;
.
.
.
31.  f3_out:       b2  +         b4  + b5  >=  1;
32.  f4:           b2  + b3  + b4          >=  1;
33.  f6_out:                         b5  >=  1;
```

Figure 5.4: ILP encoding of the test data given in Table 5.1.

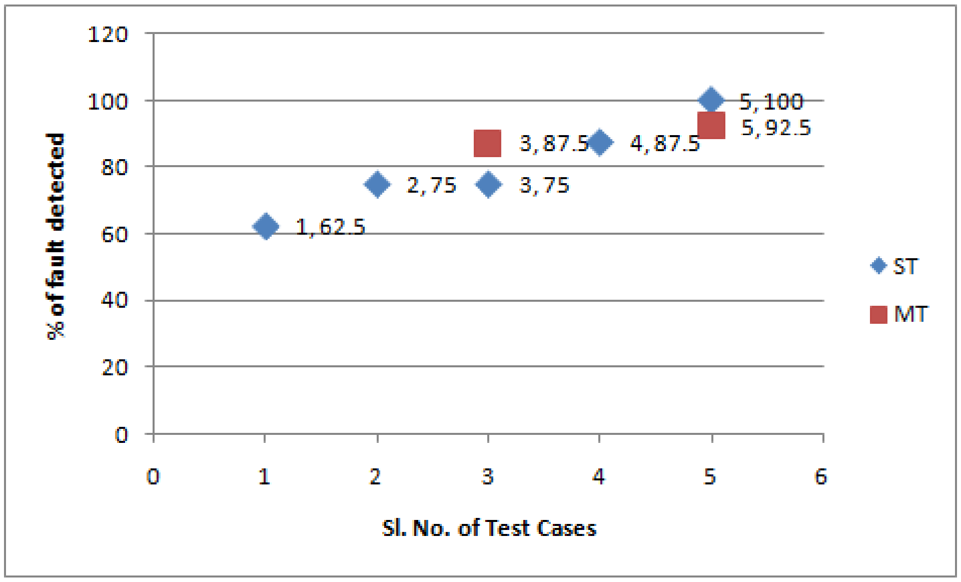


Figure 5.5: % of fault detected by ST and MT.

5.3 Experimental study

After demonstrating the effectiveness and usefulness of the proposed approach (refer Figure 5.5) for the example program given in Figure 4.2, we are posed with the following research questions (RQ):

RQ1: Effectiveness. Does the minimized test suite actually guarantee acceptable

Table 5.2: Comparison of our proposed change-based cohesion metric with different existing approaches.

Comparison Features		LCOM	TLCOM	RCI	CBMC	DRC	ACCo
Excluded special Methods	Constructor	No	No	Yes	Yes	Yes	No
	Destructor	No	No	No	Yes	Yes	Yes
	Access	No	No	Yes	Yes	Yes	Yes
	Delegation	No	No	No	Yes	Yes	Yes
Briand's Properties	Property 1	No	No	Yes	Yes	Yes	Yes
	Property 2	Yes	Yes	Yes	Yes	Yes	Yes
	Property 3	No	No	Yes	No	Yes	Yes
	Property 4	Yes	Yes	Yes	Yes	Yes	Yes
Transitive dependency		No	Yes	No	No	Yes	Yes
Inheritance		No	No	No	No	No	Yes
Interface		No	No	No	No	No	Yes
Polymorphism		No	No	No	No	No	Yes
Templates		No	No	No	No	No	Yes

quality (detection of faults) as compared to original test suite for all the experimental programs?

RQ2: **Usefulness**. Is it feasible to generate the minimized test suite within acceptable time limits?

A change set is maintained that refers to the set of concurrent changes carried out on the program. The test cases for the input program are generated using Junit Eclipse plugin [1]. To find the fault detection capability of the test cases, the program was seeded with mutants. To generate the mutants for the input program, we used MuClipse. MuClipse [183] is the Eclipse plugin version of μJava that generates two types of mutation operators both for traditional mutation and class mutation. We have considered both types of mutations in our approach. Smith et al. [183] and Do et al. [60] have carried out an extensive empirical study and justified mutants as good proxy of real faults.

[1]http://www.tutorialspoint.com/junit/junit_plug_with_eclipse.htm

Table 5.3: Test-suite minimization result of different programs.

Sl. No.	Programs	LOC	Avg. # of Affected Nodes	Total # of Test Cases	# Mutants	Selected Test Suite		Minimized Test Suite	
						% of selected test cases	% of faults detected	% of minimized test cases	% of faults detected
1	Expt. Program	54	33	20	14	25	100	54.8	91.6
2	Calculator	75	51	15	42	46.7	94	57.1	90.8
3	Elevator	90	54	25	27	40	98	57.2	92.1
4	Stack	114	72	22	35	40.9	96	57.3	92.6
5	Sorting	130	86	16	43	31.3	89	51.5	92.6
6	BST	130	74	20	51	60	100	54.5	90
7	CrC	261	94	18	46	33.3	93	52.8	91.5
8	DLL	277	83	24	47	25	98	52.9	89.4
9	Notepad	300	68	17	17	47.1	89	51.8	86.3
10	ATM	900	97	33	39	36.4	97	54.2	91.8
11	Elevator_spl	1046	105	15	53	66.7	97	54.2	91.3
12	Email_spl	1233	98	18	18	61.1	100	50.3	94.8
13	GPL_spl	1713	112	22	22	63.6	94	53.5	92.2
14	Jtopas	5400	241	16	28	56.3	92	59	88.2
15	Nanoxml	7646	544	14	32	50	95	50.4	91.6

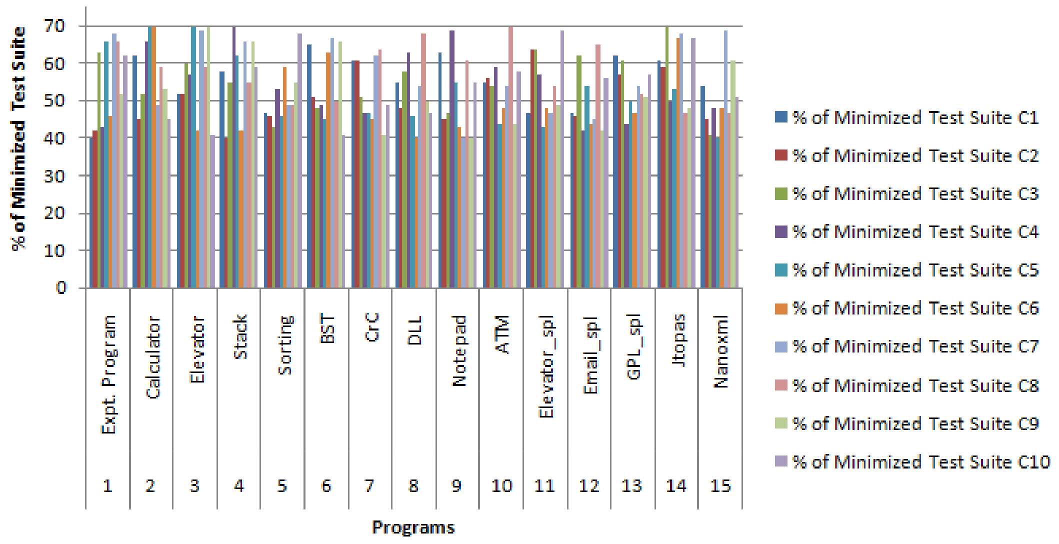


Figure 5.6: Test suite minimization results for all the ten changes made to the program.

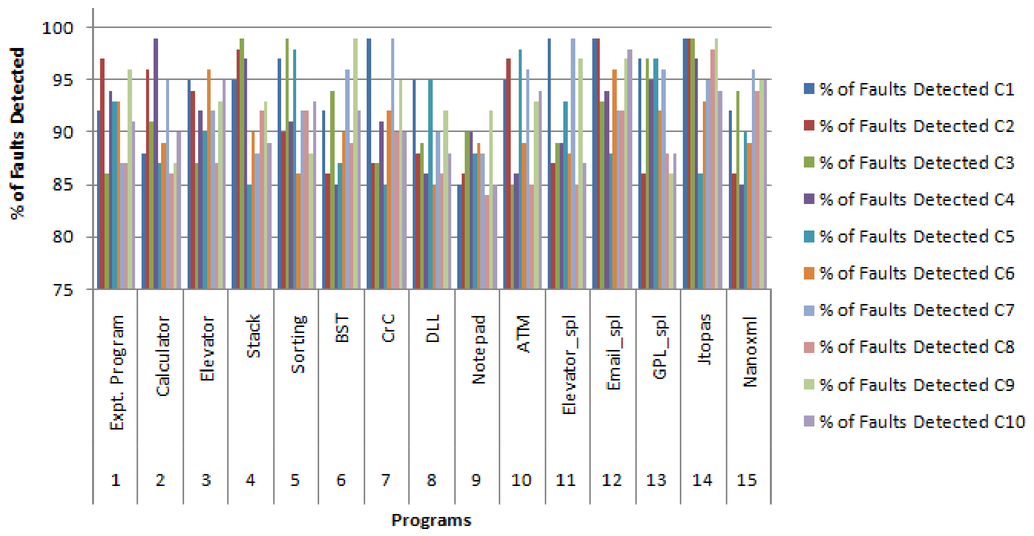


Figure 5.7: Fault detection results of the minimized test suite for all the ten changes made to the program.

5.3.1 RQ1: Effectiveness

To represent the minimization problems, we computed the affected statements with respect to every change made to the experimental programs and computed their affected component cohesion values as discussed in Section 5.2.4. We also theoretically validated our ACCo metric in Section 5.2.4. In Table 5.2, we compare our proposed ACCo metric with some of existing metrics. In Table 5.2, it can be observed that the approaches such as LCOM [47], TLCOM [7] and CBMC [40] fail to satisfy all the four basic properties of cohesion [31], whereas RCI [32], DRC [218] and ACCo satisfies all the properties. Among these three approaches that satisfy Briand's properties only DRC and ACCo consider transitive dependency to compute the cohesion. In addition to the transitive dependency among program parts, our proposed ACCo approach considers the impact of inheritance and other object-oriented features (such as interface, polymorphism, and templates) on the cohesion measurement. Thus, ACCo metric gives a better cohesion result than DRC.

A total of ten changes are made to each program and slices are computed for every change made to the programs. The total number of computed slices for all the fifteen programs is 150. These slices are used to access the impact of change and select the regression test cases. Table 5.3 shows the initial percentage of the selected test cases and the result of seeded fault detection for every experimental program. Then, we computed and compared the effectiveness of the minimized test suite with the selected test suite. The percentage of minimized test suite and the percentage of faults detected by the minimized test suite are shown in Table 5.3. The proposed minimization approach achieved an overall test suite minimization of 54% approximately for all the fifteen programs. It is evident from Table 5.3 that the minimized test suite revealed approximately 91% of the faults as compared to 95% by the selected test suite, which is quite acceptable. Figure 5.6 shows the percentage of test-suite minimization achieved with respect to all the ten changes made to each program. These changes are summarized in Table 4.3. Figure 5.7 shows the percentage of faults detected by the minimized test suite with respect to all the ten changes made to the individual programs. Thus, our results confirm that the proposed test suite minimization approach is effective in minimizing the selected test-suite and can reveal most of the faults to ensure the quality of the software.

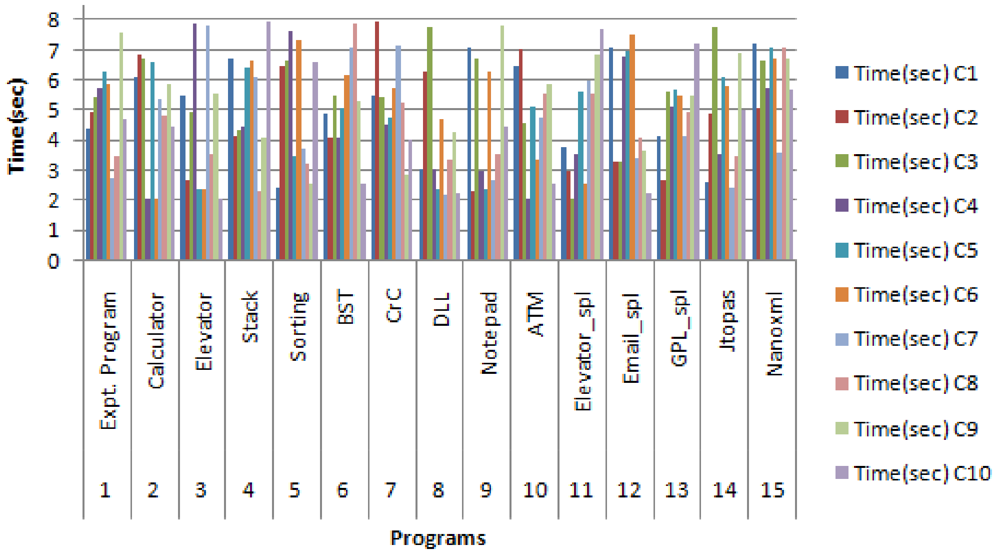


Figure 5.8: Timing results of the minimized test suite for all the ten changes made to the program.

5.3.2 RQ2: Usefulness

This research question addresses our concern that whether the proposed approach can generate the minimized test suite in a reasonable amount of time. The usefulness of the proposed approach is shown in terms of time taken to generate the minimized test suite. It is observed that the proposed approach can generate the minimized test suite in less than 1 second for all the changes made to the programs, provided the selected test suite and their coverage information are available before computation. The timing results in Figure 5.8 show the time taken to minimize the test suite for every change made to the programs. This result includes the time to compute the slices, select the test cases hierarchically, compute the cohesion values with respect to the change impact analysis, and the time to minimize the selected test suite. The percentage of minimization achieved is shown in Figure 5.6. However, the timing results would improve when we fully integrate the different components of our proposed minimization framework shown in Figure 5.1. Thus, the results show that the proposed approach is very useful and scale better if the requisite test data is collected during the initial testing of the software.

5.3.3 Threats to validity

Like many other techniques on minimization, the proposed approach also has some threats to its validity.

- All the programs considered for experimentation represent various domains of application. However, real industrial applications can be huge in size and complexity as compared to the chosen programs.
- Intermediate graph-based slicing techniques can suffer scalability issues. To overcome this limitation to some extent, we restricted our regression test selection to method level only. Hence, the size of the selected test suite are much less at finer granularity of test case selection. As a result, this could have lessen the time of minimization.
- The proposed minimization problem is formulated based on the cohesion measure given in Section 5.2.4. However, many other researchers have proposed various cohesion measures. Thus, the ILP problem may yield different results if the cohesion measure of other researchers are taken into consideration.
- The mutants generated by MuJava sometimes may not represent the real-faults of industrial applications. Thus, to remain close to the real-faults, we asked our graduate and post-graduate students to seed the errors. This may have resulted in some biasness in seeding the errors. Therefore, we considered only those test cases that gave high coverage of these faults.
- Since minimization problems are NP-complete, we focused on a single criterion for minimizing our test suite. However, considering other criteria for

minimization such as coupling measure of affected components, time for fault detection, energy utilization of the test cases, etc. may give some interesting results. Research outputs of such multi-criteria minimization problems are not addressed in this chapter.

5.4 Comparison with related work

The work in [131] motivates the proposed work presented in this chapter to use integer linear programming for test suite minimization. Minimization techniques focus on selecting a minimum number of test cases that satisfy a given criterion. According to Li et al. [131], the proposed minimization technique selects those test cases in less than 1 second that consume 95% less energy and maintains the coverage of testing requirements. However, the minimization time does not include the time required to pre-compute the test data (such as energy consumption and coverage information). Whereas, the timing results of our experimental studies include the time required to compute all the testing requirements. Unlike minimizing the energy consumption, our work focuses on executing minimum test cases to achieve comparable level of fault detection. Like our proposed approach, the techniques discussed by Yoo and Harman [210] provided an elaborate, recent study on the available techniques for test suite minimization, selection, and prioritization, and are designed for regression testing. However, our approach introduces a new idea of using cohesion measure of the affected program parts as the limiting criteria to minimize the test suite for regression testing.

The results of the empirical study to investigate the limitations of single criterion minimization techniques cariied out by Rothermel et al. [174] and Wong et al. [208] concludes that single criterion-based minimization techniques comparatively detected fewer faults as compared to the original test suite considered. The minimization technique focused on fault detection capability of the test cases. The experimental results of our proposed work also confirm with the results of [174, 208]. However, our results show that under constrained conditions of time, our minimized test suite gives acceptable performance in terms of fault detection. Jeffrey and Gupta [105] considered multiple sets of testing requirements (e.g. coverage of different entities) to overcome the limitations of single-criterion minimization techniques. The results in [105] had shown an improvement over the existing techniques. In our approach, we also considered similar coverage criterion. But, instead of seeking complete program coverage, our approach rather focuses on achieving full coverage of the program parts affected by the changes made to the programs. A two-criteria variant of test suite minimization technique by Black et al. [28] computed optimal result using an integer linear programming solver. The technique considered both definition-use association coverage and the ability of test cases to reveal errors. The results show that the error revealing ability of the test cases measured with respect to a collection of program faults helped in revealing other program faults. As shown in our results, the proposed minimization approach also focuses on the fault detection capability of the test cases. The test suites are minimized with respect to the mutation faults that are accepted as good measures of the real faults [60, 183].

5.5 Summary

In this work, we have introduced a new approach of using cohesion measures of the affected program parts to minimize the test suite for regression testing. We formulated the minimization problem in integer linear programming and obtained an optimal minimized test suite. The results of our studies show that the minimized test suite is both effective and useful for regression testing. This approach enables the testers to decide on the magic number of test cases to choose that would ensure acceptable quality, especially during scenarios of constrained budget and time for regression testing. Even though test suite minimization approach removes the redundant test cases, but it does not lay any focus on the fault revealing capabilities of the test cases. It is observed that some test cases reveal faults early during the testing process than others. Finding an optimal order of execution of the test cases will enhance the chances of detecting more errors early. So, in the next chapter, we focus on finding an optimal order of test case execution through prioritization.

Chapter 6

Regression Test Case Prioritization

In this chapter, we focus on Test Case Prioritization (TCP) of a given test suite T to address the problem of regression testing. Test case prioritization focuses on reordering the sequence of execution of test cases [66, 69, 104, 108, 175, 184]. The sequencing of the test cases in a given test suite is done based on some established criteria. The test cases having higher priority are executed earlier than the test cases with lower priority. Many researchers [66, 104, 108, 132, 142, 148, 165] have proposed different approaches to prioritize the test cases. All these approaches target to find an optimal ordering of the test cases based on the rate of fault detection or rate of satisfiability of coverage criterion under consideration. These techniques have evolved mainly to improve the effectiveness of regression testing and/or to reduce the cost of test case execution. This prioritization approach can be used with the selective retest technique to obtain a version specific prioritized test suite [41]. The required steps are as follows [41]:

i. Select $T^{'}$ from T, a set of test cases to execute on $P^{'}$.

ii. Find $T^{'}_{P}$, a permutation of $T^{'}$, such that $T^{'}_{P}$ will have a better rate of fault detection than $T^{'}$.

iii. Test $P^{'}$ with $T^{'}_{P}$ in order to establish the correctness of $P^{'}$ with respect to $T^{'}_{P}$.

iv. If necessary, create $T^{''}$, a set of new functional or structural test cases for $P^{''}$.

v. Test $P^{'}$ with $T^{''}$ in order to establish the correctness of $P^{'}$ with respect to $T^{''}$.

vi. Create $T^{'''}$, a new test suite for $P^{'}$, from T, $T^{'}$ and $T^{''}$.

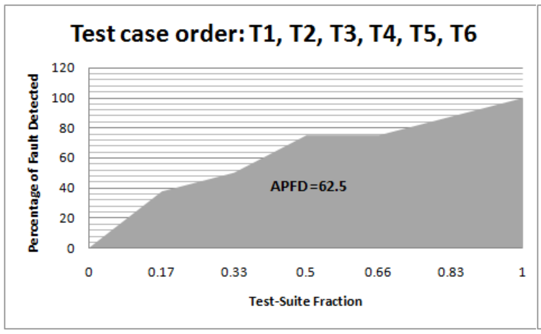
Table 6.1: A sample test case distribution and the faults detected by them.

$TestCases/Faults$	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$
$f1$	X				X	X
$f2$			X	X		X
$f3$	X		X		X	
$f4$			X			X
$f5$		X		X		
$f6$					X	X
$f7$						X
$f8$	X					
$No.\ of\ faults$	3	1	3	2	3	5
% of faults detected by two sample test case orderings.						
$T1, T2, T3, T4, T5, T6$	37.5	50	75.0	75.0	87.5	100
$T1, T3, T2, T6, T5, T4$	37.5	62.5	75.0	100	100	100

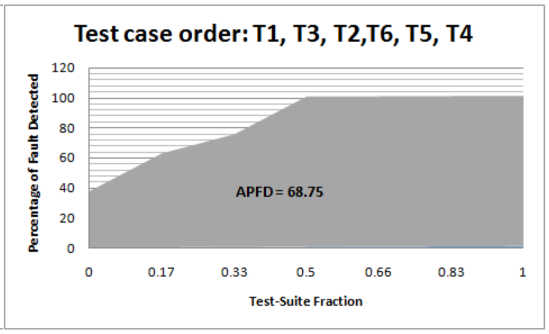
The rest of the chapter is organized as follows: In Section 6.1, we provide the motivation of this chapter through an example and set our objectives. Section 6.2 discusses issues related to coupling measure in object-oriented programs. In this section, we present our proposed coupling measurement approach. We prove the correctness of the proposed approach by theoretically validating it as per the established guidelines and properties. In Section 6.3, we discuss our proposed prioritization approach that is based on the estimation of the coupling factor, and the weight assigned to each node in the Affected Slice Graph (ASG). Section 6.4 discusses a case study and shows the working of our proposed algorithms. In Section 6.5, we provide the correctness proof of our algorithms. The complexity analysis of the proposes algorithms is given in Section 6.6. In Section 6.7, we discuss the implementation of this proposed approach and compare our work with some existing related work by other researchers in Section 6.8. Finally, we summarize the chapter in Section 6.9.

6.1 Motivation

Test case prioritization problem is best described using the example in Table 6.1. The 'X' mark in the cells of the table represents that the particular test case in the column reveals the presence of the corresponding fault in the row. Supposing the test cases are executed in the order $\{T1, T2, T3, T4, T5, T6\}$, then we achieve 100%



(a) APFD measure for the first test case ordering.



(b) APFD measure for the second test case ordering.

Figure 6.1: APFD measure for the test case orderings in Table 6.1.

coverage of faults only after the sixth test case is executed. Whereas, if the ordering of the test case execution is changed to {T1, T3, T2, T6, T5, T4}, then we achieve 100% coverage after the execution of the fourth test case. Therefore, finding the order of test case execution is essential to detect the faults early during regression testing. All the existing approaches [66, 69, 104, 108, 162, 163, 175, 184] target to find an optimal ordering of the test cases based on the rate of fault detection or rate of satisfiability of coverage criterion under consideration. Rothermel et al. [175] proposed a metric to ensure the efficiency of any of the existing prioritizing techniques. This metric is called *Average Percentage of Fault Detected (APFD)* and is used by many researchers to evaluate the effectiveness of their proposed techniques. APFD measure is calculated by taking the weighted average of the number of faults detected during execution of a program with respect to the percentage of test cases executed. A sample distribution of the test cases and the number of faults detected by them are shown in Table 6.1.

Let T be a test suite and $T^{'}$ be a permutation of T. The APFD for $T^{'}$ is defined as follows:

$APFD = 1 - \frac{\sum_{i=1}^{n-1} F_i}{n*l} + \frac{1}{2n}$

Here, n is the number of test cases in T, l is the total number of faults, and F_i is the position of the first test case that reveals the fault i.

The value of APFD can range from 0 to 1, but it is shown in percentage. Higher is the APFD value for any ordering of the test cases in the test suite, higher is the rate at which software faults are discovered [60, 175]. APFD measures for the two test case orderings given in Table 6.1 are shown in Figure 6.1a and Figure 6.1b.

The existing techniques [66, 69, 104, 108, 175, 184] were primarily developed to target procedural programs. Therefore, these techniques are hardly proven to be efficient when applied to object-oriented programs. In case of object-oriented programs [38, 49], the programming complexity shifts from method interaction to object relations and interaction among objects. Very few existing work [162, 163] focus on the test case prioritization for object-oriented programs. The different dependences present in an object-oriented program need to be considered in order to find an efficient order of the test cases. This is because, these dependences between distinct program parts affect the behavior of other parts in the program in the context of some modification done to some part of the program. In this context, it is needed to make a thorough analysis of the dependences between different programming constructs and to detect the critical parts of the programs.

To identify the interactions and dependences among the program constructs, it is essential to graphically model the program under test through some intermediate graph representation [119, 152, 198]. From this intermediate graph representation, named *affected slice graph (ASG)*, the nodes that directly affect or get affected with respect to some modification done to the program are sliced. The process of constructing an ASG has been discussed in Section 5.2.3. The critical nodes in ASG that have a high probability of being erroneous are then determined by estimating their change-based coupling values. We calculate the *change-based coupling* of each node in the ASG to estimate its criticality based on fault proneness. Therefore, any test case which covers these critical nodes has a higher chance to uncover the error(s) early in the testing process and hence is given more priority than other test cases in the test suite.

Based on the above motivations, we propose an approach to prioritize the test cases present in a given test suite using the *affected component coupling (ACC)* value of each node ASG that are covered by the test cases. We fix our research objectives as follows:

i. To develop a mechanism to compute the ACC value of each node in the obtained affected slice graph (ASG).

ii. To cluster the ACC values [197] into groups based on their criticality and to assign a weight [150] against each group such that, a node having a higher ACC value will get more weight in comparison to a node having lower ACC value.

iii. To assign a weight to each test case t_i in the given regression test suite T based on the total weight of all the nodes that are covered by t_i.

iv. To obtain the prioritized test suite by sorting all the selected test cases in the decreasing order of their computed weights.

6.2 Coupling in Object-Oriented Programs

Coupling is a software metric [99, 147] that gives a measure which signifies how one module depends upon or affects the behavior of another module. It is reported that a module having high coupling value is more erroneous than other modules [112]. This is because software defects are more often the result of incomplete or incorrect comprehension of a program segment. Therefore, locating such a program segment that posses a challenge to comprehension is essential and is represented by the coupling measure of the program segment. As a result, a test case that executes a module with high coupling value will reveal more faults than other test cases. Many techniques exist to measure the coupling value of the program segments [15, 30, 89, 99, 128, 150]. Among all the coupling metrics [30, 65, 150, 171], *export coupling metric* has the strongest association with fault proneness [65] in an object-oriented program.

A study on a C++ telecommunication system [29] also establishes the association of export coupling with fault proneness.
The two hypotheses presented in support of the above claim are as follows:

i. Classes with high export coupling values are used more frequently than other classes. This is because, more the number of out going dependences imply that more classes and methods are using or dependent on it. Therefore, even if all the classes have the same number of faults in them, more faults are detected in a class with high export coupling value[65].

ii. The class with high export coupling value acts as a server and other classes using it are its clients. Therefore, a client of class D makes usage of D's behavior. A class with higher export coupling value has more clients and hence more times the methods are used. Since the number of usages can be quite large, it is more likely that this class D will have a subtle fault that migrates to the clients. This migration of fault makes testing difficult [65].

In procedure-oriented programs, two modules are said to be coupled if they interchange data among them during function calls or if the interaction occurs through some shared data. In these circumstances, the modules are said to be tightly coupled. Coupling gives the complexity of a module. Slicing based approaches [89, 150] can be used to measure how one module affects another module in a traditional software system. Henry and Kafura [97] developed an approach to measure coupling based on information flow. Harman et al. [89] used program slicing to measure coupling in traditional software systems. The different types of coupling [64] that can exist between any two modules m1 and m2 are data coupling, stamp coupling, control coupling, common coupling, and content coupling. The details and definitions of these types of couling can be found in [64].

i. *Data coupling*: Data coupling exists between m1 and m2, if m1 passes some elementary data as parameters to communicate with m2.

ii. *Stamp coupling*: Stamp coupling exists between m1 and m2, if m1 passes some composite data item as a parameter to communicate with m2.

iii. *Control coupling*: Control coupling exists between m1 and m2, if the data from m1 order the execution of instruction in m2.

iv. *Common coupling*: Modules m1 and m2 have common coupling if they share some global data items.

v. *Content coupling*: Content coupling exists between modules m1 and m2, if they share code.

However, in an object-oriented programming environment, coupling can exist not only at the level of methods but also at the class level and package level. Therefore, coupling represents the degree of interdependence between methods, between classes and between packages, etc. Many researchers have proposed different slicing based mechanisms [15, 30, 128] to measure coupling in an object-oriented framework. There exists three different frameworks to measure coupling factors in an object-oriented paradigm.

Eder et al. [64] first proposed three different types of relationships among program components that contribute to coupling. The three different relationships are: interaction relationship between the methods, component relationship between classes and inheritance between classes. These relationships are used to derive different dimensions of coupling, which are then classified according to different strengths. Hitz and Montazeri [99] next proposed methods to measure coupling at object level and at the class level for an object-oriented program. The object level coupling is determined by the state of an object. Then, the class level coupling is determined by the state of the object implementation. They also proposed different strengths of the coupling measurement. Briand et al. [30] computed coupling as the measure of the interactions between the classes of an object-oriented program.

The coupling strength is determined by the type of interaction, frequency of interaction between the classes and the locus of impact of the interaction. In all the previous existing frameworks [30, 64, 99], coupling measurement has been considered as interactions at method and class levels only. Our approach is a combination of all the three proposed frameworks for object-oriented programs. The comparison of our mechanism with some previous mechanisms [30, 64, 99] that measure coupling is shown in Table 6.2. From Table 6.2, it can be observed that our approach incorporates all the features suggested in previous frameworks. Our mechanism does not consider the feature "pointers to methods," as the concept of pointers is not explicitly present in Java programs. In our technique, we extend the existing frameworks by computing coupling at all the hierarchical levels, i.e. at statement level, method level, class level and package level.

There are many factors such as information hiding, encapsulation, inheritance, message passing and abstraction mechanisms, that contribute to coupling in object-oriented programs. High coupling affects program comprehension and analysis. As a result, it becomes very difficult to maintain software systems. In an object-oriented program, coupling can exist between any two components due to message passing, polymorphism and inheritance mechanisms of object-oriented programs. These components include packages, classes, methods and statements. Two statements s1 and s2 are said to be coupled if s1 has some dependence (control, data or type dependence) on s2. Methods in an object-oriented program belong to the constituent classes. It implies that a method is either coupled with a method in the same class or with another method in a different class. Thus, coupling between two methods m1 and m2 in an object-oriented program can exist in the following situations:

i. when m1 invokes m2 by passing some data as parameters.

ii. when m1 depends on the data returned by m2.

iii. when m1 shares some global data with m2.

If the methods of any two classes are coupled, then the corresponding classes are said to be coupled. Similarly, the container packages of the coupled classes are also said to be coupled.

Table 6.2: Comparison with mechanisms that measure coupling.

Sl. No.	Properties	Eder et al. [64]	Hitz & Montazeri [99]	Briand et al. [30]	Our approach
1	Methods share data	X			X
2	Method references attribute		X		X
3	Method invokes method	X	X	X	X
4	Method receives pointer to method			X	
5	Class is type of a class attribute	X	X	X	X
6	class is type of a method's parameter or return type	X	X	X	X
7	class is type of a method's local variable	X	X		X
8	class is type of a parameter of a method invoked from within another method	X			X
9	Class is ancestor of another class	X	X		X
10	Template class				X

6.2.1 Affected Component Coupling (ACC)

Harman et al. [89] used slicing technology to compute inflow and outflow of a node, as an application of slicing to coupling. In their approach, the inflow into a module m is calculated by identifying the definitions of variables that are outside m but contained in the slice. Similarly, outflow of m is computed by identifying the program parts outside m, whose slice includes a variable of m. Harman et al. [89] again observed that presence of a single node n in the slice that is outside the body of m indicates an information flow either from m to n or vice-versa. We represent these information flow (inflow & outflow) between any two nodes in the form of edges in ASG. For any node n in ASG, we compute the inflow to n by counting the number of nodes traversed in the backward direction from node n. Similarly, outflow is computed by counting the number of nodes traversed in the forward direction from node n. Coupling of a node n in ASG $G_a = (N_a, E_a)$ is defined as the amount of inflow and outflow of n among other nodes. Below, we define the terms related to the computation of proposed export coupling factor.

Definition 6.1. *Affected component coupling of a given node n is computed as the normalized ratio of dependence of n, $\psi(n)$, to the total number of affected nodes in* *the ASG, $|N_a| - 1$, as the node under consideration is excluded. This coupling is measured with respect to the change made to the program that was taken as slicing criterion to generate ASG. This coupling measure is given as,*

$$ACC\,(n) = \frac{|\psi(n)|}{|N_a| - 1},$$

where $\psi(n) = Dep(n)$. $Dep(n)$ is defined in Definition 5.1.

Definition 6.2. *The updated coupling of a method node M in ASG $G_a = (N_a, E_a)$ is defined as the average of the coupling values of all its elements (parameters and statements) along with its own coupling measure. Let a method node M have j number of elements i.e. $n_1, n_2, \ldots, n_j$. Thus, coupling of the method node M is given as*

$$ACC\,(M) = \frac{ACC\,(M) + \sum_{i=1}^{j} ACC\,(n_i)}{j+1}$$

Definition 6.3. *The updated coupling of a class node C in ASG $G_a = (N_a, E_a)$ is defined as the average of the coupling values of all its elements(attributes and methods) along with its own coupling measure. Let a class node C have k number of elements i.e. $n_1, n_2, \ldots, n_k$. Thus, coupling of the class node C is given as*

$$ACC\,(C) = \frac{ACC\,(C) + \sum_{i=1}^{k} ACC\,(n_i)}{k+1}$$

Definition 6.4. *The updated coupling of a package node P in ASG $G_a = (N_a, E_a)$ is defined as the average of the coupling values of all its elements(classes and sub-packages) along with its own coupling measure. Let a package node P have l number of elements i.e. $n_1, n_2, \ldots, n_l$. Thus, coupling of the package node P is given as*

$$ACC\,(P) = \frac{ACC\,(P) + \sum_{i=1}^{l} ACC\,(n_i)}{l+1}$$

Definition 6.5. *The coupling of the sliced component ξ^c, where c is the point of modification taken as the slicing criterion, is thus defined as the average coupling value of all the nodes (packages, classes, methods and statements) in ASG $G_a = (N_a, E_a)$ and is given as*

$$ACC\,(\xi^c) = \frac{\sum_{i=1}^{|N_a|} ACC\,(n_i)}{|N_a|}, \text{ where } n_i \in N_a.$$

Definition 6.6. *The coupling between multiple changes made to the program is given as the ratio of the size of common program elements present in the respective slices ξ_i^c, $1 \leq i \leq m$, where m is the number of changes made to the program, to the* *sum of the size of each slice. The slice ξ_i^c represents the slice obtained with respect to the i^{th} change made to the program. Thus, the coupling between the changes made to a program P is given as,*

$$ACC(P^{\xi^c}) = \frac{|\bigcap_{i=1}^{m} \xi_i^c|}{\sum_{i=1}^{m} |\xi_i^c|}$$

6.2.2 Theoretical Validation

This section shows the theoretical soundness of the proposed measure. We show that this approach satisfies the four basic properties for coupling measure as suggested by Briand et al. [31]. Even though these four basic properties are not sufficient to characterize the proposed coupling measure in a rigorous manner but are necessary to prove the correctness of this approach. There exists many more validation frameworks [189, 204] in the literature, but we validate our approach with Briand's framework and have left other frameworks for future study. The properties for validating this approach along with their proof are given below:
Let $G_a = (N_a, E_a)$ be the ASG, where N_a is the set of affected nodes and E_a be the set of relations (edges) connecting the affected nodes. $ACC\,(n)$ gives the coupling value of any node n in G_a.

Property 1: *Non-Negativity and Null*
This property states that the coupling value of an entity should be null, i.e. $ACC\,(n) = 0$, when there exist no relationships for the entity, otherwise, the coupling value is non-negative and normalized, i.e. $0 \leq ACC\,(n) \leq 1$, for any node n in G_a.
***Proof*.** For any node n in G_a, if there exists no relationships (represented as edges), then $\psi(n) = 0 \implies ACC(n) = \frac{|\psi(n)|}{|N_a|-1} = 0$.
Further, for any node $n \in G_a$, $inflow(n), outflow(n) \subseteq N_a \implies \psi(n) \subseteq N_a$. If $\psi(n) = N_a - \{n\}$, then $|\psi(n)| = |N_a| - 1 \implies ACC(n) = 1$. Hence, it implies $0 \leq \frac{|\psi(n)|}{|N_a|-1} \leq 1$. Therefore, $0 \leq ACC(n) \leq 1$ will always hold true.

Property 2: *Monotonicity*
This property states that adding a new relationship (edge) to the ASG must not decrease the coupling value of any node, i.e. if $G_{a2} = (N_{a2}, E_{a2})$ is obtained by adding an edge $< n1, n2 >$ to $G_{a1} = (N_{a1}, E_{a1})$, then $ACC_2\,(n) \geq ACC_1\,(n)$, where $ACC_1\,(n)$ and $ACC_2\,(n)$ are the coupling values of node n in G_{a1} and G_{a2}, respectively.
***Proof*.** We use $\psi_1\,(n)$, and $\psi_2\,(n)$ to represent the dependence set of a node n in G_{a1} and G_{a2}, respectively.
Case 1: If $n_1, n_2 \in \psi_1\,(n)$, then adding a new edge $< n_1, n_2 >$ will not change

$\psi_1(n)$. Hence, $\psi_1(n) = \psi_2(n) \implies ACCo_1(n) = ACCo_2(n)$.

Case 2: If $n_1 \notin \psi_1(n) \wedge n_2 \in \psi_1(n)$, then

Case i: if $n_2 \in Inflow_1(n)$, then adding $< n_1, n_2 >$ will result in $Inflow_2(n) = Inflow_1(n) \cup \{n_1\}$. This implies $|Inflow_2(n)| = |Inflow_1(n)| + 1$. Since, $|Outflow_2(n)| = |Outflow_1(n)|$ in this case, so $|Inflow_2(n)| + |Outflow_2(n)| = |Inflow_1(n)| + |Outflow_1(n)| + 1 \implies |\psi_2| = |\psi_1| + 1$. Therefore, $ACC_2(n) > ACC_1(1)$, since $|N_{a1}| = |N_{a2}|$.

Case ii: if $n_1 \in Outflow_1(n)$, then adding $< n_1, n_2 >$ will result in $Outflow_2(n) = Outflow_1(n) \cup \{n_2\}$. This implies $|Outflow_2(n)| = |Outflow_1(n)| + 1$. Since, $|Inflow_2(n)| = |Inflow_1(n)|$ in this case, so $|Inflow_2(n)| + |Outflow_2(n)| = |Inflow_1(n)| + |Outflow_1(n)| + 1 \implies |\psi_2| = |\psi_1| + 1$. Therefore, $ACC_2(n) > ACC_1(1)$, since $|N_{a1}| = |N_{a2}|$.

Therefore, adding an edge does not decrease the coupling of any node in G_a.

Property 3: *Merging*

When two sets of nodes N_1 and N_2 are merged to form a single set of nodes N s.t. $N = N_1 \cup N_2$ in G_a then this should not increase the coupling value of any node $n \in N$, i. e. $ACC(N_1) + ACC(N_2) \geq ACC(N)$, since the two sets of nodes may have some common relationships.

Proof. Let N_1 and N_2 be two sets of nodes such that $N_1 \cap N_2 \neq \phi$. For any node $n_1 \in N_1$, $n_2 \in N_2$ and $n \in N$, let $\psi(n_1) \cap \psi(n_2) \neq \phi$.

Case 1: One of the possibility is that $Inflow(n_1) \cap Inflow(n_2) \neq \phi \wedge Outflow(n_1) \cap Outflow(n_2) \neq \phi$.

This implies that $|Inflow(n_1) \cup Inflow(n_2)| = |Inflow(n_1)| + |Inflow(n_2)| - |Inflow(n_1) \cap Inflow(n_2)|$ and $|Outflow(n_1) \cup Outflow(n_2)| = |Outflow(n_1)| + |Outflow(n_2)| - |Outflow(n_1) \cap Outflow(n_2)|$.

Therefore, $|\psi(n)| = |Inflow(n_1) \cup Inflow(n_2)| + |Outflow(n_1) \cup Outflow(n_2)| \implies |\psi(n)| = |Inflow(n_1)| + |Inflow(n_2)| - |Inflow(n_1) \cap Inflow(n_2)| + |Outflow(n_1)| + |Outflow(n_2)| - |Outflow(n_1)| \cap |Outflow(n_2)|$.

Hence, $\psi(n) = |\psi(n_1)| + |\psi(n_1)| - k, where\ k = -(|Inflow(n_1)| \cap |Inflow(n_2)| + |Outflow(n_1)| \cap |Outflow(n_2)|),\ is\ a\ constant.$

$Thus, ACC(n) = \frac{|\psi(n)|}{|N_a|-1} = \frac{|\psi(n_1)|+|\psi(n_2)|-k}{|N_a|-1} \implies ACC(n) = ACC(n_1) + ACC(n_2) - k', where k' = -\frac{k}{|N_a|-1}$. Hence, it is proved that $ACC(n_1) + ACC(n_2) > ACC(n)$.

Case 2: The other possibility is that $Inflow(n_1) \cap Inflow(n_2) \neq \phi \vee Outflow(n_1) \cap Outflow(n_2) \neq \phi$.

Case i: Supposing $Inflow(n_1) \cap Inflow(n_2) \neq \phi$ and $Outflow(n_1) \cap Outflow(n_2) = \phi$. This implies that $|Inflow(n_1) \cup Inflow(n_2)| = |Inflow(n_1)| + |Inflow(n_2)| -$

$|Inflow(n_1) \cap Inflow(n_2)|$
$and\ |Outflow(n_1) \cup Outflow(n_2)| = |Outflow(n_1)| + |Outflow(n_2)|$.
Therefore, $|\psi(n)| = |Inflow(n_1) \cup Inflow(n_2)| + |Outflow(n_1) \cup Outflow(n_2)|$
$\implies |\psi(n)| = |Inflow(n_1)| + |Inflow(n_2)| - |Inflow(n_1) \cap Inflow(n_2)| + |Outflow(n_1)| + |Outflow(n_2)|$.
Hence, $\psi(n) = |\psi(n_1)| + |\psi(n_1)| - k$, where $k = -(|Inflow(n_1) \cap Inflow(n_2)|$ is a constant.
Thus, $ACC(n) = \frac{|\psi(n)|}{|N_a|-1} = \frac{|\psi(n_1)|+|\psi(n_2)|-k}{|N_a|-1} \implies ACC(n) = ACC(n_1)+ACC(n_2)-k'$,
where $k' = -\frac{k}{|N_a|-1}$. Hence, it is proved that $ACC(n_1) + ACC(n_2) > ACC(n)$.
Similarly, it can be proved for **Case ii**: such that $Inflow(n_1) \cap Inflow(n_2) = \phi\ and\ Outflow(n_1) \cap Outflow(n_2) \neq \phi$. Therefore, it is proved that merging two dependent sets of nodes will not decrease the coupling value.
Property 4: *Disjoint Additivity*
When two disjoint sets of nodes N_1 and N_2 are merged to form a single set of nodes N such that $N = N_1 \cup N_2$ in G_a and $N_1 \cap N_2 = \phi$, then the coupling value of any node $n \in N$ is equal to the coupling in two original set of nodes, i. e. $ACC(N_1) + ACC(N_2) = ACC(N)$.
Proof. Let N_1 and N_2 be two sets of nodes s.t. $N_1 \cap N_2 = \phi$. For any node $n_1 \in N_1$, $n_2 \in N_2$ and $n \in N$, $\psi(n_1) \cap \psi(n_2) = \phi$, as the sets are disjoint. Therefore, $Inflow(n_1) \cap Inflow(n_2) = \phi\ and\ Outflow(n_1) \cap Outflow(n_2) = \phi$.
$\psi(n) = Inflow(n) \cup Outflow(n)$
$\implies |\psi(n)| = |Inflow(n_1)| + |Outflow(n_1)| + |Inflow(n_2)| + |Outflow(n_2)|$
$\implies |\psi(n)| = |\psi(n_1)| + |\psi(n_2)|$
$\implies \frac{|\psi(n)|}{|N_a|-1} = \frac{|\psi(n_1))|}{|N_a|-1} + \frac{|\psi(n_2)|}{|N_a|-1}$.
Hence, $ACC(n) = ACC(n_1) + ACC(n_2)$. Therefore, it is proved that after adding two disjoint sets, the coupling value of a node does not change.

6.2.3 Framework Criteria

This section discusses the six guidelines framed by Briand et al. [30] for any coupling measuring framework to satisfy and shows how these issues are addressed in case of our approach. Though these criteria are not sufficient to validate the coupling measure as compared to the properties, but these are necessary as they strongly influence the goal of the stated measurement. These criteria are as follows:

i. *The type of connection*: The type of connection refers to the mechanism that constitutes the relationship between two program components. In our approach, the program components correspond to the nodes of the ASG and the connection between the nodes corresponds to the different dependence edges that exist between the nodes. The various dependences considered here are discussed in Section 4.2.1. Hence, we have a well formed mechanism of identifying the connections between the program components to measure their coupling values.

ii. *The locus of impact*: It refers to the decision of using import coupling or export coupling. Import coupling is used for the analysis of attributes, methods, classes, or packages as clients. However, export coupling is used to analyze the attributes, methods, classes, or packages as servers. In our approach, we used export coupling as our proposed metric [65]. This is because, it is difficult to keep track of error propagation when faulty information flow from the server (attributes, methods, classes, or packages) to the clients. This requires more testing to be carried out.

iii. *Granularity of the measure*: Granularity refers to the level of detail at which information for coupling is gathered. Granularity depends upon two factors: i) first is the components that are considered to measure the coupling factor, ii) second is how exactly the connections are counted. The affected program components (statements, methods, classes and packages) of an object-oriented program those correspond to the nodes in the ASG, are considered for computing the coupling factor. These affected program components are identified with respect to some modification made to the program under consideration. The inter dependences between these affected program components are represented as edges in the intermediate graph ASG. The second factor of granularity, i.e. counting the connections, is addressed by counting the number of nodes that are connected by the edges to determine the frequency of interactions between these affected program components.

iv. *Stability of the server*: This constitutes stable and unstable classes. Stable classes being library classes are not subjected to changes, so they do not trigger any change in the classes using them. Whereas, the unstable classes are liable to changes and hence can trigger modifications in client classes deep in the hierarchy. We consider both the stable and unstable classes and/or packages that can either have a very high or negligibly small impact on the modifications of the client classes.

v. *Direct or indirect coupling*: If a method m1 invokes another method m2, then there exists a direct coupling between m1 and m2. If m2 further invokes another method m3, then there exists an indirect coupling between m1 and m3. The depth of indirect coupling can be very deep in the hierarchy and can be a bottleneck for testing. We address both direct and indirect coupling in our approach through our backward and forward graph traversal mechanism.

vi. *Inheritance*: Inheritance based coupling can exist under following conditions: when a class directly inherits another class, a class instantiates the object of another class and when polymorphism exists. All the above three situations are represented as dependences in our intermediate graph representation. When a class directly inherits another class or instantiates the object of another class, it is represented by the *inheritance edge* and *instantiation edge*, respectively. The attributes and methods of the server class that are referred in the client class are shown through *inherited membership edge.* In the graph representation, each method call is associated with every possible method definition through *polymorphic call edges.* Hence, our approach considers all forms of coupling due to inheritance.

6.3 Our Proposed Approach for Regression Test Case Prioritization

In this section, we first discuss our proposed approach to prioritize a given test suite based on the test cases selected for regression testing. The activities of our proposed prioritization process are shown in Figure 6.2. These activities are summarized below:

Step i. Construct the intermediate representation affected slice graph (ASG).

Step ii. Compute the affected component coupling (ACC) value of each node of the ASG.

Step iii. Cluster the ACC values and assign weights to the nodes of ASG.

Step iv. Compute the weight of test cases and prioritize the test cases based on their weights.

6.3.1 Construction of ASG

ASG is the graphical representation of the slice that is computed with respect to some change made to the program. The steps of constructing the ASG are discussed in Section 5.2.3.

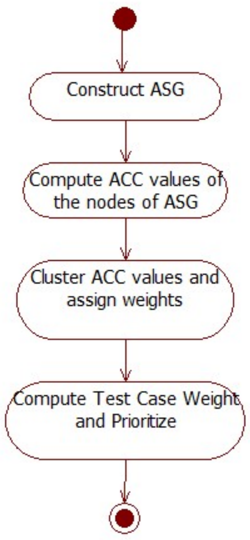

Construct ASG
Compute ACC values of the nodes of ASG
Cluster ACC values and assign weights
Compute Test Case Weight and Prioritize

Figure 6.2: Activities of Test Case Prioritization.

Table 6.3: Test case coverage of fault prone affected nodes.

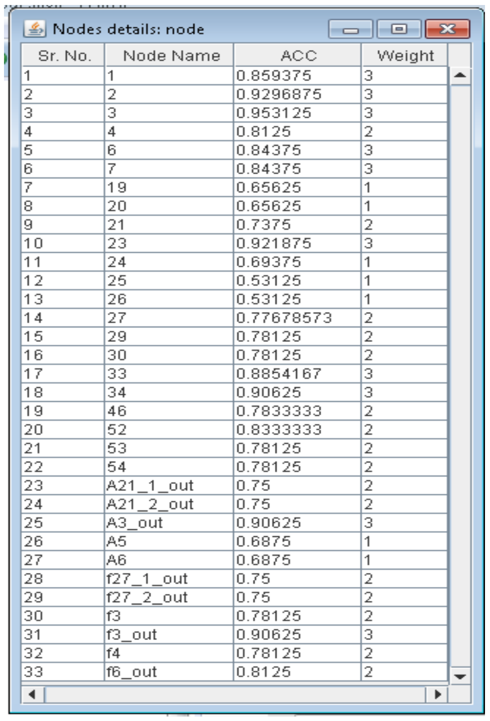
Sl. No.	Test Case	Nodes Covered	# Nodes	Test Case Weight
1	$T6$	**1, 2, 3, 4, 6, 7**	6	17
2	$T7$	**1, 2, 21, 46, 27, f3, f4, 29, 30, f27_1_out, f27_2_out, 33, 34, f3_out,A3_out**, 24, 25, 26, A5, A6	20	41
3	$T8$	**1, 2, 3, 4, 6, 7, 21, 46, 27, f3, f4, 29, 30, f27_1_out, f27_2_out**, 19, 20, A5, A6, 25, 26	21	41
4	$T9$	**1, 2, 3, 4, 6, 7, 21, 46, 52, 27, f4, 33, 30, F3_out, 34**, 24, A6	17	40
5	$T10$	**1, 2, 3, 4, 21, 23, A3_out, 46, 34, 33, F3_out**, 24	12	31

6.3.2 Computation of ACC

We propose an algorithm named *find Weighted Affected Component Coupling (find-WACC)* that is given in Algorithm 7. It takes the ASG and its total number of nodes as input. It uses the formula given in Equation 6.1 to compute the ACC value of each node in the ASG. It invokes Algorithm 5 at Line 4 to compute the outflow of a node and invokes Algorithm 6 at Line 5 to compute the inflow of a node. Algorithm 7 computes the ACC values of each node and then updates these values for some specific nodes such as package nodes, class nodes, method nodes and method call nodes. The calculated ACC values of different nodes of the ASG given in Figure 5.2 are shown in Figure 6.3. Algorithm 7 takes the ASG as input and calculates the ACC of each node as shown in Figure 6.3. We use the concept of information inflow and outflow in this approach of export coupling measurement. The ASG represents all form of information flow between any two nodes of a program in the form of edges. So, we compute the outflow by counting the number of nodes traversed in the forward direction from node n. We use Algorithm 5 to count the outflow of a given node. It traverses forward from the input node and counts the number of nodes that depend on the input node. Algorithm 5 counts the visited node, only if its status is marked *false*. Otherwise, the algorithm skips the node and moves to the next successor node. Similarly, inflow to a node is computed by Algorithm 6. It traverses backward from the input node and counts the number of nodes on which the input node depends. Algorithm 6 first checks the status of the visited node. It counts the visited node, only if its status is marked *false*. Otherwise, it visits the next predecessor node. Thus, our proposed *affected component coupling (ACC)* for a given node n is computed as the normalized ratio of the sum of inflow and outflow of n with total nodes in ASG. This is expressed as,

$$ACC\,(n) = \frac{|inflow\,(n)| + |outflow\,(n)|}{|N_a| - 1} \tag{6.1}$$

The detail computation of ACC for each node is shown in Section 6.4. The reason for taking a change-based coupling into consideration is that any node having higher ACC value is an indicator for that node to be more error-prone [65]. This is because higher ACC value of a node indicates more dependence of other nodes on this source of information. As a result, any error that occurs at the origin of information is likely to be propagated to the dependent nodes. Therefore, higher ACC value of a node implies the need for thorough testing at the origin of information to save time and cost of retesting. Our approach of calculating the ACC values, includes the effect of nested function calls, multi-level inheritance, method overriding and



Nodes details: node

Sr. No.	Node Name	ACC	Weight
1	1	0.859375	3
2	2	0.9296875	3
3	3	0.953125	3
4	4	0.8125	2
5	6	0.84375	3
6	7	0.84375	3
7	19	0.65625	1
8	20	0.65625	1
9	21	0.7375	2
10	23	0.921875	3
11	24	0.69375	1
12	25	0.53125	1
13	26	0.53125	1
14	27	0.77678573	2
15	29	0.78125	2
16	30	0.78125	2
17	33	0.8854167	3
18	34	0.90625	3
19	46	0.7833333	2
20	52	0.8333333	2
21	53	0.78125	2
22	54	0.78125	2
23	A21_1_out	0.75	2
24	A21_2_out	0.75	2
25	A3_out	0.90625	3
26	A5	0.6875	1
27	A6	0.6875	1
28	f27_1_out	0.75	2
29	f27_2_out	0.75	2
30	f3	0.78125	2
31	f3_out	0.90625	3
32	f4	0.78125	2
33	f6_out	0.8125	2

Figure 6.3: The calculated ACC values of different nodes of the ASG in Figure 5.2 and their weights.

overloading, message passing, etc. on a node n. Our technique is more precise as we compute the coupling value of only those nodes that are affected or get affected by the modification done to the program under test (represented as ASG), instead of the program as a whole.

Table 6.4: Impact types of ACC values.

Sl. No.	ACC range	Impact	Weight
1	0.7 – 1.0	*Strong(Critical)Association*	3
2	0.6 – 0.7	*Moderate Association*	2
3	0.0 – 0.6	*Weak or no Association*	1

6.3.3 Clustering and Assigning Weights

Once the ACC values of all the nodes, have been computed, then the values are clustered [150, 197] based on the category of the ACC value. K-means clustering technique [197] is used to cluster the coupling values of each node in the ASG. K-means clustering is a technique of automatically partitioning a set of given data into k groups. It first randomly selects k initial cluster centres and then iteratively does the followings:

i. It assigns each data set to the cluster centre from which it is at minimum distance.

ii. Then, each cluster center is updated to the mean value of its constituent instances.

iii. The above steps are repeated until there is no further change in the cluster centre.

The k cluster centres are chosen randomly from the data set. The value of k for our approach is 3 for dividing the coupling values into three categories as shown in Table 6.4. These three categories of fault association are: *critical fault association*, *moderate fault association* and *weak fault association*. The computed ACC value of any node of ASG can belong to either of these three categories.
Algorithm 7 assigns weight to the nodes of ASG based on the impact type of the ACC value of each node. If the ACC value of a node x belongs to the category of *critical fault association* i.e. $0.7 \leq ACC(x) \leq 1.0$, then x is assigned with a weight 3. Similarly, if ACC value of a node x belongs to the category of *moderate fault association* i.e. $0.4 \leq ACC(x) < 0.7$, then x is assigned with a weight 2. Otherwise, x belongs to the category of *weak fault association* and is assigned a weight 1. The ACC value of each node in ASG and the corresponding weights assigned to them are shown in Figure 6.3.

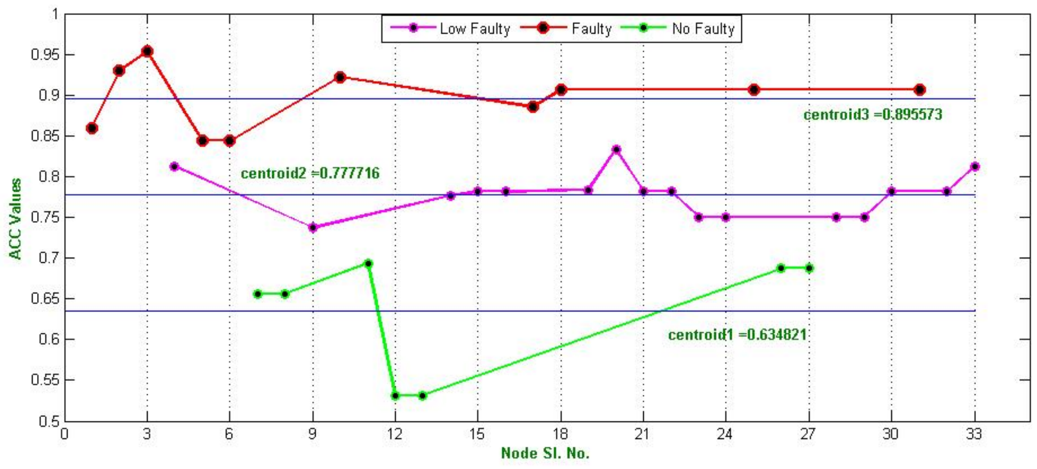


Figure 6.4: K-Means Clustering of the ACC values of the nodes of ASG.

Algorithm 5 Forward Traversal

```
1: procedure FTRAVERSE($ASG, x$)
2:     $\forall u \in Succ[x]$    ▷ Where $x$ is the node under consideration for ACC calculation and $Succ[x]$ is an array of succcessor nodes of $x$.
3:         $if\,(u = FALSE)$
4:             $count = count + 1$
5:             $Set\ u = TRUE$
6:             $Call\ FTraverse(ASG, u)$
7:         $else\ break$
8:     return $count$
9: end procedure
```

Algorithm 6 Backward Traversal

```
1: procedure BTRAVERSE($ASG, x$)
2:     $\forall u \in Pred[x]$    ▷ Where $x$ is the node under consideration for ACC calculation and $Pred[x]$ is an array of predecessor nodes of $x$.
3:         $if\,(u = FALSE)$
4:             $count = count + 1$
5:             $Set\ u = TRUE$
6:             $Call\ BTraverse(ASG, u)$
7:         $else\ break$
8:     return $count$
9: end procedure
```

6.3.4 Computation of Test Case Weights and Prioritization of Test Cases

The program under consideration is executed against each selected test case in a given test suite to find the coverage information as shown in Table 6.3. The weight of a test case depends upon the weight of the nodes that it covers. All the critical and moderate nodes (as per the classification given in Table 6.4) with high weights are shown in bold in the *Nodes Covered* column. The fourth column on *Test Case Weight*, gives the total weight assigned to each test case. To compute the weights and prioritize the given test suite, we propose an algorithm named *Hierarchical Prioritization of Test Cases using Affected Component Coupling (H-PTCACC)*. This is given in Algorithm 8. This algorithm takes the test suite (containing selected test cases for regression testing) along with their coverage information and ACC values of each node in the ASG as its input. The output of the algorithm is a prioritized set of test cases. For any test case $t_i \in T$, Algorithm 8 first computes the sum of the weights of all the critical fault prone nodes covered by t_i (critical weight, Wtc). In Table 6.5, the third column gives the critical weights of all test cases T6 to T10. Similarly, Algorithm 8 computes the sum of the weights of all the

moderate fault prone nodes covered by t_i (moderate weight, Wtm). This is shown in the fourth column of Table 6.5 for each of the corresponding test cases. In the same way, Algorithm 8 computes the sum of the weights of all weak fault prone nodes (weak weight, Wtw) for each test case that are shown in the fifth column of Table 6.5. The algorithm then computes the total weight of each test case by adding its critical weight (Wtc), moderate weight (Wtm) and weak weight (Wtw). The sixth column of Table 6.5 shows the total weight computed for each of the test cases. The assigned weight of a test case is the summation of all the weights of the nodes covered by that test case. Similarly, the weights of $T7, T8, T9$ *and* $T10$ are calculated.
Algorithm 8 assigns priority to the test cases based on their different computed weights. The test case having a higher total weight is given higher priority in the test suite. If any of the two test cases have same total weight then their priority is decided based on their critical weight. The test case with higher critical weight is given higher priority. Similarly, if the critical weights are also same then the moderate weights are taken into consideration for prioritization. If the moderate weight of the test cases are again same then the weak weights are considered for prioritization. If still the weak weights are same for any two test cases, then both the test cases are given equal priority. The seventh column in Table 6.5 shows the final prioritized sequence of the selected test cases.

Table 6.5: Distribution of test case weights on the basis of fault prone impact.

Sl. No.	Test Case	Weight of Critical Fault Prone Nodes Covered	Weight of Moderate Fault Prone Nodes Covered	Weight of Weak Fault Prone Nodes Covered	Total weight of Test Case	Priority (according to the total wt. of Test Case)
1	$T6$	15	2	0	17	V
2	$T7$	18	18	5	41	I
3	$T8$	15	20	6	41	II
4	$T9$	24	14	2	40	III
5	$T10$	24	6	1	31	IV

6.4 Case Study

Consider the example Java program shown in Figure 4.2. This program though a very small program in size but it represents all the important features of a Java program and ideally suits to explain the working of this approach. We perform our decomposition slicing on EOOSDG as described in Section 4.2.4. The slice is

Algorithm 7 *findWACC(ASG, n)*

Input: Affected Slice Graph (ASG), total number of nodes n

Output: Weighted Affected Component Coupling (WACC) of each node

```
1: $for\ x := V_1, V_2, V_3, \dots, V_n$    ▷ Where x is any node in ASG.
2:     $Setstatus_x = FALSE$
3:   $outflow := call\ FTraverse(ASG, x)$
4:   $inflow := call\ BTraverse(ASG, x)$
5:   $ACC[x] := \frac{(inodes+enodes)}{n-1}$
6: $End\ for$    ▷ To update the coupling value of all the method, class and package nodes.
7: $for\ u := M_1, M_2, M_3, \dots, M_m$    ▷ Where m is the number of method nodes in the graph.
8:   $ACC[u] := \frac{(ACC[u]+\sum_{i=1}^{j} ACC[n_i])}{(j+1)}$    ▷ $n_i$ is the statement/parameter node of method $M_i$, j is the total number of statement/parameter nodes of each $M_i$.
9: $End\ for$
10: $for\ u := C_1, C_2, C_3, \dots, C_c$    ▷ c is the total number of class nodes.
11:   $ACC[u] := \frac{(ACC[u]+\sum_{i=1}^{k} ACC[n_i])}{(k+1)}$    ▷ $n_i$ is the attribute/method node of class $C_i$, k is the total number of attribute/method nodes of each $C_i$.
12: $End\ for$
13: $for\ u := P_1, P_2, P_3, \dots, P_p$    ▷ p is the total number of package nodes.
14:   $ACC[u] := \frac{(ACC[u]+\sum_{i=1}^{l} ACC[n_i])}{(l+1)}$    ▷ $n_i$ is the subpackage/class node of package $P_i$, l is the total number of subpackage/class nodes of each $P_i$.
15: $End\ for$
16: $ACC(S) = \frac{\sum_{i=1}^{|N_a|} ACC(n_i)}{|N_a|}$    ▷ ACC(S) represents the cohesion of slice S    ▷ To assign a weight to each node of ASG.
17: $for\ u := V_1, V_2, V_3, \dots, V_n$    ▷ Where u is any node in ASG.
18:   $if\ ACC[u] >= 0.7\ and\ ACC[u] <= 1.0$
19:     $WACC[u] := 3$
20:   $End\ if$
21:   $else\ if\ ACC[u] >= 0.6\ and\ ACC[u] < 0.7$
22:     $WACC[u] := 2$
23:   $End\ else\ if$
24:   $else$
25:     $WACC[u] := 1$
26:   $End\ else$
27: Exit
```

computed by assuming that the object at statement 23 of the example program in Figure 4.2 is changed. Statement 23 is taken as the slicing criterion. The ASG shown in Figure 5.2 represents all those program parts as nodes that affect or get affected by the change made at statement 23. Figure 6.3 shows the corresponding coupling values along with their weights. Now, we discuss the working of our proposed algorithms. We have taken an example Java program shown in Figure 4.2 as a case study to discuss the working of the proposed algorithms. We construct the ASG as given in Figure 5.2 by performing HD slicing on the intermediate graph representation (EOOSDG in Figure 4.3) of this example Java program. The details

Algorithm 8 *H-PTCACC(T, WACC)*

Input: Test Suite T with coverage information, Weight of each node in ASG
Output: Prioritized Test Suite T'

```
1: Set $T' = NULL$
2: for each test case $t \in T$ do
3:     $Wtc(t) = \sum_{i=1}^{j} Wt\left(V_{ci}(t)\right)$    $\triangleright$ Where $V_{ci}(t)$ is the node covered by t whose weight is 3, $Wtc(t)$ is the total weight of j critical fault prone nodes covered by t.
4:     $Wtm(t) = \sum_{i=1}^{k} Wt\left(V_{mi}(t)\right)$    $\triangleright$ Where $V_{mi}(t)$ is the node covered by t whose weight is 2, $Wtm(t)$ is the total weight of k moderate fault prone nodes covered by t.
5:     $Wtw(t) = \sum_{i=1}^{l} Wt\left(V_{wi}(t)\right)$    $\triangleright$ Where $V_{wi}(t)$ is the node covered by t whose weight is 1, $Wtw(t)$ is the total weight of l weak fault prone nodes covered by t.
6:     $Wt(t) = Wtc(t) + Wtm(t) + Wtw(t)$    $\triangleright$ Where $Wt(t)$ is the total weight of t.
7: End for    $\triangleright$ sort on the basis of $Wt(t)$.
8: $T' = sort\left(T_{Wt(t)}\right)$
9:     $if\ \exists\ t_i, t_j \in T'\ s.t\ Wt(t_i) = Wt(t_j), i \neq j$
                                        $\triangleright$ sort on the basis of $Wtc(t_i),\ Wtc(t_j)$.
10:        $T' = sort\left(T_{Wtc(t_i,t_j)}\right)$
11:        $if\ Wtc(t_i) = Wtc(t_j), i \neq j$    $\triangleright$ sort on the basis of $Wtm(t_i),\ Wtm(t_j)$.
12:            $T' = sort\left(T'_{Wtm(t_i,t_j)}\right)$
13:            $if\ Wtm(t_i) = Wtm(t_j), i \neq j$    $\triangleright$ sort on the basis of $Wtw(t_i),\ Wtw(t_j)$.
14:                $T' = sort\left(T'_{Wtw(t_i,t_j)}\right)$
15: Exit
```

of the ASG generation is discussed earlier in Section 5.2.3. Algorithm 5 and Algorithm 6 generate forward and backward dependence sets and calculate the outflow and inflow, respectively, for each node of the ASG.

Algorithm 7 uses the formula given in Equation 6.1 to compute the ACC value of each node in the ASG. For example, we show the ACC calculation for the class *Triangle* represented as *node 24* in Figure 5.2. Initially, ACC value of *node*24 is given by

$$ACC(24) = \frac{|outflow\,(24)| + |inflow\,(24)|}{|N_a| - 1} = \frac{20+4}{32} = 0.75$$

Similarly, the ACC values of all the associated nodes (25, 26, 27, f3, f4, 29, 30, f27_1_out, f27_2_out, 33, f3_out, 34) of node 24 as shown in Figure 5.2 are computed as follows:

$$ACC(25) = \frac{|outflow(25)|+|inflow(25)|}{|N_a|-1} = \frac{3+14}{32} = 0.5312$$
$$ACC(26) = \frac{|outflow(26)|+|inflow(26)|}{|N_a|-1} = \frac{3+14}{32} = 0.5312$$
$$ACC(27) = \frac{|outflow(27)|+|inflow(27)|}{|N_a|-1} = \frac{12+7}{32} = 0.5937$$
$$ACC(f3) = \frac{|outflow(f3)|+|inflow(f3)|}{|N_a|-1} = \frac{13+12}{32} = 0.7812$$
$$ACC(f4) = \frac{|outflow(f4)|+|inflow(f4)|}{|N_a|-1} = \frac{13+12}{32} = 0.7812$$
$$ACC(29) = \frac{|outflow(29)|+|inflow(29)|}{|N_a|-1} = \frac{12+13}{32} = 0.7812$$

$ACC(30) = \frac{|outflow(30)|+|inflow(30)|}{|N_a|-1} = \frac{12+13}{32} = 0.7812$
$ACC(f27\_1\_out) = \frac{|outflow(f27\_1\_out)|+|inflow(f27\_1\_out)|}{|N_a|-1} = \frac{10+14}{32} = 0.75$
$ACC(f27\_2\_out) = \frac{|outflow(f27\_2\_out)|+|inflow(f27\_2\_out)|}{|N_a|-1} = \frac{10+14}{32} = 0.75$
$ACC(33) = \frac{|outflow(33)|+|inflow(33)|}{|N_a|-1} = \frac{3+24}{32} = 0.8437$
$ACC(f3\_out) = \frac{|outflow(f3\_out)|+|inflow(f3\_out)|}{|N_a|-1} = \frac{1+28}{32} = 0.9062$
$ACC(34) = \frac{|outflow(34)|+|inflow(34)|}{|N_a|-1} = \frac{2+27}{32} = 0.9062$
Then, Algorithm 7 updates the ACC value of each node of ASG. The reason behind this updation is that, for any node that represents a method, the statements contained inside that method also contribute to the ACC of the method. Even if a method does not have any statement inside it, still it will have some ACC value as some other method may be overriding it. Therefore, we have taken the average of all the ACC values of all the statements and the ACC value of the method under consideration, to compute the updated ACC value of the method. For example, the ACC values of nodes $\{24, 27, 33\}$ are updated. The average ACC value of $node27$ along with the ACC values of all its member nodes $\{f3, f4, 29, 30, f27\_1\_out, f27\_2\_out\}$ are computed and assigned to $node27$, i.e.
$ACC(27) = \frac{ACC(27)+ACC(f3)+ACC(f4)+ACC(29)+ACC(30)+ACC(f27\_1\_out)+ACC(f27\_2\_out)}{7}$
$= \frac{0.5937+0.7812+0.7812+0.7812+0.7812+0.75+0.75}{7} = 0.7455$
Similarly, ACC values of $Node33$ and $Node24$ are updated as follows:
$ACC(33) = \frac{ACC(33)+ACC(f3\_out)+ACC(34)}{3} = \frac{0.84375+0.90625+0.90625}{3} = 0.88542$
$ACC(24) = \frac{ACC(24)+ACC(25)+ACC(26)+ACC(27)+ACC(33)}{5} = \frac{0.75+0.5312+0.5312+0.7455+0.88542}{5}$
$= 0.688664$
Therefore, ACC value of *class Triangle* in Figure 4.2 represented as $node24$ in Figure 5.2 is found to be 0.68866. Similar procedure is followed to update the ACC values of all the nodes representing the classes and packages in the ASG.

Algorithm 8 computes the critical fault prone weight $Wtc(t_i)$, moderate fault prone weight $Wtm(t_i)$, weak fault prone weight $Wtw(t_i)$ and the total weight $Wt(t_i)$ for each test case $t_i \in T$. For example, the nodes covered by test case $T8$ as given in the second column of Table 6.3 are {1, 2, 3, 4, 6, 7, 21, 46, 27, f3, f4, 29, 30, f27_1_out, f27_2_out, 19, 20, A5, A6, 25, 26 }. The critical fault prone nodes covered by $T8$ are $\{1, 2, 3, 6, 7\}$. So, critical fault prone weight of $T8$ is calculated as $Wtc(T8) = Wt(1)+Wt(2)+Wt(3)+Wt(6)+Wt(7) = 3+3+3+3+3 = 15$. The moderate fault prone nodes covered by $T8$ are $\{4, 21, 46, 27, f3, f4, 29, 30, f27\_1\_out, f27\_2\_out\}$. So, moderate fault prone weight of $T8$ is calculated as $Wtm(T8) = Wt(4) + Wt(21)+Wt(46)+Wt(27)+Wt(f3)+Wt(f4)+Wt(29)+Wt(30)+Wt(f27\_1\_out)+ Wt(f27\_1\_out) = 2+2+2+2+2+2+2+2+2+2 = 20$. Similarly, the weak fault prone nodes covered by $T8$ are $\{25, 26, 29, 30\}$ and the weak fault prone weight of $T8$ is calculated as $Wtw(T8) = Wt(19)+Wt(20)+Wt(A5)+Wt(A6)+Wt(25)+Wt(26) = 1 + 1 + 1 + 1 + 1 + 1 = 6$. Therefore, total weight of the test case $T8$ is calculated as $Wt(T8) = Wtc(T8) + Wtm(T8) + Wtw(T8) = 15 + 20 + 6 = 41$.

Then, the algorithm sorts the test cases in the decreasing order of their total weights $Wt(t_i)$. If there exist some test cases t_i, t_j such that $Wt(t_i) = Wt(t_j)$, then the algorithm sorts t_i and t_j based on their critical fault prone weights $Wtc(t_i)$ and $Wtc(t_j)$. If for some test cases $Wtc(t_i) = Wtc(t_j)$, then t_i and t_j are sorted based on their moderate fault prone weights $Wtm(t_i)$ and $Wtm(t_j)$. If again, $Wtm(t_i) = Wtm(t_j)$, then test cases are sorted by their weak fault prone weights $Wtw(t_i)$ and $Wtw(t_j)$. In a very unlikely case, if the weak fault prone weights are still identical, i.e. $Wtw(t_i) = Wtw(t_j)$, then the test cases are given equal priority. The prioritized order of the test cases $T6 - T10$ based on their respective weights is obtained as $\{T7, T8, T9, T10, T6\}$.

6.5 Correctness of the Algorithms

In this section, we prove the correctness of the proposed algorithms.

Theorem 6.1. *findWACC algorithm correctly computes the ACC values and terminates after finite number of steps.*

Proof. Let the graph ASG be defined as $G_a = (N_a, E_a)$, where N_a is the set of nodes in the graph and E_a is the set of edges connecting the nodes. $|N_a|$ cardinality of the set N_a, is a finite number. Algorithm 6 invoked at Line 4 counts the number of nodes reached during backward traversal. To avoid recounting of the same node, the algorithm updates the count iff the reached node is marked false. Otherwise, it skips the node if already marked true. Since, the number of nodes to be traversed (all the predecessor nodes) is finite, the algorithm terminates after all the predecessor nodes are traversed. Similarly, Algorithm 5 invoked at Line 5 counts the number of nodes reached during forward traversal and terminates after finite number of nodes are traversed. Therefore, the for loop at Line 1 of the algorithm computes the ACC value of each node and terminates after finite number of iterations. Next, Lines 8 - 16 in the algorithm updates the ACC values of all the method nodes, class nodes and package nodes. Let m, c, p be the total number of method nodes, class nodes and package nodes respectively, in the graph ASG. Therefore $m < |N_a|$, $c < |N_a|$ and $p < |N_a|$ imply that the for loop in Line 8, Line 11 and Line 14 compute correctly the respective updated ACC values and terminate after finite number of iterations. Lines 17 - 26 assigns a weight to each node in the ASG based on their ACC values. Since, the number of nodes to be accessed is finite (as discussed earlier in this section), the for loop at Line 17 terminates after finite number of iterations. The algorithm finally exits at Line 27. This establishes the correctness of the algorithm. □

Theorem 6.2. *H-PTCACC algorithm correctly prioritizes the test cases and terminates after finite number of steps.*

Proof. Let the graph ASG be defined as $G_a = (N_a, E_a)$, where N_a is the set of nodes in the graph and E_a is the set of edges connecting the nodes. Let $n = |N_a|$ be the cardinality of the set N_a, where n is a finite number. Let there be m number of test cases in the test suite T, where m is a finite number. Therefore, the for loop at Line 2 controls the number of iterations and terminates after m iterations. Lines 3 - 7 correctly compute the different critical weights for each of the m test cases depending on their node coverage. Lines 8 - 14 arrange the test cases in the decreasing order of their associated weights computed earlier. The *if* conditions at Line 9, Line 11 and Line 13 ensure the execution of the appropriate ordering criteria. Since, finite number of test cases are there and finite number of conditions are used to sort them, so the prioritization process terminates after finite number of steps. The algorithm finally exits at Line 15. This establishes the correctness of the algorithm. □

6.6 Complexity Analysis of the Algorithms

The complexity analysis of the proposed algorithms is given as follows:
Space Complexity: Let the computed slice represented as ASG has n nodes. Each node in the ASG corresponds to each statement of the computed slice along with the actual and formal arguments present. Hence, the space requirement is given as $O(n)$. Each node may have dependences on other nodes. These dependences on other nodes are represented as edges. Since, each node can be dependent on maximum $(n-1)$ other nodes, the space requirement for the edges is $O(n^2)$. Hence, the total space requirement is $O(n^2+n) \equiv O(n^2)$.
Time Complexity: Let n be the set of nodes in the ASG. To compute the inflow to the input node, each node is traversed only once, so the time complexity is $O(n)$. If the time spent in each recursive call is ignored, then each node u can be processed

in $O(1+pred[u])$, where pred[u] represents the set of predecessor nodes of u. If each node has every other node in the graph as its predecessor node, then each node has $(n-1)$ predecessor nodes. So, the time complexity to process each node is $O(1+(n-1)) \approx O(n)$. Similarly, to compute the outflow from the input node the time complexity is calculated as $O(n)$. Then, the total time required to compute the coupling values of all the nodes is calculated as $O(N^2)$.
Let m, c and p be the number of method nodes, class nodes and package nodes respectively, whose ACC values need to be updated. If each method node has j member nodes, then the time required to update m method nodes is $O(mjn^2)$. Since m and j are small bounded positive integers, the time complexity is calculated as $O(n^2)$. Similarly, If each class node has k member nodes, and each package node has l member nodes, then the respective time complexities for c class nodes and p package nodes are $O(ckn^2)$ and $O(pln^2)$. Since c, k, p and l are small bounded positive integers, the time complexities are calculated as $O(n^2)$ and $O(n^2)$ respectively, for the class and package nodes. As n nodes are there with n ACC values, so the time required to assign a weight to each of the n nodes depending on their respective ACC value is $O(n)$. Therefore, the worst-case run-time of the findWACC algorithm is calculated as $O(n^2+n^2+n^2+n^2+n) \equiv O(n^2)$.
Let t be the number of test cases to be prioritized in the given test suite T. Suppose a test case covers at most n number of nodes. Let j, k and l be the critical, moderate and weak fault prone nodes, respectively, covered by a test case, such that $n = j+k+l$. So, the time complexity to compute the weight of each test case is calculated as $O(j+k+l) \equiv O(n)$. As a result, the total time complexity to compute the weight of t test cases in the given test suite T is $O(tn)$. Assuming $t \equiv n$, the time complexity to compute the weights is calculated as $O(n^2)$. The time complexity to sort the $t \equiv n$ test cases is calculated as $O(n^2)$. Therefore, the worst-case run-time of the H-PTCACC algorithm is calculated as $O(n^2+n^2) \equiv O(n^2)$.

6.7 Implementation

In this section, we briefly describe the implementation of our work. We computed the coupling factors of all the nodes and used K-means technique to cluster these computed coupling values. K-means algorithm is implemented in Matlab [2]. The test cases for the input program are generated using Junit [3] Eclipse plugin. To

[2]http://www.mathworks.in/products/matlab/
[3]http://junit.org/

generate the mutants for the input program, we used an Eclipse plugin known as MuClipse. MuClipse [183] is the Eclipse plugin version of μJava that generates two types of mutation operators both for traditional mutation and class mutation. We have considered both types of mutation in our approach.

For implementation, we have taken the sample Java program shown in Figure 4.2. A total of twenty test cases (T1-T20) were taken along with their node coverage information. All those test cases that covered the affected nodes (with respect to a modification point) are selected hierarchically. First, the test cases (T1-T20) covering the affected package nodes are selected. After that, out of these selected test cases, all those test cases (T1-T10) that executed the affected classes, are selected. The similar process is then followed to select the test cases executing the affected methods and statements. Finally, five test cases (T6-T10) are selected for regression testing of our program under consideration. In this approach, we propose a technique to prioritize these five selected test cases (T6-T10). Table 6.3 shows the nodes covered by each of the test cases and the weight assigned against it based on the export coupling factor. Distribution of ACC values of all these nodes into three different fault prone categories by K-means technique [197] is shown in Figure 6.4. These coupling values are clustered, and a weight is assigned to each impact category of fault proneness as shown in Table 6.4. Based on these assigned weights, Algorithm 8 computes the following four different weights for each test case: *critical node coverage weight*, *moderate node coverage weight*, *weak node coverage weight* and *total weight*. The algorithm then sorts the test cases in the decreasing order of these computed weights and produces the prioritized sequence of test cases. From Table 6.5, it may be observed that the prioritized sequence of test cases is $\{T7, T8, T9, T10, T6\}$.

6.7.1 Experimental Program Structure

To show the effectiveness of our approach, we have taken total fifteen programs of different specifications as shown in Table 6.6. The last column of Table 6.6 shows the time taken for prioritizing the selected test cases. The prioritization time includes the time for computing the weights of the test cases and the time taken to order the test cases in decreasing weight value. Out of these fifteen programs, ten benchmark programs (Stack, Sorting, BST, CrC, DLL, Elevator_spl, Email_spl, GPL_spl, Jtopas, Nanoxml) are taken from Software-artifact Infrastructure Repository (SIR) [59] and other five programs are developed as academic assignments. These smaller programs are chosen to ascertain the correctness and accuracy of the

Table 6.6: Result obtained for regression testing of different programs.

Sl. No.	Programs	Lines of Code	Total # Test Cases	# Mutants	# Selected Test Cases for regression testing	Time for prioritization (sec)
1	Expt. Program	54	20	14	5	1.3
2	Calculator	75	15	42	7	1.8
3	Elevator	90	25	27	10	2.59
4	Stack	114	22	35	9	2.27
5	Sorting	130	16	43	5	1.65
6	BST	130	20	51	12	3.21
7	CrC	261	18	46	6	1.64
8	DLL	277	24	47	6	1.78
9	Notepad	300	17	17	8	2.07
10	ATM	900	33	39	12	3.87
11	Elevator_spl	1046	15	53	10	2.63
12	Email_spl	1233	18	18	11	2.89
13	GPL_spl	1713	22	22	14	3.7
14	Jtopas	5400	16	28	9	2.36
15	Nanoxml	7646	14	32	7	1.72

approach, keeping in mind that they represent a variety of Java features and applications, the test cases are available and otherwise easily developed, and coverage information can be computed.

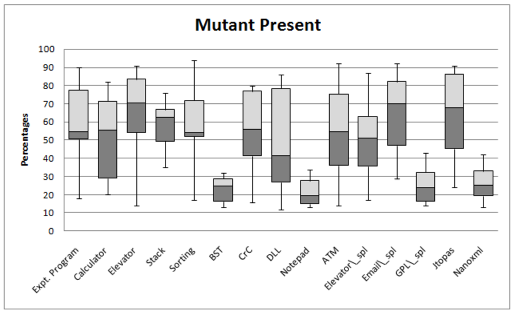
6.7.2 Mutation Analysis

To generate the mutants for the input program, we used an Eclipse plugin of MuJava known as MuClipse [44]. Fault mutants are considered to be good representative of real faults. [11, 102, 196]. MuClipse supports both the traditional and object-oriented operators for mutation analysis. Table 6.7 gives an overview of the mutation operators considered in the experimental study. A brief description of the operators is given for every operator in Table 6.7. The first five operators are the traditional operators. The remaining 23 operators relate to the faults in object-oriented programs. Out of which *JTD*, *JSC*, *JID*, *JDC* are specific to Java features that are not available in all object-oriented languages. Apart from this, there are some other operators, such as *EOA*, *EOC*, *EAM*, *EMM*, that reflect the typical coding mistakes common during development of an object-oriented software. The mutant generator generates the mutants for the sliced program (representing the affected program parts) according to the operators selected by the testers. Very

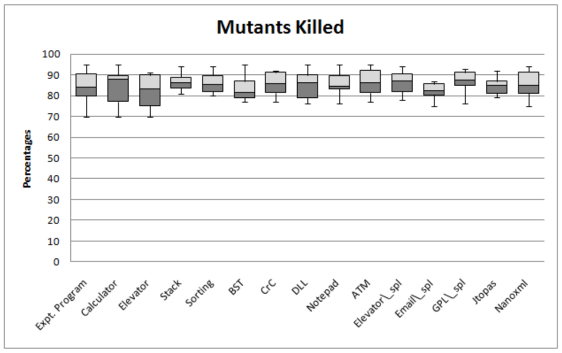
Table 6.7: Overview of Mutation Operators

Traditional Operators	
Operator	*Description*
ABS	Absolute value insertion
AOR	Arithmetic operator replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
UOI	Unary operator insertion
Java Inter-Class Operators	
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	super keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	new method call with child class type
PMD	Instance variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAO	Argument order change
OAN	Argument number change
JTD	this keyword deletion
JSC	static modifier change
JID	Member variable initialization deletion
JDC	Java-supported default constructor create
EOA	Reference and content assignment replacement
EOC	Reference and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change

large number of mutants are generated. The location of these mutants in the source code is visualized through mutant viewer. It allows a tester to select appropriate number of mutants and design test cases to kill the mutants. As the number of generated mutants are too large, we randomly selected a less number of mutants for our experimental programs. This process was repeated for 10 times and the rate of fault detection for the prioritized test suite was computed. The average number of mutants selected for every program is shown in Table 6.6. The test cases are written in a specific format such that each test case is in a form of invoking a method in the class under test. The test method has no parameters and return the result in the form of a string. The mutant is said to be killed if the obtained output does not match with the output of the original program. The test cases for the input program are generated using JUnit Eclipse plugin as the JUnit test cases closely match the required format. The total number of fault mutants for all the fifteen programs is 514, and the average number of mutants per program is 34. Figure 6.5 shows the boxplots of the results of our mutation analysis for all the experimental programs. The average percentage of affected nodes covered by the prioritized



(a) Box-plot of the % of fault mutants present in affected parts of the programs.



(b) Box-plot of the % of mutants killed in affected parts of the programs.

Figure 6.5: Mutation analysis of programs.

test cases using the approach of Panigrahi and Mall and our approach are shown in Figure 6.6 and Figure 6.7, respectively, for the experimental program given in Figure 4.2. The comparison of APFD values for fifteen different programs using our approach and the approach of Panigrahi and Mall [72] is shown in Figure 6.8.

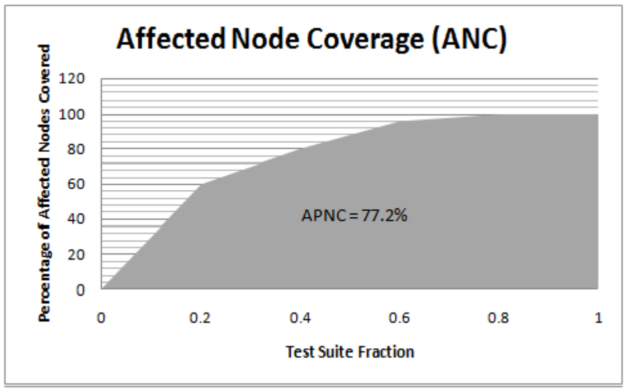


Figure 6.6: Average percentage of affected nodes covered by the prioritized test cases using the approach of Panigrahi and Mall.

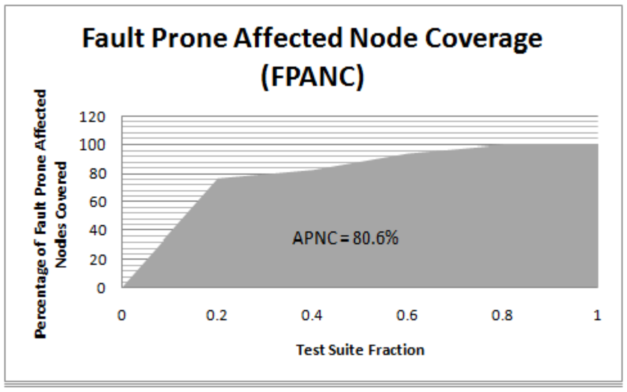


Figure 6.7: Average percentage of fault prone affected nodes covered by the prioritized test cases using our approach.

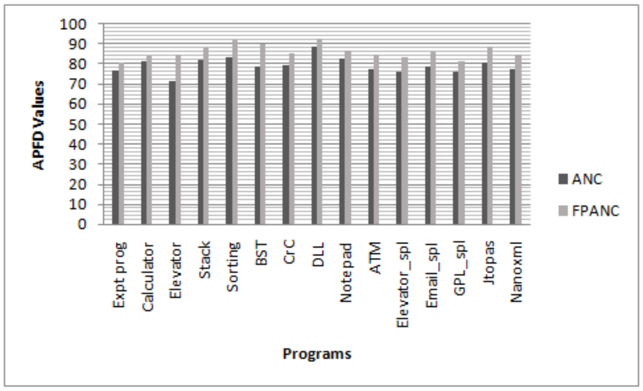


Figure 6.8: Comparison of APFD values for different programs.

6.7.3 Results

Figure 6.5a shows the presence of mutants in percentage in the affected parts of the programs. The presence of mutants in the affected parts of the programs ranges from a minimum of 12% (*DLL* program) to a maximum of 94% (*Sorting* program). The affected program parts in five programs have more that 90% of the mutants and four programs have little more than 10% mutants. The result shows that an average of 47% of mutants are scattered in the affected program parts of the sample programs. Figure 6.5b shows the percentage of mutants killed in each of the experimental programs. The percentage of mutants killed by the prioritized test cases varies from 70% to 95%. The average percentage of mutants killed by the prioritized test suite is 85%. This show that our prioritized test cases are efficient in revealing the faults.

The average percentage of affected nodes covered by the prioritized test cases using the approach of Panigrahi and Mall and our approach are shown in Figure 6.6 and Figure 6.7, respectively, for the experimental program given in Figure 4.2. From Figure 6.6 and Figure 6.7, it may be observed that the average percentage of nodes covered (APNC) using the approach of Panigrahi and Mall [72] is 77.2%. Whereas the APNC value using our approach is 80.6%. Thus, there is an increase of 3.4% in APNC measure by our approach. Hence, our approach detects faults better than the approach of Panigrahi and Mall [72] as our approach covers more number of fault-prone nodes. We evaluated the effectiveness of our approach by using APFD metric. We named Panigrahi and Mall approach [72] as Affected Node Coverage (ANC) and our approach as Fault Prone Affected Node Coverage (FPANC) in Figure 6.8. The comparison of APFD values for these fifteen different programs obtained using ANC and FPANC approaches is shown in Figure 6.8. The results show that our FPANC approach achieves approximately 8% increase in the APFD metric value over ANC approach.

The experimental results show that the performance of our approach varies significantly with program attributes, change attributes, test suite characteristics, and their interaction. To assume that a higher APFD implies a better technique, independent of cost factors, is an oversimplification that may lead to inaccurate choices among prioritization techniques. For a given testing scenario, cost models for prioritization can be used to determine the amount of difference in APFD that may yield desirable practical benefits, by associating APFD differences with measurable attributes such as prioritization time. A prioritization technique would be acceptable provided the time taken is within acceptable limits, which also reflects the cost of retesting. Korel et al. [135] have also focused on less time of execution to decrease the overhead of prioritization process. However, the acceptable time limit greatly depends upon the testing time available with the tester. An empirical analysis on the prioritization time is outside the scope of this paper and is kept for our future work. We have reported the prioritization time of our approach to indicate the time taken to prioritize the test cases when the pre-computed test coverage information and the ASG are available with the tester. The last column of Table 6.6 shows the time taken for prioritizing the selected test cases. The prioritization time varies from a minimum of 1.3 seconds to a maximum of 3.87 seconds for the experimental programs. The total time taken to prioritize the test cases of all the programs is 35.48 seconds and the average time for prioritizing the test cases is 2.4 seconds. The prioritization time includes the time for computing the weights of the test cases and the time taken to order the test cases in decreasing order of their weights.

6.7.4 Threats to Validity

It is obvious for any new proposed work to be associated with some threat to its validity, and it is likely for this work as well.

- This proposed approach, incorporates the effect of the inheritance feature on the coupling value of classes. However, coupling between classes in a subclass-superclass relationship can have a different impact on software maintainability and fault proneness than the coupling of classes that are not in such a relationship. Therefore, it is essential to make a distinction between coupling within an inheritance hierarchy and coupling across inheritance hierarchies. We believe a detail empirical study on such relationships, and their impact on the proposed coupling measurement is essential and is kept for future study.

- Even though the programs under consideration give a good understanding of the proposed approach and are beneficial in validating our approach, but they may not be good representatives of the real-world programs. Hence, this is considered to be a threat to the validity of this approach because of the limited size and complexity of the programs.

- The use of mutation analysis for the fault manipulation of these small programs may not represent the actual fault occurrence in the complex industrial programs and hence, considered a threat to our approach.

- Our approach of prioritization of the test cases is based on the rate of fault detection, which we represent through measure. However, APFD measure is associated with some limitations [66]. APFD measure does not assign any value to the test case for detecting a fault that is already discovered. However, such a fault may be of higher priority and its detection may help the debugging process. APFD does not consider the variations in the weight (in terms of cost) of different faults and test cases. APFD measure may not lead us to a global optimal ordering of the test cases rather constrain us to a local optimal ordering.

6.8 Comparison with Related Work

In this section, we give a comparative analysis of our work with some other related works.

Elbaum et al. [68] performed an empirical investigation to find out the testing scenarios where a particular prioritization approach will prove to be efficient. They analyzed the rate of fault-detection that resulted from several prioritization techniques such as Total function coverage, Additional function coverage, Total binary-diff function coverage, Additional binary-diff function coverage. The authors considered eight C programs for their experimentation. They used the documentation on the programs, and the parameters and special effects that they determined to construct a suite of test cases that exercise each parameter, special effect, and erroneous condition affecting program behavior. Then augmented those test suites with additional test cases to increase code coverage at the statement level. The regression fault analysis was done on the faults inserted by the graduate and undergraduate students with more than two years of coding experience. The experimental results show that the performance of test case prioritization techniques varies significantly with program attributes, change attributes, test suite characteristics, and their interaction. Our results also confirm to similar findings. However, our approach concerns Java programs. We have considered the dependencies caused by the object-oriented features in our proposed intermediate graph. Our approach targets the fault exposing potential and the coverage of affected nodes which is more changed based than the approach in [68].

Korel et al. [115] proposed a model based test prioritization approach. The approach is based on the assumption that execution of the model is inexpensive as compared to execution of the system; therefore the overhead associated with test prioritization is relatively small. This approach is based on the EFSM system models. The original EFSM model is compared with the modified EFSM model to identify the changes. After the changes are identified, the EFSM model is executed with the test cases to collect different information that are used for prioritization. The authors propose two types of prioritization: selective test prioritization and model dependence-based test prioritization. The selective test prioritization approach assigns higher priority to the test cases that execute the modified transitions. Model dependence-based test prioritization mechanism carries out dependency analysis between the modified transitions and other parts of the model and uses this information to assign higher priorities to the test cases. EFSM models consist of two types of dependences: control and data dependences. The results show that model dependence-based test prioritization (considering only two types of dependences) gives improvement in the effectiveness of test prioritization. The corresponding system for each model was implemented in C language. In another work, Korel et al. [114] compared the effectiveness of different prioritization heuristics. The results show that model based prioritization along with heuristic 5 gave the best performance. Heuristic 5 states that each modified transition should have the same opportunity of getting executed by the test cases. Korel et al. [113] proposed another approach of prioritization using the heuristics discussed in [114]. In this new approach, they considered the changes made in the source code and identified the elements of the model that are related to these changes to prioritize the test cases. In our approach, the Java program is represented by our proposed intermediate graph. The graph is constructed by considering many more dependences that exist among the program parts in addition to control and data dependences, giving a clear visualization of the dependences. Then, we identify the effect of modifications and represent the affected program parts in another graph. Our representation is more adaptable to the frequent changes of the software and our approach relies on the execution of these affected program parts. Thus, our prioritization approach is based on both the coverage of the affected program parts and the fault exposing potential of the test cases.

Jeffrey et al. [104] proposed a prioritization approach using the relevant slices. They also aimed for early detection of faults during regression testing process. This approach considers the execution of the modified statements for prioritizing the test cases. The assumption is that if any modification results in some faulty output for a test case, then it must affect some computation in the relevant slice of that test case. Therefore, the test case having higher number of statements is given higher priority assuming they have a better potential to expose the faults. However, intuitively, not all statements depending upon some modification will have the same level of fault-proneness. It may so happen that a test case executing less number of statements will detect more faults than another test case that executed more number of statements. The level of fault-proneness of the statements executed by the test cases affects the fault exposing potential of that test case. Therefore, in our approach, we computed the coupling values of the affected program parts to identify the probable fault-proneness of these programs parts. Our approach assigns a higher priority to that test case which executes maximum number of high fault-prone statements. Further, unlike our hierarchical decomposition slicing approach, relevant slicing depends upon the execution trace of the test cases and is proposed to work on C programs. Even though execution trace based slicing would result in slices of smaller sizes but the computational overhead is very high. The efficiency of our slicing approach is shown in Table 4.5. We have also shown the time requirement of our prioritization approach in Table 6.6.

The performance goal of the prioritization approach proposed by Kayes [108] is based on how quickly the dependences among the faults are identified in the regression testing process. An early detection of the fault dependences would enable faster debugging of the faults. The paper assumes that the knowledge of the fault presence is extracted from the previous executions of the test cases. A fault dependence graph is constructed using this information. However, one major limitation of this approach is that regression testing aims at discovering new faults introduced by the changes made to the software. But, the prioritization approach proposed in this thesis only enhances the chances of finding the faults which has already been revealed and present in the fault dependence graph. New faults if any cannot be discovered. Further, this approach does not take into account the fault-proneness of the statements. However, our approach relies on the dependence of the affected program parts represented as affected slice graph (ASG), so that error propagation because of the change is better visualized and analyzed. We compute the fault-proneness of the statements by computing their coupling values as coupling measures are proven to be good indicator of fault-proneness. Thus, our approach has a higher probability of exposing new faults, if any, in the software.

Mei et al. [148] proposed a static prioritization technique to prioritize the JUnit test cases. This prioritization technique is independent of the coverage information of the test cases. It works on the analysis of the static call graphs of JUnit test cases and the program under test to estimate the ability of each test case to achieve code coverage. The test cases are scheduled based on these estimates. The experiments are carried out on 19 versions of four Java programs of considerable size considering their method and class level JUnit test cases. The heuristic to prioritize the test cases in this approach is to cover system components (in terms of total components covered or components newly covered). The coverage of the system components acts as a proxy for evaluating a test caseâĂŹs true potential of exposing faults. If any two test cases carry the same heuristic value then the approach randomly decides on the test case to be given higher priority. Though this is a scalable approach as it works at coarse granularity level and incurs less computational cost, it suffers from many limitations. The prioritization techniques that work at a finer granularity level give better performances (in terms of fault exposing potential) as compared to the techniques that work at coarse granularity level [68]. This approach ignores the faults caused by many object-oriented features such as inheritance, polymorphism, and dynamic binding and focuses only on the static call relationships of the methods in the form of a call graph. Static call relationships are more to procedure-oriented programs. Interaction and communication between methods in the form of message passing is highly important in object-oriented programs. A single method is invoked by different objects and the behavior of the method also differs accordingly. Any prioritization technique is efficient if it is based on the characteristics of the program to be tested. Therefore, considering the object-oriented features is essential. Java supports encapsulation and provides four access levels (private, public, protected, and default) to access the data members and member methods. Any misinterpretation of these access levels forms a rich source of faults. Java supports a feature named "superâĂİ to have access to the base class constructor from the derived class constructor. This additional dependence between constructors of the derived class and the bases class needs attention of the testers. Method overriding allows a method in the derived class to have the same function signature as the method in its parent. If invocation to such methods is not resolved correctly, then it can cause some serious faults. Another powerful feature and a potential source of fault is variable hiding. It allows declaration of a variable with the same name and type in the derived class as it is in the base class and allows both the variables to reside in the derived class. Problem arises when an incorrect variable is accessed. Inheritance is a powerful feature but sometimes unintentional misuse of this feature can result in serious faults. Polymorphism in Java exists both for attributes and methods and both use dynamic binding. An object of its class type can access an attribute or method of its subclass type. The subclass object can also access the same attributes and methods. These attributes and methods behave differently depending upon the kind of object that is referring it. Such polymorphic dependences if not resolved can cause faults. Interested readers are requested to refer [110, 111, 159] for more number of faults introduced by the misuse of the object-oriented features. Therefore, any prioritization technique with a performance goal of revealing more faults must consider the object-oriented features as they can induce many kinds of faults in the system. Our approach considers all the object-oriented features in the form of intermediate graph. Our approach works at a finer granularity level, therefore may not be as scalable as [148], but has better fault exposing potential.

Fang et al. [69] have proposed similarity based prioritization technique. The authors have taken five Java programs from Software artifacts Infrastructure Repository (SIR) [59] to validate their approach. The prioritization process is based on the ordered sequence of the program entities. They propose two algorithms farthest-first Ordered Sequence (FOS) and greed-aided clustering ordered sequence (GOS). The FOS approach first selects the test case having largest statement coverage. The next test case that is selected is the one that is farthest in distance from the already selected test case. It computes two types of distances: a pair wise distance between the test cases and distance between a candidate test case and the already selected ones. GOS approach consists of clusters of test cases in which initially each cluster consists of only one test case. Then the clusters are merged depending upon the minimum distance between any two clusters. This process of merging the clusters is repeated until the size of the cluster set is less than some given n. Then, the algorithm iteratively chooses one test case from each cluster and adds to the prioritized test suite until all the clusters are empty. The experimental results in this study show that statement coverage is most efficient and preferred for prioritization. When the size of the test suite is large, then additional measures are taken to reduce the cost of prioritization. This approach gives equal importance to all the test cases assuming that all the test cases have equal potential of exposing the faults. Intuitively, a test case executing less number of statements can expose more faults provided the covered statements have high proneness to faults. It also does not consider the object-oriented features and the faults generated by these features. Unlike Fang et al. [69], we consider the fault inducing capability of the object-oriented features based on which we detect the affected program parts. We propose to prioritize a set of *change-based selected* test cases that are relevant to validate the change under regression testing. We compute the fault-proneness of the affected statements and then prioritize the test cases based on the coverage of these high fault-prone statements (represented as nodes in our proposed graph).

Lou et al. [138] proposed a mutation-based prioritization technique. In this approach, they compared the two versions of the same software to find the modification. Then, they generate the mutants only for the modified code. They selected only those test cases of the original version that worked on the new version of the software for prioritization. The test case that killed more mutants was given higher priority. The authors used a mutation generation tool, named *Javalanche*. Unlike our approach, Lou et al. [138] have not considered the object-oriented features and the faults likely to occur because of these features. It is also silent on the type of mutation operators (faults) considered for their experimentation. Like Lou et al.

[138], we generate mutants only for the sliced program (representing the affected program parts). However, we used MuClipse (an eclipse version of MuJava) to generate the mutation faults. We use coupling measure of the affected program parts as a surrogate to imply fault-proneness. Our hypothesis assumes that the test cases that execute the nodes with high coupling value have a higher chance of detecting faults early during regression testing. We used mutation analysis only to validate our hypothesis.

The detail survey conducted on available coverage based prioritization techniques [68, 69, 104, 108, 113, 114, 115, 138, 148] reveals that these techniques have not considered the object-oriented features. The presence of many faults arising due to different object-oriented features are inherent to object-oriented programs, and hence must be considered. Therefore, we find that the approaches contributed by Panigrahi and Mall [162, 163] relates closely to our approach for an experimental comparison. Panigrahi and Mall proposed a version specific prioritization technique [162] to prioritize the test cases of object-oriented programs. Their technique prioritizes the selected regression test cases. The test cases are prioritized based on the coverage of affected nodes of an intermediate graph model of the program under consideration. The affected nodes are determined due to the dependences arising on account of the object relations in addition to the data and control dependences. The effectiveness of their approach is shown in form of improved APFD measure achieved for the test cases. In another work, Panigrahi et al. [163] have improved their earlier work [162] by achieving a better APFD value. In this technique, the affected nodes are initially assigned a weight of 1. The weight is decreased by 0.5, whenever that node is covered by previous execution of the test cases. In both the approaches [162, 163], they have assumed that all the test cases have equal cost, and all faults have same severity. The assumption is also that all the affected nodes have a uniform distribution of faults. As a result, a test case executing more number of affected nodes will detect more faults and therefore, has a higher priority. The average percentage of affected nodes covered by this approach is shown in Figure 6.7. Unlike the approach in [163] that is based on node coverage only, our proposed approach is based on the fact that some nodes are more fault-prone than other nodes. We used an intermediate graph that represents only those nodes that are affected by the modification made to the program to compute the fault-proneness of the nodes. The coupling factor of each node in the ASG is computed to predict its level of fault-proneness. The test cases are then prioritized based on the fault-prone nodes that they execute. Unlike [163], a test case executing more number of fault-prone nodes has a higher computed weight and gets a higher priority in our approach.

6.9 Summary

In this chapter, we proposed a coupling metric based technique to improve the effectiveness of test case prioritization in regression testing. Analysis is done to show that prioritized test cases are more effective in exposing the faults early in the regression test cycle. We performed hierarchical decomposition slicing on the intermediate graph of the input program. The slice obtained is then modeled as a graph named affected slice graph (ASG). The affected component coupling (ACC) value of each node of the ASG is calculated as a measure to predict its fault proneness. In this technique, weight is assigned to each node of ASG based on its ACC value. The weight of a test case in a given test suite is then calculated by adding the weights of all the nodes covered by it. The test cases are prioritized based on their coverage of fault prone affected nodes. Thus, the test case with a higher weight is given higher priority in the test suite. The results show that our FPANC approach achieves approximately 8% of increase in the APFD metric value over ANC approach. The regression testing problem lay equal focus in validating the correctness of the software for every change made to it. Intuitively, not all software changes require same amount of testing. Therefore, some metrics are required that would highlight those changes that require more attention of the tester. We address this issue of identifying and quantifying the effect of the software changes in the next chapter.

Chapter 7

Identifying and Quantifying the Effect of Changes

Many times, the testers may not have the liberty of exhaustive retesting of every change made to the program in a looming scenario of time and cost (due to project deadline, customer impatience, market pressure, etc.). Many aspects of regression testing (such test case selection, prioritization, and minimization) can help the tester to overcome the retest-all approach in demanding situations. The tester can make a decision regarding what to test and what not to test, and also in what order to test. But, it is essential for the tester to decide what to retest more, what to retest less, and when to stop testing [75]. This is because, obviously not all changes would require the same amount of retesting to ensure that the software is compliant. Some changes may be more severe than others. Thus, retesting to validate the changes requires the testers to harness the art of testing less and selective without sacrificing the quality [98]. Even though it is desirable to selectively run the tests [27, 90, 188, 210], but this approach may suffer from a potential threat of missing out the critical defects. Hence, minimizing and then prioritizing the test cases [57, 67, 132, 148, 163, 177, 184] allow some test cases with high potential of revealing defects to execute early, and discover more faults with fewer tests. But, still the regression testing process can become very costly if the testing effort is not properly distributed for handling the different types of changes.

Thus, the goal of this chapter is to propose some metrics that can quantify the severity of the changes made to a program. These metrics will act as indicators for the testers to decide what to test more and what to test less without sacrificing the product quality. We make the following contributions in this chapter:

- Constructing a graphical representation of the dependences that exist between the various changes to identify the clusters of changes (discussed in Section 7.2.2).
- Proposing metrics that can indicate the severity of the changes (given in Section 7.2.3).
- Making an assessment of the usefulness of these metrics (discussed in Section 7.3.2).

The rest of the chapter is organized as follows: Section 7.1 introduces the technique used for change impact analysis. We justify the use of this technique by showing its advantages over another existing approach. We describe the program changes and define the change metrics in Section 7.2. In this section, we discuss the structure of our program model, describe the cluster of changes, define metrics for these changes, and demonstrate the calculation of these metrics. The details of our experimental studies are presented in Section 7.3. Here, we describe the characteristics of the program samples taken for our experimentation, and analyze the results to justify our hypothesis. In Section 7.4, we discuss and compare our work with some related work. We also highlight some of the limitations of our approach in this section. We summarize the chapter in Section 7.5.

7.1 Background

In this section, we briefly discuss the basic ideas of the techniques used to realize the proposed approach.

7.1.1 Change Identification

Before we can estimate the severity of the changes, it is essential to identify the changes. Then only, we can perform a change impact analysis. To determine and track the series of changes made by several stakeholders is a major problem. This problem becomes more evident when changes made by one group are regression tested by another group. In this thesis, we assume, the group that makes the changes is the same group that carries the regression testing. Every change made to the program is maintained globally as a *change set*. After the changes are made and identified (recorded), it is essential to analyze their ripple effect on other parts of the program. A clear understanding of the ripple effects not only saves cost

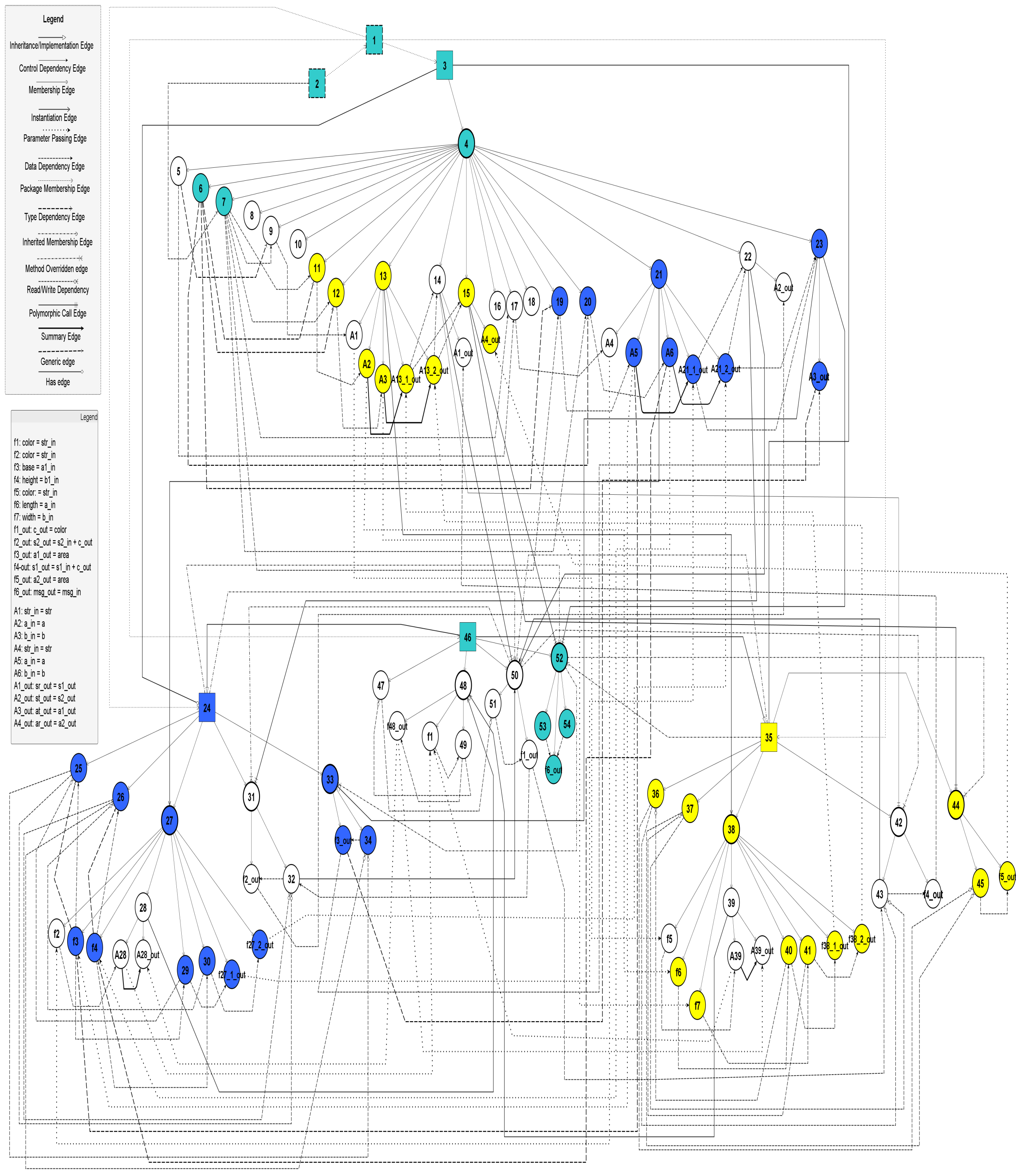


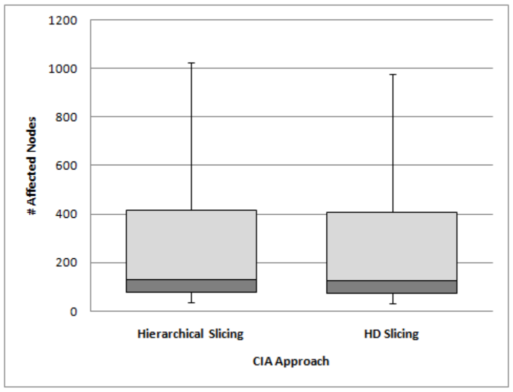
Figure 7.1: Change Ripple Graph (CRG) of the example Java program given in Figure 4.2.

but also helps in effective regression testing. The testers can focus only on the affected program parts instead of testing the whole program. Manually finding these affected program parts even with an available change set is a notoriously hard problem. Thus, we the slicing technique discussed in Section 4.2.3 to find these affected program parts. These affected program parts are then analyzed to determine the amount of code that the change crosscuts across the program. The crosscutting aspect enables us to quantify the scattering of change across the affected program parts (statements, methods, classes, and packages) that often tangles with other changes. Many research work are available on prioritization and minimization of test cases based on the results of change impact analysis, but no guidance is available on how to distribute the regression testing effort to validate these changes, and ensure quality.

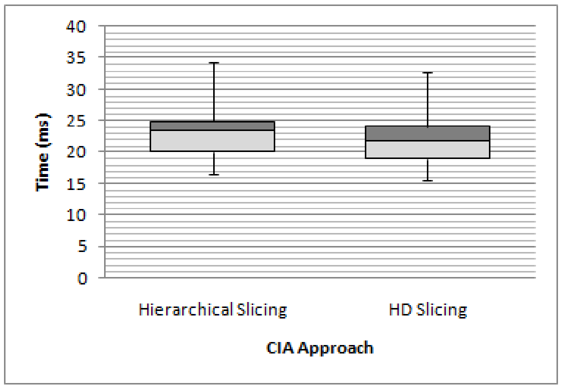
7.1.2 Change impact analysis (CIA)

As mentioned in the previous section, we assume that the same group of people those who make the changes also carry out the regression testing. This group records these changes in a *change set C*, to refer later for the necessary impact analysis. A list of ten different types of changes considered in our experimentation is shown in Table 4.3. We use an existing program slicing technique, named *hierarchical decomposition (HD) slicing* discussed in Section 4.2.3, to make the required analysis. An earlier work on hierarchical slicing [130] based CIA [186] inspires the HD slicing approach used in this thesis for CIA. The reason for opting *HD slicing* approach is because of its relative advantages over the hierarchical slicing technique as shown in Table 4.5. HD slicing works on an intermediate graphical representation of the dependences that exist between various program parts of the input program. This intermediate graph is same as the *extended object-oriented system dependence graph (EOOSDG)* discussed in Section 4.2.1. EOOSDG is renamed as *change ripple graph (CRG)* in this chapter as it shows the ripple impact of different changes made to the corresponding program. An example Java program taken as our running example is shown in Figure 4.2. The CRG for the example program given in Figure 4.2 is shown in Figure 7.1. The nodes of the graph correspond to the statements of the program and edges correspond to the dependences [119, 198]. The blue nodes and yellow nodes mark the ripple effect of the changes made at *node 23* and *node 15*, respectively. The cyan nodes are the common nodes that are affected by both the changes. The point of modification (*node 15* and *node 23* in Figure 7.1), $c \in C$, are considered as the slicing criterion to find the effect of change. The slicing algorithm discussed in Section 4.2.3 traverses through the edges (dependences) from the slicing criterion in three phases to compute the slice. Thus, the slice denotes the ripple effect of the change. The computed slice is then hierarchically decomposed into affected packages,classes, methods, and statements. The boxplot in Figure 7.2 shows the comparison between our approach and the approach in [186] in terms of nodes discovered (7.2a) and time taken (7.2b) to compute the affected program parts in the CRG (Figure 7.1) of the example program (Figure 4.2). In both the comparisons, HD slicing has clear advantages over [186]. The steps to identify the affected program parts using *HD slicing* are given in Section 5.2.3.

The CIA results in a slice that is represented in the form of a graph, named *affected slice graph (ASG)*, as shown in Figure 5.2. Each node in the ASG corresponds to the statement affected by the change, and each edge corresponds to the dependence between them. Thus, ASG consists of a set of CRG nodes V and a set of edges E of the form $e_{i,j}$, where $\forall\ e_{i,j} \in E$, $v_j \in HDslice(v_i)$.



(a) Node comparison.



(b) Time comparison.

Figure 7.2: Comparison between CIA approaches.

7.2 Proposed Metrics for Describing Program Changes

The proposed metrics are based on the hypothesis that the *changes* made to a program can be treated as *concerns* of the program. The term *concern* is loosely defined to represent any consideration of the stakeholders of a software project that can impact the implementation of the program [63, 141]. Thus, a change made to the program can rightly be defined as a concern of the tester (developer) that can affect the correctness of the program. Thus, our hypothesis needs to be proved before the metrics can be defined.

Hypothesis 1: *A change made to the program can be defined as a potential concern of that program.*
Justification:
A change can be identified and defined as a concern provided it satisfies the identification guidelines laid down for concerns. Below, we show how a change satisfies these guidelines to qualify as a concern. Since, we state the guidelines in the context of program changes, these are termed as *Change Impact Guidelines (CIG)* [52, 53, 62] and are given below:

CIG 1. *Objective and definitive membership criteria*
The first guideline states that any concern should have an objective, i.e. there should be no ambiguity in the identification of the concern. Something that is a concern to one stakeholder should remain valid for another. As discussed in Section 7.1.1, we have stated that the group that makes the changes is the same group to carry out the regression testing, and also record the details of the changes in a list. Therefore, there is absolutely no ambiguity for any stakeholder in identifying the changes. The definitive membership criteria answer if some $'X'$ is a concern that requires some necessary analysis. In the context of a change made to the program, the answer is obviously affirmative. Otherwise, there would not have been any necessity of CIA and regression testing. Since, CIA and regression testing are indispensable in the domain of maintenance, so the changes are defined as concerns.

CIG 2. *Finite Domain*
The finite domain criteria state that the number of concerns should be limited. As stated in Section 7.1.1, we have C_c as a set of changes for $1 \leq c \leq m$. Thus, the cardinality of the change set is, $|C_c| = m$. Since, m is a small positive integer, the number of changes made to the program is finite.

Therefore, any *change* made to the program can be termed as a potential *concern* of the program.

7.2.1 Structural program model

Here, we describe the various program elements that can get affected by the changes made to the program. The program that undergoes changes is formally represented as a graph $CRG = (V_a, E_a)$, where V_a is the set of vertices corresponding to program statements, and E_a is the set of directed edges $e = (v_1, v_2)$ corresponding to the dependences between the program parts. Interested readers can refer [187] for detail definitions on various dependences considered in CRG. A vertex in CRG is of any of the following types:

- **Package vertex (Pk)**, that is connected to sub-packages and classes connected by *package membership edge*.
- **Class vertex (Cl)** is connected to its data (field) member and member method vertices by *membership edges*.
- **Field vertex (F)**, represents a data member of a class.
- **Method vertex (Mt)**, represents a member method of a class connected to its statement vertices (St) that belong to the method by membership edges.

The HD slice at some change point c is defined on CRG as follows:

$$HDslice(c) = HDslice_f(c) \cup \{ \bigcup_{k=1}^{|HDslice_f(c)|} HDslice_b(v_k)\}, v_k \in HDslice_f(c)$$

$$HDslice_f(c) = \{c, v_1, v_2, \ldots, v_k \mid (c, v_1), (v_1, v_2), \ldots, (v_l, v_k) \in E', E' \subseteq E_a, E_{spl} \notin E', 1 < l \leq k\}, E_{spl} = \{\overset{imem}{\rightarrow}, \overset{poly}{\rightarrow}, \overset{par-out}{\rightarrow}, \overset{generic-out}{\rightarrow}\}$$

$$HDslice_b(v_k) = \{v_1, v_2, \ldots, v_j \mid (v_1, v_2), \ldots, (v_j, v_k) \in E'', E'' \subseteq E_a, E_{spl}{}' \notin E'', v_k \in HDslice_f(c)\}, E_{spl}{}' = \{\overset{par-in}{\rightarrow}, \overset{generic-in}{\rightarrow},\ any\ e \in E'\}$$

7.2.2 Proposed Change Cluster Graph (CCG)

The dependences between the changes made to a program can be represented in a graphical form, which is defined as follows:

Definition 7.1. *A change cluster graph (CCG) consists of a set of vertices V_C (that corresponds to the change points) and a set of edges E_C of the form $e_{i,j}$, where $\forall e_{i,j} \in E_C, v_j \in HDslice_f(v_i) \implies v_i \in HDslice_b(v_j)$.*

The CCG for the example program given in Figure 4.2 is shown in Figure 7.3. We partition the CCG shown in Figure 7.3 into clusters based on the concept of

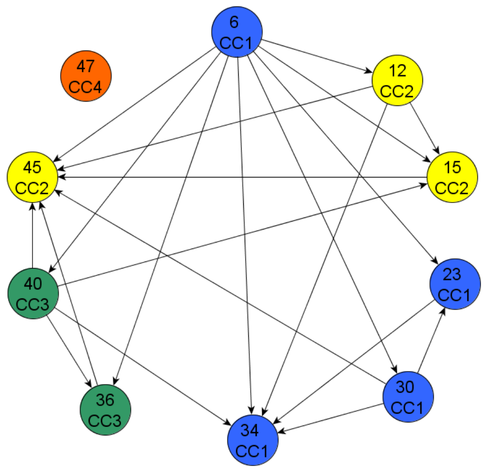


Figure 7.3: Change Cluster Graph (CCG) of the example Java program given in Figure 4.2.

dependence community [83]. A change cluster CC is a set of nodes (changes) in CCG defined as

$$CC = \{v_{i,j} \mid \forall i,j \; v_i \in HDslice(v_j), i \neq j, 1 \leq i,j \leq |CCG|\}.$$

The result shows the presence of four clusters in the CCG shown in Figure 7.3. Each node is labeled with a number (corresponding to the node in CRG where the change is made). These four clusters are: $CC_1\{6,23,30,34\}$, $CC_2\{12,15,45\}$, $CC_3\{36,40\}$, and $CC_4\{47\}$. Each cluster is marked with a separate color and labeled accordingly. The properties of the clusters are presented below:

- Clusters cannot overlap, i.e. $CC_1 \cap CC_2 = \phi$. Therefore, any node that can be a member of any two clusters is randomly assigned to any one of them. For example, *node 6* satisfies the condition to either belong to cluster CC_1 or CC_2, we assigned it to cluster CC_1.

- A cluster can comprise of a single node to all the nodes of the CCG, i.e. $1 \leq |CC| \leq |CCG|$.

A rank is given to the clusters in the decreasing order of their cardinality. The cluster having highest cardinality is given the highest rank. If any two clusters have the same cardinality, then both are given the same rank. The rank indicates the complexity of the changes in the form of their interdependence. If a change clusters with many other changes, then it is tough to validate that change. Since, cluster

CC_4 contains only one change, so it gives a sense that the change made at *node 47* is easy to handle. CCG provides an initial assessment of the complexity and risk of the changes made. Based on the ranking of these change clusters, the testers can measure the metrics described next to find the severity of the changes that belong to these clusters of high rank.

7.2.3 Definition of the proposed metrics

Let C be a set of changes organized into C_C, defined as

$$C_C = \{(C_i, C_j) \mid C_i, C_j \in C, C_i \neq C_j, i \neq j, 1 \leq i, j \leq n\},$$

and I be a set of program parts organized into C_I, defined as

$$C_I = \{(I_i, I_j) \mid I_i, I_j \in I, I_i \neq I_j, i \neq j, 1 \leq i, j \leq m\}.$$

Every program part I_j refers to a node in CRG that has an outgoing *membership* or *package membership* edge.

$$I_j(v_k) = \{v_i \mid (v_k, v_i) \in \{\overset{mem}{\rightarrow}, \overset{pkg-mem}{\rightarrow}\}, 1 \leq i \leq n\},$$

where n is a positive integer. Thus, every program part $I_j \in C_I$ is defined as the set of nodes connected by either membership or package membership edge.
Let $f : C_C \rightarrow C_I$ and $g : C_I \rightarrow C_C$ be two mapping functions defined as follows:

$$\forall C \in C_C, f(C) = \{I \in C_I : f'(C) = I\},$$

$$\forall I \in C_I, g(I) = \{C \in C_C : g'(I) = C\}.$$

Definition 7.2. *(Scattering). A change is said to be scattered, if its effect ripples from the point of origin across multiple program parts, i.e.* $|f(C)| > 1$.

Definition 7.3. *(Tangling). A program part is said to be tangled, if it has the impact of multiple changes, i.e.* $|g(I)| > 1$.

Definition 7.4. *(Crosscutting). Let* $C_1, C_2 \in C_C$, $C_1 \neq C_2$, *implies* C_1 *crosscuts* C_2, *if* $|f(C)| > 1$ *and* $\exists\ I \in f(C_1) : C_2 \in g(I)$. *Thus, we can have the following lemma on crosscutting changes defined in terms of the computed slices.*

Lemma 1. *Let* C_1, *and* C_2 *be the points of change, and* $C_1 \neq C_2$. *Then* C_1 *crosscuts* C_2 *iff* $|HDslice(C_1)| > 1$, *and* $HDslice(C_1) \cap HDslice(C_2) \neq \phi$.

Definition 7.5. *Concentration (CONC) measures the number of nodes affected by the change that are contained within a component, I. Thus, $CONC$ is defined as follows:*

$$CONC(C_i, I_j) = \frac{|HDslice(C_i) \cap I_j|}{|HDslice(C_i)|},\ 1 \leq i \leq n,\ 1 \leq j \leq m,$$

where n refers to the number of changes and m refers to the number of program parts.
Now, to find the concentration that change C_i has in other components, we define Degree of Change Scattering (DOCS) as follows:

$$DOCS(C_i) = 1 - \frac{|I| \sum_{j=1}^{m} (CONC(C_i, I_j) - \frac{1}{|I|})^2}{|I| - 1}$$

where I is the set of nodes and $|I| > 1$.

Thus, $DOCS$ is a measure of the range of impact of a change over all program components. $DOCS$ is represented in the context of change that was originally proposed as a concern metric (*Degree of Scattering (DOS)*) by Eaddy et al. [62, 63]. $DOCS$ satisfies the following properties:

- $DOCS$ value varies between 0 (completely localized) and 1 (uniformly affects all program parts).

- $DOCS$ is directly proportional to the number of components affected by the change.

- $DOCS$ is inversely proportional to concentration.

$DOCS$ also gives an indication on the modularity of the program. An important characteristic of a component is that its implementation should be localized. Therefore, the components across which the impact of change has scattered are less modular. Thus, $DOCS = 0$ implies high modularity.

Definition 7.6. *Impact (IMP) measures the number of nodes contained within a component I those are affected by the change C_i. Thus, IMP is defined as:*

$$IMP(I, C_i) = \frac{|HDslice(C_i) \cap I_j|}{|I_j|},\ 1 \leq i \leq n,\ 1 \leq j \leq m,$$

where n refers to the number of changes and m refers to the number of program parts.

Now, to find the impact that other changes have on component I_j, we define Degree of Change Focus (DOCF) as follows:

$$DOCF(I_j) = \frac{\sum_{i=1}^{n}(IMP(I_j, C_i) - \frac{1}{|C|})^2}{|C| - 1}$$

where C is the set of changes and $|C| > 1$.

$DOCF$ is a measure that shows the extent to which the changes are tangled. The characteristics of $DOCF$ are:

- $DOCF$ value ranges between 0 (completely unfocused, i.e. uniformly affected by all changes) and 1 (completely focused).
- $DOCF$ is inversely proportional to the number of changes affecting a component.
- $DOCF$ is directly proportional to the impact, i.e. the more uniform is the impact of change; the lower is the focus, thus higher is the tangles.

Thus, it is desirable that a change in the program should have low degree of scattering and high degree of focus. Therefore, any change having high scattering and low degree of focus requires more effective retesting. However, the granularity of measurement should be pre-defined, i.e. whether the metrics are computed for statements, methods, classes, or packages, should be stated clearly.

7.2.4 Metrics Computation

In this section, we discuss the method to compute the metrics proposed in the previous section (Section 7.2.3). The CRG in Figure 7.1 shows the ripple impacts of the two changes made to the program in Figure 4.2. These changes are made at *Line 23 (node 23)* and *Line 15 (node 15)*, named $C1$ and $C2$, respectively. We demonstrate the computational steps concerning these two changes, thus $C = \{C_1, C_2\}$ and $|C| = 2$. By the definition of I given in the previous section and in the context of the CRG in Figure 7.1, we have $I_1(1) = \{1, 3, 46, 24, 35\}$, $I_2(3) = \{3, 4\}$, $I_3(46) = \{46, 47, 48, 50, 52\}$, $I_4(24) = \{24, 25, 26, 27, 31, 33\}$, ..., $I_{23}(13, A1, A2, A3, A13\_1\_out, A13\_2\_out)$. The numbers in the curly braces refer to the nodes of CRG in Figure 7.1. Thus, $I = \{I_1, I_2, \ldots, I_{23}\}$ and cardinality of I is 23, i.e. $|I| = 23$. By Definition 7.5, the concentration of change C_1 in I_4 is given as: [1]

$$CONC(C_1, I_4) = \frac{|HDslice(C_1) \cap I_4|}{|HDslice(C_1)|} = \frac{5}{32} = 0.15625$$

Similarly, the concentration of change C_1 in I_4 is

$$CONC(C_2, I_4) = \frac{|HDslice(C_2) \cap I_4|}{|HDslice(C_1)|} = \frac{0}{32} = 0.$$

It can be seen in Figure 7.1 that C_2 does not affect I_4 (node 24), therefore the intersection result is ϕ. In this way, the concentration of C_1 and C_2 in the rest of the program parts are computed. $DOCS$ is computed for every change as per the formula in Definition 7.5. Thus, the normalized variance of the concentration of C_1 in all the program parts gives us $DOCS(C_1)$. Thus,

$$DOCS(C_1) = 1 - \frac{|I| \sum_{j=1}^{23}(CONC(C_1, I_j) - \frac{1}{|I|})^2}{|I| - 1} = 1 - \frac{23 * \sum_{j=1}^{23}(CONC(C_1, I_j) - \frac{1}{23})^2}{23 - 1}$$
$$= 0.845836293$$

Coincidentally, $DOCS(C_1) = DOCS(C_2) = 0.845836293$. Therefore, average DOCS is $ADOCS = 0.845836293$, which is a high value and signifies that both these changes are highly scattered. Recall from Figure 7.3 that C_1 and C_2 belong to different clusters, but $ADOCS$ value shows that these are highly scattered even though they have no interdependence. Now, the next objective is to check whether the affected program parts of C_1 have impact of other changes as well. If $HDslice(C_1) \cap HDslice(C_2) \neq \phi$, then it implies that there are some program parts that have the impact of both C_1 and C_2. We compute the amount of impact that the changes can have on a affected program part by using the impact formula given in Definition 7.6. Thus, impact of C_1 on I_4 is given by

$$IMP(I_4, C_1) = \frac{|HDslice(C_1) \cap I_4|}{|I_4|} = \frac{5}{6} = 0.833333333$$

Similarly, the impact of C_2 on I_4 is given by

$$IMP(I_4, C_2) = \frac{|HDslice(C_2) \cap I_4|}{|I_4|} = \frac{0}{6} = 0$$

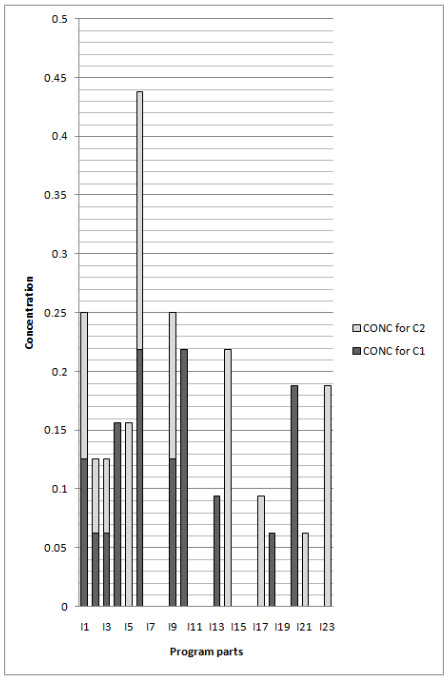
Now, to assess the overall impact of all the concerned changes on the affected program part, we compute $DOCF$ using the formula given in Definition 7.6. Thus,

[1] $|HDslice(C_1)| = 33$, since node 2 refers to a library package in the program given in 4.2, it has been excluded from computation.

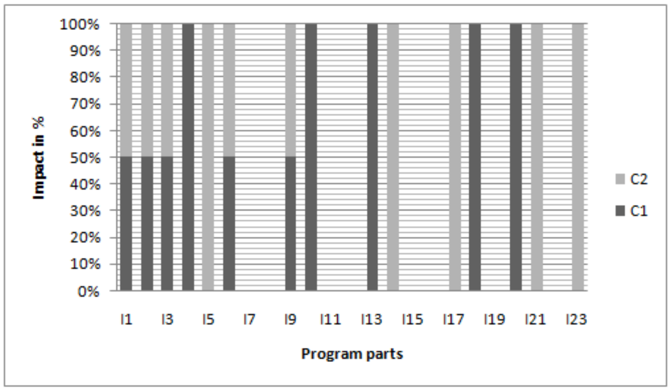
$DOCF(I_4)$ is expressed as follows:

$$DOCF(I_4) = \frac{\sum_{i=1}^{2}(IMP(I_4, C_i) - \frac{1}{|C|})^2}{|C| - 1} = \frac{(0.833333333 - \frac{1}{2})^2 + (0 - \frac{1}{2})^2}{2 - 1}$$
$$= 0.361111111$$

Similarly, $DOCF$ of all other program parts are computed. Thus, the average of $DOCF$ values is $ADOCF = 0.432241009$, which is a low value that signifies that the impact of the changes are tangled in some of the program parts. We consider the average value as low or high based on the median value. If $average < median$, then it implies that the value is low, otherwise it is high. Thus, from the $ADOCS$ and $ADOCF$ values, we conclude that the two changes, C_1 and C_2, are highly scattered and tangled. Therefore, the testers have to spend more time to validate these changes. Figure 7.4 shows the concentration and impact of C_1 and C_2 in various program parts of the example program given in Figure 4.2. Both Figure 7.4a and 7.4b show the degree of scattering and tangling of these two changes.



(a) Concentration of C_1 and C_2 in different program parts.



(b) Impact of C_1 and C_2 in different program parts.

Figure 7.4: Change impact analysis of the two changes made to the program given in Figure 4.2.

7.3 Experimental Studies

In this section, we present the characteristics of the program samples that are considered for our study. We elaborate on the experimental results for our change metrics. We also analyze the results to check whether our hypotheses hold good or not. We follow the same experimental settings as described in Section 4.3.2.

Table 7.1: The list of the sample programs used in the study.

Sl. No.	**Programs**	**LOC**	**No. of CRG Nodes**	**Avg. ASG Nodes**	**Total # of Test Cases**	**# Mutants**
1	Expt. Program	54	91	33	20	14
2	Calculator	75	499	51	15	42
3	Elevator	90	532	54	25	27
4	Stack	114	325	72	22	35
5	Sorting	130	542	86	16	43
6	BST	130	2754	74	20	51
7	CrC	261	1897	94	18	46
8	DLL	277	1838	83	24	47
9	Notepad	300	2234	68	17	17
10	ATM	900	3461	727	33	39
11	Elevator_spl	1046	5362	864	15	53
12	Email_spl	1233	5548	562	18	18
13	GPL_spl	1713	12904	803	22	22
14	Jtopas	5400	24073	7462	16	28
15	Nanoxml	7646	26451	1132	14	32

7.3.1 The sample programs

We conducted the experiments on fifteen medium-sized programs of different specifications as shown in Table 7.1. Out of these fifteen programs, ten benchmark programs (Stack, Sorting, BST, CrC, DLL, Elevator_spl, Email_spl, GPL_spl, Jtopas, Nanoxml) are taken from Software-artifact Infrastructure Repository (SIR) [59] and other five programs are developed as academic assignments. These smaller programs are chosen to ascertain the correctness and accuracy of the approach, keeping in mind that they represent a variety of Java features and applications, the test cases are available or can be easily developed, and coverage information can be computed.

The smallest program has 54 LOC, and the largest program has 7646 LOC. The total LOC for all the fifteen programs is 19369 and the average LOC per program are 1291. The fifteen CRGs are constructed using our prototype tool. The smallest CRG has 91 nodes, and the largest has 26451 nodes. The total number of nodes for all the fifteen CRGs is 88511, and the average number of nodes per CRG is 5901. The smallest ASG has 17 nodes, and the largest has 533 nodes. The total number of ASGs computed at each point of change is 150. The total number of affected nodes in all the fifteen programs due to all the 150 changes is 22800, and the average number of nodes per each change is 152. The total number of fault mutants for all the fifteen programs is 514, and the average number of mutants per program is 34. Similarly, the total number of test cases considered for all the programs is 295 with a mean of 20 test cases per program.

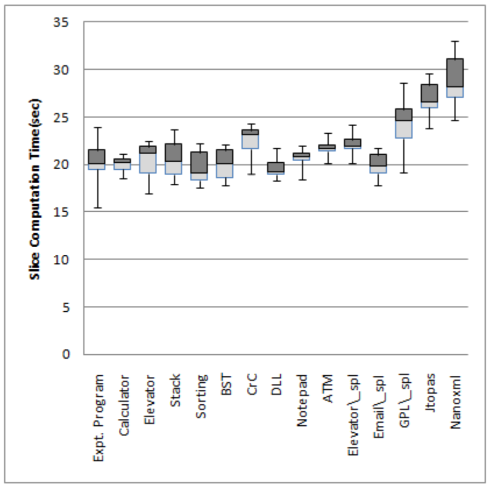
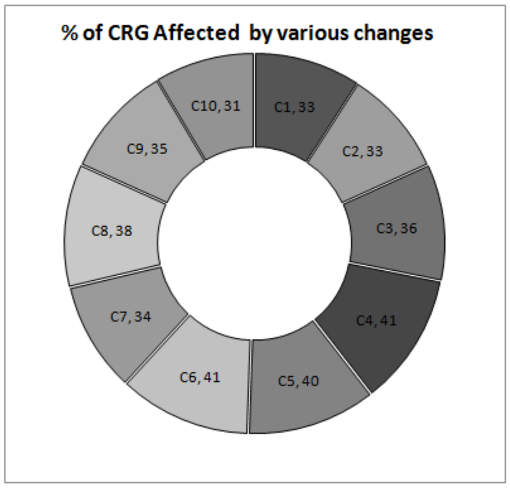


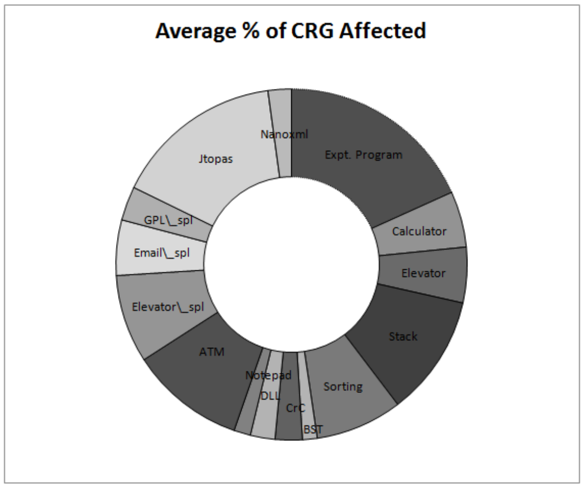
Figure 7.5: Box-plot of the time taken to compute the slices of the sample programs.

7.3.2 Observations

In this section, we analyze the experimental results to assess the usefulness of the proposed metrics, i.e. whether the proposed metrics can give the required information to the testers in a reasonable amount of time. It is observed that the proposed metrics can be computed in less than 1 second (with a very few exceptions) for the changes made to the programs, provided the corresponding slices at the point of changes are pre-computed. The boxplot in Figure 7.2b shows the time taken to compute the slices. The box-plot shows that our approach is better and can calculate the slices more accurately in less time as compared to an existing CIA approach [186]. Figure 7.5 displays the boxplot of the time taken to calculate the slices of all the sample programs. The lowest time taken is 15.52 sec and the highest time



(a) Percentage of nodes affected by the different changes made to the example program given in Figure 4.2.



(b) Average percentage of affected CRG nodes for all programs.

Figure 7.6: Change ripple analysis of programs.

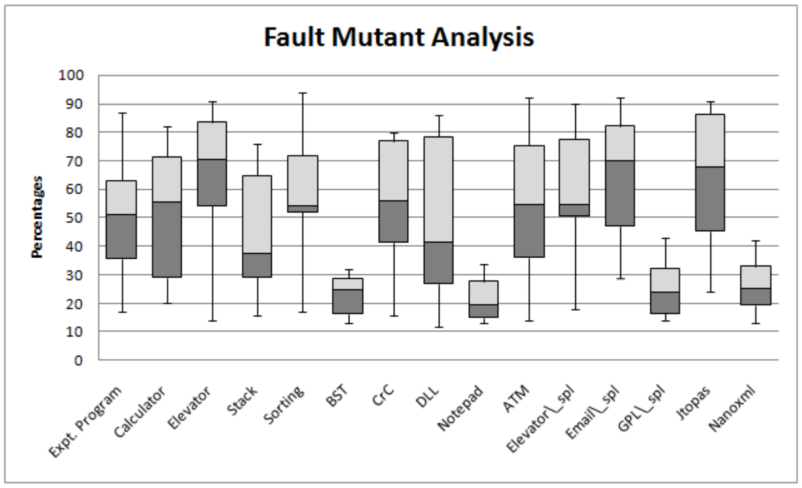


Figure 7.7: Box-plot of the percentage of fault mutants present in affected parts of the programs.

taken is 33.02 sec. The total time to calculate the 150 slices for all the ten changes in fifteen sample programs is 3268.39 sec. The average duration to calculate the proposed metrics ($DOCS$, and $DOCF$) is 0.68 sec.

We are also interested in analyzing the effect that the changes have on other program parts, fault-proneness of the affected parts, and the fault revealing capability of the affected test cases. Therefore, we compute the percentage of CRG

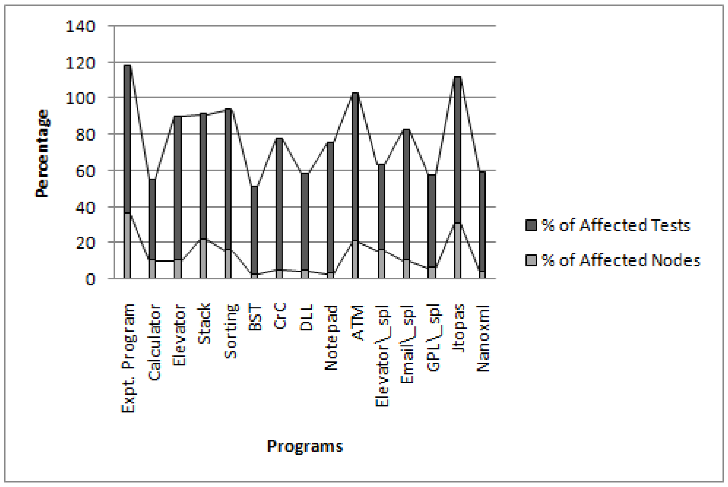
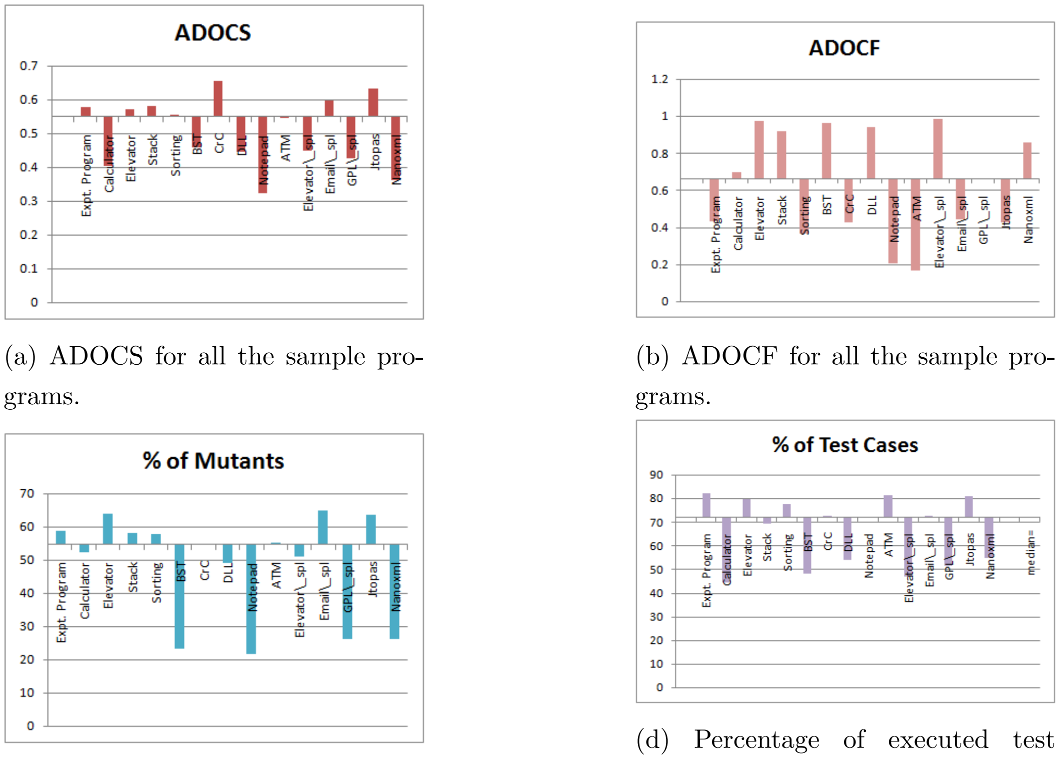


Figure 7.8: Average percentage of affected nodes versus affected test cases.



(a) ADOCS for all the sample programs.

(b) ADOCF for all the sample programs.

(c) Percentage of mutants.

(d) Percentage of executed test cases.

Figure 7.9: Crosscutting change analysis.

nodes that are affected by these changes to make a correct assessment of the ripple impact of the changes. The change ripple analysis result is shown in Figure 7.6. The percentage of CRG (in Figure 7.1) affected by the different changes made to

Table 7.2: Degree of scattering and focus of the sample programs.

Sl.No.	Program	ADOCS		ADOCF	
		High	Low	High	Low
1	Expt. Program	✓			✓
2	Calculator		✓	✓	
3	Elevator	✓		✓	
4	Stack	✓		✓	
5	Sorting	✓			✓
6	BST		✓	✓	
7	CrC	✓			✓
8	DLL		✓	✓	
9	Notepad		✓		✓
10	ATM	✓			✓
11	Elevator_spl		✓	✓	
12	Email_spl	✓			✓
13	GPL_spl		✓	✓	
14	Jtopas	✓			✓
15	Nanoxml		✓	✓	

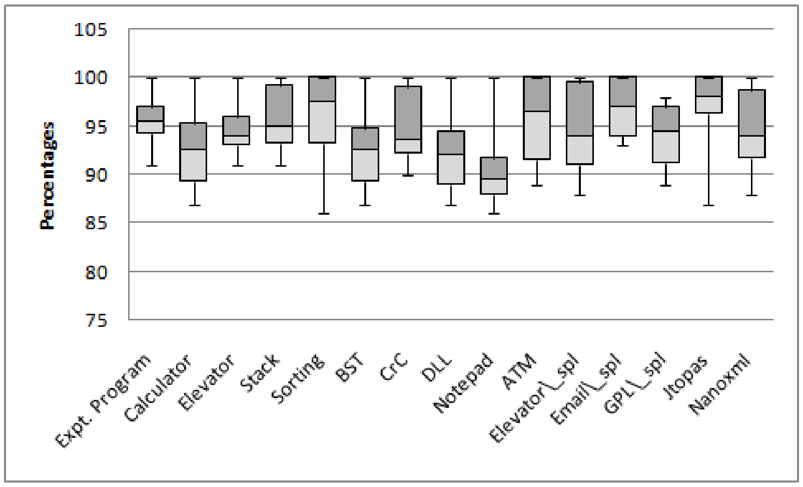


Figure 7.10: Box-plot of the percentage of faults detected in the sample programs.

our running example program is shown in Figure 7.6a. Each sector in Figure 7.6a displays the change and its corresponding percentage of nodes that it affects. The average percentage of the affected nodes is 36.2% for all the ten changes made to the example program given in Figure 4.2. The average percentage of nodes affected by the changes in each sample program is shown in Figure 7.6b. The results indicate that 25.75% of the nodes are affected by all the changes made to the sample programs. We seeded the programs with fault mutants [140, 160] using MuJava [140] to study the fault-proneness of the program parts affected by the changes made to these programs. A fault mutant is associated with a change C if the location of the mutant belongs to $HDslice(C)$. The last column in Table 7.1 shows the number of mutants considered for each program. The boxplot in Figure 7.7 displays the result of the fault analysis for all the sample programs. The presence of mutants in the corresponding changes varies from a minimum of 12% (*DLL* program) to a maximum of 94% (*Sorting* program). The changes in five programs have more that 90% of the mutants and four programs have little more than 10% mutants. The result shows that an average of 47% of mutants are scattered in the affected program parts of the sample programs.

The average percentage of the affected program parts and the average percentage of test cases that are affected by the changes are shown in Figure 7.8. A test case is said to be affected if it executes the affected nodes. Next, we computed the ADOCS and ADOCF for the programs to justify our hypothesis that the metrics proposed in this chapter are useful indicators of the presence of faults. These metrics are also helpful in the initial assessment of the testing effort that is required to detect these faults. We show the results of crosscutting change analysis in Figure 7.9 to see if our hypothesis holds good. The sub-figures in Figure 7.9 display the result in columns distributed around the median axis to differentiate the low and high values. Figure 7.9a displays the average degree of change scattering and Figure 7.9b displays the average level of change focus for the sample programs. The inference of these two results is summarized in Table 7.2.

The check marks against a program denote whether that program has high/low ADOCS and ADOCF values. The sample programs with low ADOCS and high ADOCF values are *Calculator, BST, DLL, Elevator_spl, GPL_spl, and Nanoxml.* Our assumption is that these programs with low ADOCS and high ADOCF values require less testing effort to discover the faults. Whereas, if the characteristics are otherwise, i.e. the programs have high ADOCS and low ADOCF, then the testing effort is more to reveal the faults because of high scattering and tangling. The programs with high ADOCS and low ADOCF are *Expt. program, sorting, CrC, ATM, Email_spl, and Jtopas.* However, there were some programs with some different characteristics such as *Elevator, Stack* (exhibited high ADOCS and high ADOCF), and *Notepad* (with low ADOCS and low ADOCF). We show the results of our mutant fault analysis in Figure 7.9c to prove that ADOCS and ADOCF values are good representatives of the fault proneness. Figure 7.9c shows a high mutants count for the programs with high ADOCS, low ADOCF and low count for programs with low ADOCS, high ADOCF values. Even programs with high

ADOCS, and high ADOCF values had a higher count of the mutants. Thus, it can be concluded that ADOCS and ADOCF values are good representatives of mutant presence.

We counted the number of test cases selected for efficient regression testing of the same set of programs and the same set of changes to check whether ADOCS and ADOCF values are useful indicators of the required testing effort. The results in Figure 7.9d show that programs with high ADOCS, low ADOCF values require comparatively more test cases than the programs with low ADOCS, high ADOCF and low ADOCS, low ADOCF values. Even programs with high ADOCS and high ADOCF values also have a higher count of test cases. Before, we can arrive at any conclusion regarding the required testing effort, we need to check whether these test cases could reveal an acceptable number of faults. The boxplot in Figure 7.10 shows the percentage of mutation faults detected by the selected test cases. The results show that the test cases revealed 100% faults that are associated with some changes made to the sample programs. The minimum fault detection rate is 85% (for the *ATM* program). The results show that the average rate of fault detection is 94.3% for all the faults associated with the concerned changes in all the sample programs. Thus, we can conclude that ADOCS and ADOCF values are also good indicators of the testing effort that can help the testers to make correct regression testing decisions. Therefore, the results justify all our hypotheses about the metrics proposed in this chapter.

7.4 Comparison with related work

To the best of our knowledge, no work has been done for the quantification of the impact of crosscutting changes. In the absence of any work that can be directly compared with our work, we discuss some of the existing work on change impact analysis that closely relate to our work. The work that most closely relates to our work is proposed by Sun et al. [186]. The approach of CIA in [186] is based on identifying a hierarchical set of changes at different granularity levels. The impact of the change was computed using hierarchical slicing proposed in [130]. In our work, we used HD slicing for CIA that is different and has relative advantages (such as precise and correct selection of affected nodes, computational cost effectiveness, and a simple approach) over [130] (refer Figure 7.2). Unlike in [126], we aim to focus on the implication of CIA by quantifying the impact of changes that enables the testers to make better testing decisions.

The approach by Kung et al. [121] automatically categorizes the identified potential changes into different granularity levels such as data, method, class, and class library changes. Automation of the change identification is kept for our future work and is not addressed here. However, our impact analysis assesses the effect of the change at different hierarchical levels of package, class, method, and statements. Therefore, our proposed metrics consider the affected program parts at various levels of hierarchy and give the result for the program as a whole.

Rajlich [168] proposed a change propagation model targeting the software maintenance activity. This propagation model is based on graph rewriting that analyzes the dependences between the changes. A prototype tool, named *Ripples 2*, implements two basic processes of change propagation through *change-and-fix* and *top-down* propagation. The dependence analysis assumes that there are no incoming inconsistencies after a change is made. Therefore, the analysis is made only for outgoing dependences. Whereas our approach overcomes this limitation. We address both incoming and outgoing inconsistencies of a change in HD slicing that implements both forward and backward traversals for impact analysis.

Briand et al. [33] found a significant correlation between the coupling dimensions of the classes with the ripple effects of the changes in a commercial C++ system. The authors used this coupling dimension to rank the classes according to their probability of containing the ripple effects. In our work, the sample programs are based on Java. We have shown that changes can be treated as potential concerns of the testers and thus the concern metrics can be used in the context of change to quantify its effect on different program parts of the program. However, we believe an investigation on the correlation between coupling and cohesion dimensions with our proposed DOCS and DOCF measures will shed some more interesting insights to our objective. We defer this correlation analysis as our future work.

The change impact analysis by Ryder et al. [178] identifies the test cases that fail or pass due to the set of changes. In our work, we identify these set of changes and the affected tests to validate the predictiveness of our proposed metrics to reduce regression testing effort.

Tonella [192] carried out impact analysis using concept lattice of decomposition slices. The decomposition slice graph represents the dependences that exist between the computations performed on different variables. The concept lattice groups the computations that share the common variables and arranges the groups into a hierarchy of concepts. The main contribution in [192] is the graphical representation, called lattice of decomposition slices, to support software maintenance. The graph provides the relevant information regarding the computations and a data structure to conduct impact analysis. The major drawback of this approach is that it only works at intra-procedural level. Whereas our graph-based approach is proposed to work for the whole program.

Badri et al. [17] proposed a call graph based predictive change impact analysis. It generates the different control flow paths in a program that are then used to identify the components affected by the change. The reported technique supports the prediction of impact sets and regression testing. Whereas, we identify the affected program parts by performing slicing on the CRG at the point of changes. Our approach focuses on using these computed slices to calculate our metrics and predict the test effort required to confirm the quality of the program after the changes are made. The CRG used in our approach represents many other dependences (such as *type dependence*, *read/write dependence*, *generic dependence*, etc. [187]) among the program parts in addition to the control and data dependences.

Ren et al. [169] identified the causes of failure of Java programs through CIA. The reported approach used the results from a CIA tool, name Chianti, to build a compilable intermediate version of the program. This intermediate version of the program is re-executed with the tests for specific changes to locate the exact reasons for failure. Unlike [169], our proposed work on CIA is implemented on our prototype tool that works on the intermediate graphical representation of the program. We focus on using the CIA results to indicate the regression testing efforts. We plan to use CIA for debugging in near future.

The CIA proposed by Sheriff et al. [180] is based on singular value decomposition. The proposed approach is based on the collection of historical change records that may not be correct. And if these records are not available then the approach will fail to locate the effects. Our approach depends only on the current changes that are registered. These changes are correct as they are recorded by the group that makes these changes. Hattori et al. [96] have proposed to measure the precision and accuracy of the impact analysis techniques. Whereas our approach measures the scattering and tangling of the impact of changes.

German et al. [76] proposed a *change impact graph (CIG)* to visualize the impacts of change. The unaffected nodes are removed from the graph. We followed a similar graphical approach to show the effect of change as the marked nodes. The nodes unaffected by the changes are then pruned to get the affected slice graph. However, the CRG in our approach represents more dependences that exist between the program parts for an elaborate analysis propagation of error due to the changes.

The approach by Gethers et al. [77] estimates the impact set by analyzing the change request, source code, and semantic indexing. But, this approach suffers from a limitation that if the change request is inaccurate and inefficient, it may result in erroneous omission of some methods during filtration. Unlike [77], our approach is based on the computed slices that are context-sensitive and accurate. Also, our approach works not only at method levels but also for the program as a whole. Some more work on change impact analysis is available in [1].

7.4.1 Threats to validity

In this section, we highlight some of the limitations of our approach.

- Although we used a diverse set of programs for our experiments, more empirical studies with industrial applications are needed for a conclusive validation of this approach.
- We assumed that the changes are made by the same group of people those who conduct regression test for these changes. This may not always be possible in industries. Therefore, automation of the change identification is highly desirable that is not addressed in this thesis. Also, the domain of changes can be much larger than considered in Table 4.3. We plan to automate the change identification process in future.
- Graph-based approaches always suffer from scalability issues unless the granularity of the approach is compromised. We are still exploring the possibilities to address the scalability issue without sacrificing the granularity.
- The literature survey shows the use of metrics such as coupling and cohesion for change impact analysis [33, 209]. We believe a study on these metrics and the role that they may play in change impact analysis and prediction would add more insights to the proposed work. We plan to study the comparison and correlation of the object-oriented metrics with the change impacts in our future work.

7.5 Summary

There are three main contributions of this Chapter. First, we presented a dependence cluster based approach to identify and represent the dependences that exist between the changes. This dependence was represented in the form of a graph named change cluster graph (CCG). Each cluster required only one change to be retested especially when the testing time and budget have loomed. As in most of the cases, other dependent changes in that cluster are also validated and need not be retested separately. This shows that the program need not be retested for every change. This reduces the testing time. Second, we proposed metrics to quantify the effect of these changes in terms of degree of change scattering and degree of change focus. Third, we conducted experimental studies to prove the usefulness of the proposed metrics. The results show that the proposed metrics turn out to be useful indicators of fault presence and testing effort. Thus, in demanding situations, the testers can use the proposed approach to opt what to test more and what to test less and still ensure acceptable software quality.

Chapter 8

Conclusions

The primary aim of our work was to develop some efficient regression testing approaches for object-oriented software using program slicing techniques. In this chapter, we summarize our important contributions and provide some insights for future work.

8.1 Contributions

In this section, we summarize the important contributions of our work. There are four important contributions, *Regression Test Case Selection*, *Regression Test Suite Minimization*, *Regression Test Case Prioritization*, and *Identifying and Quantifying the Effect of Changes.*

8.1.1 Regression Test Case Selection

We proposed a novel regression test case selection approach by decomposing an object-oriented (OO) program into packages, classes, methods and statements that are affected by some changes made to the program. This decomposition was based on the proposed hierarchical slicing of OO programs. By mapping these decomposition to the existing test suite, we selected a new reduced change-based regression test suite to retest the modified program. We first developed a suitable intermediate representation for representing Java programs. This intermediate graph incorporated Java features like *inheritance*, *interface*, *super*, *polymorphism*, *generic classes*, etc. This intermediate representation is named *Extended Object-Oriented System Dependence Graph* (EOOSDG). This intermediate graph represents all the possible dependences among the various parts of a Java program. We have constructed EOOSDG statically only once before the execution of the program had started. Then, we applied our proposed program slicing technique on EOOSDG. We improved the scalability of the intermediate graph to a considerable extent by identifying and removing the redundant edges from the graph without affecting the computation of the slices and its application to the test case selection. This graph reduction approach helped in detecting the affected program parts in less time. The average reduction in time achieved for all the ten programs under experimentation is approximately 28.1%. The test cases that cover these affected parts of the program were then selected for regression testing. We have shown that the space complexity of our algorithm is $O(n^2)$, where n is the number of statements in the program. The time complexity of our algorithm is also $O(n^2)$. We have shown that our algorithm is computationally more efficient than the existing algorithms [130, 188]. Further, we have proved that our algorithm computes *correct* slices for any slicing criterion and correctly selects the test cases. The average reduction in the number of test cases selected for regression testing of the experimental programs is approximately 56.3%.

8.1.2 Regression Test Suite Minimization

We have proposed a novel graph based cohesion metric to measure the maintainability of different program parts in an object-oriented program and predict their fault proneness. We computed the cohesion of the sliced component as a measure to predict its correctness and preciseness. The new cohesion metric is named *affected component cohesion (ACCo)*. ACCo metric is based on the hierarchical decomposition slice of an object-oriented program that comprises of all the affected program parts. These extracted affected program parts are represented as nodes in the proposed *affected slice graph (ASG)*. The critical and sub-critical nodes that require thorough testing is determined by estimating their cohesion measure. In addition, the proposed approach of cohesion measurement is theoretically validated against the existing guidelines of cohesion measurement. The implementation of the new cohesion measurement approach gave results that were more precise and comparable with other existing approaches [40, 46, 218, 219].

Sometimes the change-based selected test suite can still appear enormous, and strict timing constraints can hinder regression testing. Hence, it is essential to minimize the test suite. We have introduced a new approach of using the proposed cohesion measure of the affected program parts to minimize the test suite for regression testing. We formulated the minimization problem in integer linear programming and obtained an optimal minimized test suite. The results of our experimental studies have shown that the minimized test suite is both effective and useful for regression testing in revealing the errors. The proposed minimization approach achieved an overall test suite minimization of 54% approximately for all the experimental programs. Also, the minimized test suite revealed approximately 91% of the faults as compared to 95% by the changed-based selected test suite, which was quite acceptable. This approach will enable the testers to decide on the magic number of test cases to choose that would still ensure acceptable quality, especially during scenarios of constrained budget and time for regression testing.

8.1.3 Regression Test Case Prioritization

We have proposed a novel graph based coupling metric to measure the error-proneness of different program parts in an object-oriented program. We computed the coupling of the sliced component as a measure to predict its correctness and preciseness. The new coupling metric is named *affected component coupling (ACC)*. ACC metric is based on the hierarchical decomposition slice of an object-oriented program that comprises of all the affected program parts. These extracted affected program parts are represented as nodes in the proposed *affected slice graph (ASG)*. The critical and sub-critical nodes that require thorough testing is determined by estimating their coupling measure. In addition, the proposed approach of coupling measurement is theoretically validated against the existing guidelines of coupling measurement. The implementation of the new coupling measurement approach gave results that were more precise and comparable with other existing approaches [30, 64, 99, 128].

We have introduced a static approach of prioritizing the test cases by computing the ACC of the affected parts of object-oriented programs. We determined the fault proneness of the nodes of ASG by computing their respective ACC values. In this technique, we assigned weights to each node of ASG based on its ACC value. The weight of a test case in a given test suite was then calculated by adding the weights of all the nodes covered by it. The test cases have been prioritized based on their coverage of fault prone affected nodes. Thus, the test case that had a higher weight was given a higher priority in the test suite. Our analysis with error seeding have shown that the test cases which executed the fault prone program parts had a higher chance to reveal faults earlier than other test cases in the test suite. The results have shown that our *fault-prone affected node coverage (FPANC)* approach achieved approximately 8% of increase in the *average percentage of fault detected (APFD)* metric value over *affected node coverage (ANC)* approach. The result obtained from seven case studies justifies that our approach is feasible and gives acceptable performance in comparison to some existing techniques [162, 163].

8.1.4 Identifying and Quantifying the Effect of Changes

Testing becomes convenient if we identify the changes that require rigorous retesting instead of focusing on all the changes. We have proposed an approach to save the effort and cost of retesting by identifying and quantifying the effect of crosscutting changes on other parts of the program. The change impact analysis has been made using an existing program slicing technique. We identified change clusters and proposed metrics to quantify their severity. There are three main contributions of this work. First, we presented a dependence cluster based approach to identify and represent the dependences that exist between the changes. This dependence was represented in the form of a graph named *Change Cluster Graph (CCG)*. A tester can retest any one change from each cluster especially when budget and time do not allow validating all the changes made to a program. In most of the cases, it is found that the other dependent changes in that cluster are also tested while testing anyone of them. The results have shown that the program need not be retested for every change. This had considerably reduced the time requirement for testing. Second, we have proposed metrics to quantify the effect of these changes. We defined terms like *degree of change scattering (DOCS)* and *degree of change focus (DOCF)* to quantify the effect of change. Third, we conducted experimental studies to prove the usefulness of the proposed metrics.

We have applied this approach to identify and quantify ten different kinds of changes made to fifteen experimental programs. The results have shown that our proposed metrics were better able to quantify these changes. These metrics have been useful indicators of the fault-proneness. The presence of mutants in the corresponding changes varied from a minimum of 12% to a maximum of 94%. The changes in five experimental programs had more than 90% of the mutants and four programs had little more than 10% mutants. An average of 47% of mutants were scattered in the affected program parts of all the sample programs. The results have shown that the test cases revealed 100% faults that were associated with some changes made to the sample programs. The minimum fault detection rate was 85% and the average rate was 94.3% for all the faults associated with the concerned changes in all the sample programs. We have compared with some of the existing approaches [33, 121, 168, 186] to confirm the effectiveness of the proposed approach.

Thus, we can conclude that our proposed metrics are also good indicators of the testing effort that can help the testers to make correct regression testing decisions. Therefore, the results have justified all our hypotheses about the metrics proposed in this chapter.

8.1.5 Implementation

We have implemented our proposed algorithms to experimentally verify their *correctness* using free and open source tools and techniques. We have tested each algorithm on some diverse input programs with several executions and slicing criteria. We have observed that the results computed for the proposed approaches are *correct* for all input experimental programs. This experimentally validated the *correctness* of our proposed algorithms. Also, the performance studies discussed in the previous sections on each contributions bring out the superiority of the performance of our algorithms compared to important related algorithms.

8.2 Future Work

In this section, we provide a brief insight to the following possible extensions to our work.

- The slicers can be used to develop efficient debuggers and test drivers for large scale object-oriented programs. We plan to explore this possibility.
- An investigation into the suitability of the proposed EOOSDG to represent dependences in other object-oriented programs (such as C#) will be studied in future.
- It would be interesting to experimentally verify the suitability of generating the intermediate graph on the fly and analyze the pros and cons of this approach.
- We aim to explore the application of other variants of slicing in regression test selection for more complex OO programs. The algorithms can be extended to compute *conditioned slices* with respect to a given condition and use them to solve regression testing problems in safety critical applications.
- We also aim to extend the proposed approaches and use slicing techniques for regression testing of both aspect-oriented and featured-oriented programs.

- We want to make an empirical study of our proposed cohesion metric along with all the existing cohesion metrics to validate its usefulness as a change-proneness indicator. The cohesion metrics can also be applied to detect the crosscutting concerns in the source codes.

- We plan to develop a multi-criteria integer linear programming model to optimize the test suite by considering many other facets of testing requirements such as coverage of low cohesive and high coupled nodes, fault-prone nodes, rate of faults detection, etc. We will investigate the effectiveness and usefulness of multi-criteria models on test suite minimization based on object-oriented metrics and their energy efficiency.

- We will incorporate different other coupling measures and metrics to predict the fault proneness of modules and prioritize the test cases based on their coverage weights.

- We want to use cohesion and coupling measures for a better fault prediction analysis and prioritization. We will extend this approach for test case prioritization of more complex object-oriented (OO) programs such as concurrent and distributed OO programs.

- Our other future activities include automation of the change identification process and investigation of the correlation between coupling and cohesion with our proposed metrics. The literature survey shows the use of metrics such as coupling and cohesion for change impact analysis [33, 209]. We believe a study on these metrics and the role that they may play in change impact analysis and prediction would add more insights to the proposed work.

- One more possible future work is on investigating the application of model checking, i.e. PAT systems [80, 185] for testing of object-oriented programs.

Bibliography

[1] Abi-Antoun, M., Wang, Y., Khalaj, E., Giang, A., and Rajlich, V. Impact Analysis Based on a Global Hierarchical Object Graph. In *Proceedings of 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2015), IEEE, pp. 221–230.

[2] Agrawal, H., Horgan, J. R., Krauser, E. W., London, S., et al. Incremental Regression Testing. In *Proceerdings of the International Conference on Software Maintenance (ICSM)* (1993), vol. 93, pp. 348–357.

[3] Agrawal, H., and Horogan, J. Dynamic Program Slicing. In *Proceeding of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices, Analysis and Verification* (1990), pp. 246–256.

[4] Al Dallal, J. An Efficient Algorithm for Computing All Program Forward Static Slices. *Transaction on Engineering, Computing and Technology 16* (2006), 108–111.

[5] Al Dallal, J. Efficient Program Slicing Algorithms for Measuring Functional Cohesion and Parallelism. *International Journal of Information Technology 4*, 2 (2007), 93–100.

[6] Al Dallal, J. Mathematical Validation of Object-Oriented Class Cohesion Metrics. *International Journal of Computers 4*, 2 (2010), 45–52.

[7] Al Dallal, J. Transitive-Based Object-Oriented Lack-of-Cohesion Metric. *Procedia Computer Science 3* (2011), 1581–1587.

[8] Al Dallal, J. Fault Prediction and the Discriminative Powers of Connectivity-Based Object-Oriented Class Cohesion Metrics. *Information and Software Technology 54*, 4 (2012), 396–416.

[9] Al Dallal, J. The Impact of Accounting for Special Methods in the Measurement of Object-Oriented Class Cohesion on Refactoring and Fault Prediction Activities. *Journal of Systems and Software 85*, 5 (2012), 1042–1057.

[10] Al Dallal, J. Object-Oriented Class Maintainability Prediction using Internal Quality Attributes. *Information and Software Technology 55*, 11 (2013), 2028–2048.

[11] Allen, M., and Horwitz, S. Slicing Java Programs that Throw and Catch Exceptions. In *Proceedings of ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM'03)* (June 2003), ACM, pp. 44–54.

[12] Alomari, H. W., Collard, M. L., Maletic, J. I., Alhindawi, N., and Meqdadi, O. srcSlice: Very Efficient and Scalable Forward Static Slicing. *Journal of Software: Evolution and Process 26*, 11 (2014), 931–961.

[13] Alpuente, M., Ballis, D., Frechina, F., and Romero, D. Using Conditional Trace Slicing for Improving Maude Programs. *Science of Computer Programming 80* (2014), 385–415.

[14] Aral, A., and Ovatman, T. Utilization of Method Graphs to Measure Cohesion in Object-Oriented Software. In *Proceedings of 37th Annual Computer Software and Applications Conference Workshops (COMPSACW)* (2013), IEEE, pp. 505–510.

[15] Arisholm, E., Briand, L. C., and Foyen, A. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering 30*, 8 (2004), 491–506.

[16] Badri, L., and Badri, M. A Proposal of a New Class Cohesion Criterion: An Empirical Study. *Journal of Object Technology 3*, 4 (2004), 145–159.

[17] Badri, L., Badri, M., and St-Yves, D. Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC'05)* (2005), IEEE, pp. 9–17.

[18] Ball, T. *The Use of Control Flow and Control Dependence in Software Tools*. PhD thesis, Computer Science Department, University of Wisconsin-Madison, 1993.

[19] Barpanda, S. S., and Mohapatra, D. P. Dynamic Slicing of Distributed Object-Oriented Programs. *IET software 5*, 5 (2011), 425–433.

[20] Beck, J., and Eichman, D. Program and Interface Slicing for Reverse Engineering. In *Proceedings of the IEEE/ACM Fifteenth International Conference on Software Enginerring (ICSE)* (1993), pp. 509–518.

[21] Bieman, J. M., and Kang, B.-K. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Software Engineering Notes* (1995), vol. 20, ACM, pp. 259–262.

[22] Bieman, J. M., and Ott, L. M. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering 20*, 8 (August 1994), 644–657.

[23] Binkley, D. Semantics Guided Regression Test Cost Reduction. *IEEE Transactions on Software Engineering 23*, 8 (1997), 498–516.

[24] Binkley, D. The Application of Program Slicing to Regression Testing. *Information and Software Technology 40*, 11 (1998), 583–594.

[25] Binkley, D. W., and Gallagher, K. B. Program Slicing. *Advances in Computers 43* (1996), 1–50.

[26] Bissyandé, T. F., Réveillère, L., Lawall, J. L., Bromberg, Y.-D., and Muller, G. Implementing an Embedded Compiler using Program Transformation Rules. *Software: Practice and Experience 45*, 2 (2015), 177–196.

[27] Biswas, S., Mall, R., Satpathy, M., and Sukumaran, S. Regression Test Selection Techniques: A Survey. *Informatica (03505596) 35*, 3 (2011).

[28] Black, J., Melachrinoudis, E., and Kaeli, D. Bi-Criteria Models for All-Uses Test Suite Reduction. In *Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE Computer Society, pp. 106–115.

[29] Briand, L., Devanbu, P., and Melo, W. An Investigation into Coupling Measures for C++. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)* (1997), ACM, pp. 412–421.

[30] Briand, L. C., Daly, J. W., and Wust, J. K. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering 25*, 1 (1999), 91–121.

[31] Briand, L. C., Morasca, S., and Basili, V. R. Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering 22*, 1 (1996), 68–86.

[32] Briand, L. C., Morasca, S., and Basili, V. R. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering 25*, 5 (1999), 722–743.

[33] Briand, L. C., Wuest, J., and Lounis, H. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. In *Proceedings of International Conference on Software Maintenance (ICSM'99)* (1999), IEEE, pp. 475–482.

[34] Briand, L. C., Wüst, J., Daly, J. W., and Victor Porter, D. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *Journal of systems and software 51*, 3 (2000), 245–273.

[35] Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H. Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)* (1999), ACM, pp. 345–354.

[36] Briand, L. C., Wust, J., and Lounis, H. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM'99)* (1999), IEEE, pp. 475–482.

[37] Brito e Abreu, F., and Goulão, M. Coupling and Cohesion as Modularization Drivers: Are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering.* (2001), IEEE, pp. 47–57.

[38] Bryant, B. R., and Vaidyanathan, V. Object-Oriented Software Specification in Programming Language Design and Implementation. In *Proceedings of 22nd Annual International Computer Software and Applications Conference (COMPSAC'98)* (1998), IEEE, pp. 387–392.

[39] Canfora, G., Cimitile, A., and Lucia, A. D. Conditioned Program Slicing. *Information and Software Technology 40* (1998), 595–607.

[40] Chae, H. S., Kwon, Y. R., and Bae, D.-H. A Cohesion measure for Object-Oriented Classes. *Software-Practice and Experience 30*, 12 (2000), 1405–1432.

[41] Chauhan, N. *Software Testing Principles and Practices.* Oxford University Press, New Delhi, India, 2010, ch. 8, pp. 255–273.

[42] Chen, J.-L., Wang, F.-J., and Chen, Y.-L. Slicing Object-Oriented Programs. In *Proceedings of Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97 / ICSC'97)* (1997), IEEE, pp. 395–404.

[43] Chen, Y.-F., Rosenblum, D. S., and Vo, K.-P. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software engineering (ICSE)* (1994), IEEE Computer Society Press, pp. 211–220.

[44] Chen, Z., and Xu, B. Slicing Object-Oriented Java Programs. *ACM SIGPLAN Notices 36*, 4 (April 2001), 33–40.

[45] Chen, Z., Xu, B., and Yang, H. Test Coverage Analysis based on Program Slicing. In *Proceedings of IRI* (2003), pp. 559–565.

[46] Chen, Z., Zhou, Y., Xu, B., Zhao, J., and Yang, H. A Novel Approach to Measuring Class Cohesion based on Dependence Analysis. In *Proceedings of International Conference in Software Maintenance (ICSM)* (2002), IEEE, pp. 377–384.

[47] Chidamber, S. R., and Kemerer, C. F. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering 20*, 6 (1994), 476–493.

[48] Ciancarini, P., Iorio, A. D., Marchetti, C., Schirinzi, M., and Vitali, F. Bridging the gap between tracking and detecting changes in XML. *Software: Practice and Experience* (2014).

[49] Clark, T., Evans, A., and France, R. Object-Oriented Theories for Model Driven Architecture. In *Advances in Object-Oriented Information Systems* (2002), Springer, pp. 235–244.

[50] Collard, M. L., Decker, M. J., and Maletic, J. I. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *Proceedings of 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2011), IEEE, pp. 173–184.

[51] Collard, M. L., Kagdi, H. H., and Maletic, J. I. An XML-Based Lightweight C++ Fact Extractor. In *Proceedings of 11th International Workshop on Program Comprehension* (2003), IEEE, pp. 134–143.

[52] Conejero, J. M., Figueiredob, E., Garciac, A., HernÃąndeza, J., and Juradoa, E. On the Relationship of Concern Metrics and Requirements Maintainability. *Information and Software Technology 54*, 2 (2012), 212–238.

[53] Conejero, J. M., Hernandez, J., Jurado, E., and Berg, K. Croscutting, what is and what is not? A Formal definition based on a Crosscutting Pattern. Technical Report TR28/07, University of Extremadura, Spain, 2007.

[54] D Lucia, A. Program Slicing: Methods and Applications. In *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation* (2001), IEEE, pp. 142–149.

[55] da Silva, B. C., Sant'Anna, C., and Chavez, C. Concern-Based Cohesion as Change Proneness Indicator: An Initial Empirical Study. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics* (2011), ACM, pp. 52–58.

[56] da Silva, B. C., Sant'Anna, C. N., and Chavez, C. v. F. An Empirical Study on How Developers Reason about Module Cohesion. In *Proceedings of the 13th International Conference on Modularity* (2014), ACM, pp. 121–132.

[57] Dandan, G., Tiantian, W., Xiaohong, S., and Peijun, M. A Test-Suite Reduction Approach to Improving Fault-Localization Effectiveness. *Computer Languages, Systems & Structures* (2013), 1–14.

[58] de AG Saraiva, J., de França, M. S., Soares, S. C., Fernando Filho, J., and de Souza, R. M. Classifying Metrics for assessing Object-Oriented Software Maintainability: A family of metrics' catalogs. *Journal of Systems and Software 103*, 1 (2015), 85–101.

[59] Do, H., Elbaum, S. G., and Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal 10*, 4 (2005), 405–435.

[60] Do, H., and Rothermel, G. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering 32*, 9 (2006), 733–752.

[61] Do, H., Rothermel, G., and Kinneer, A. Empirical Studies of Test Case Prioritization in a Junit Testing Environment. In *Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE)* (2004), IEEE, pp. 113–124.

[62] Eaddy, M., Aho, A., and Murphy, G. C. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques* (2007), IEEE Computer Society, pp. 2–7.

[63] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering 34*, 4 (2008), 497–515.

[64] Eder, J., Kappel, G., and Schrefl, M. Coupling and Cohesion in Object-Oriented Systems. Technical report, University of Klagenfurt, Austria, 1994. Citeseer.

[65] El Emam, K., Melo, W., and Machado, J. C. The Prediction of Faulty Classes using Object-Oriented Design Metrics. *Journal of Systems and Software 56*, 1 (2001), 63–75.

[66] Elbaum, S., Malishevsky, A., and Rothermel, G. Test Case Prioritization: A family of Emprical Studies. *IEEE Transactions on Software Engineering 28*, 2 (2002), 159–182.

[67] Elbaum, S., Malishevsky, A. G., and , G. Test Case Prioritization:A Family of Empirical Studies. *IEEE Transactions on Software Engineering 28*, 2 (2002), 159–182.

[68] Elbaum, S., Rothermel, G., Kanduri, S., and Malishevsky, A. G. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal 12*, 3 (2004), 185–210.

[69] Fang, C., Chen, Z., Wu, K., and Zhao, Z. Similarity-Based Test Case Prioritization using Ordered Sequences of Program Entities. *Software Quality Journal 22*, 2 (2014), 335–361.

[70] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and System 9*, 3 (1987), 319–349.

[71] Fischer, K., Raji, F., and Chruscicki, A. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference B-6-3* (1981), pp. 1–6.

[72] Forgacs, I., and Bertolino, A. Feasible Test Path Selection by Principal Slicing. In *Proceeding of 6th European Software Engineering Conference* (1997).

[73] Gallagher, K. B., and Lyle, J. R. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering 17*, 8 (1991), 751–761.

[74] Gao, K., Khoshgoftaar, T. M., Wang, H., and Seliya, N. Choosing Software Metrics for Defect Prediction: An Investigation on Feature Selection Techniques. *Software: Practice and Experience 41*, 5 (2011), 579–606.

[75] Garg, M., Lai, R., and Huang, S. J. When to Stop Testing: A Study from the Perspective of Software Reliability Models. *IET software 5*, 3 (2011), 263–273.

[76] German, D. M., Hassan, A. E., and Robles, G. Change Impact Graphs: Determining the Impact of Prior Code Changes. *Information and Software Technology 51*, 10 (2009), 1394–1408.

[77] Gethers, M., Dit, B., Kagdi, H., and Poshyvanyk, D. Integrated Impact Analysis for Managing Software Changes. In *34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 430–440.

[78] Godboley, S., Panda, S., and Mohapatra, D. P. SMCDT: A Framework for Automated MC/DC Test Case Generation using Distributed Concolic Testing. In *Distributed Computing and Internet Technology, Lecture Notes in Computer Science, Springer* (2015), vol. 8956, pp. 199–202.

[79] Green, P., Lane, P. C., Rainer, A., and Scholz, S. An Introduction to Slice-Based Cohesion and Coupling Metrics. Technical Report SE-09-488, University of Hertfordshire, 2009.

[80] Gui, L., Sun, J., Liu, Y., Si, Y. J., Dong, J. S., and Wang, X. Y. Combining Model Checking and Testing with an Application to Reliability Prediction and Distribution. In *Proceedings of the International Symposium on Software Testing and Analysis* (2013), ACM, pp. 101–111.

[81] Gupta, R., Harrold, M. J., and Soffa, M. L. Program Slicing-Based Regression Testing Techniques. *Software Testability, Verifiability and Reliability 6*, 2 (1996), 83–111.

[82] Gupta, V., and Chhabra, J. K. Package Level Cohesion Measurement in Object-Oriented Software. *Journal of the Brazilian Computer Society 18*, 3 (2012), 251–266.

[83] Hamilton, J., and Danicic, S. Dependence Communities in Source Code. In *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM)* (2012), IEEE, pp. 579–582.

[84] Hamlet, D. Foundations of Software Testing: Dependability Theory. *ACM SIGSOFT Software Engineering Notes 19*, 5 (1994), 128–139.

[85] Hammer, C., and Snelting, G. An Improved Slicer for Java. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)* (2004), ACM, 5th ACM SIGPLAN-SIGSOFT, pp. 17–22.

[86] Harman, M. Conditioned Slicing Supports Partition Testing. *Journal of Software Testing, Verification and Reliability 12* (2002), 23–28.

[87] Harman, M., Binkley, D., and Danicic, S. Amorphous Program Slicing. *The Journal of Systems and Software 68* (2003), 45–64.

[88] Harman, M., and Danicic, S. Using Program Slicing to Simplify Testing. *Software Testing, Verification and Reliability 5*, 3 (1995), 143–162.

[89] Harman, M., Okulawon, M., Sivagurunathan, B., and Danicic, S. Slice-Based Measurement of Function Coupling. In *Proceedings of IEEE/ACM ICSE workshop on Process Modelling and Emprical Studies of Software Evolution (PMESSE'97)* (1997), IEEE Press, pp. 26–32.

[90] Harrold, M. J., and et al. Regression Test Selection for Java Software. In *Proceeding of the ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA'01)* (2001), pp. 312–326.

[91] Harrold, M. J., Gupta, R., and Soffa, M. L. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology (TOSEM) 2*, 3 (1993), 270–285.

[92] Harrold, M. J., Larsen, L., Lloyd, J., Nedved, D., Page, M., Rothermel, G., Singh, M., and Smith, M. Aristotle: A System for Development of Program Analysis Based Tools. In *Proceedings of the 33rd annual on Southeast regional conference* (1995), ACM, pp. 110–119.

[93] Harrold, M. J., and Rothermel, G. A Coherent Family of Analyzable Graphical Representations for Object-Oriented Software. *Department of Computer and Information Science, The Ohio State University, Technical Report OSU-CISRC-11/96-TR60* (1996).

[94] Harrold, M. J., and Souffa, M. An Incremental Approach to Unit Testing during Maintenance. In *Proceedings of the International Conference on Software Maintenance* (1988), IEEE, pp. 362–367.

[95] Hartmann, J., and Robson, D. Revalidation during the Software Maintenance Phase. In *Proceedings of International Conference on Software Maintenance (ICSM)* (1989), IEEE, pp. 70–80.

[96] Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J., and Damásio, J. On the Precision and Accuracy of Impact Analysis Techniques. In *Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS)* (2008), IEEE, pp. 513–518.

[97] Henry, S., and Kafura, D. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 5 (1981), 510–518.

[98] Herzig, K., Greiler, M., Czerwonka, J., and Murphy, B. The Art of Testing Less without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 483–493.

[99] Hitz, M., and Montazeri, B. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing* (1995), vol. 50, pp. 75–76.

[100] Horowitz, S., and Reps, T. The use of Program Dependence Graphs in Software Engineering. In *Fourteenth International Conference on Software Engineering, Melbourne* (1992), pp. 392–411.

[101] Horowitz, S., Reps, T., and Binkley, D. Interprocedural Slicing using Dependence Graphs. *ACM SIGPLAN Notices 23*, 7 (1988), 35–46.

[102] Horowitz, S., Reps, T., and Binkley, D. Interprocedural Slicing using Dependence Graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–60.

[103] Hou, S.-S., Zhang, L., Xie, T., Mei, H., and Sun, J.-S. Applying Interface-Contract Mutation in Regression Testing of Component-Based Software. In *Proceedings of International Conference on Software Maintenance (ICSM)* (2007), IEEE, pp. 174–183.

[104] Jeffrey, D., and Gupta, N. Test Case Prioritization using Relevant Slices. In *Proceedings of 30th Annual International Computer Software and Applications Conference* (2006), pp. 411–420.

[105] Jeffrey, D., and Gupta, N. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *IEEE Transactions on Software Engineering 33*, 2 (2007), 108–123.

[106] Kamkar, M. An Overview and Comparative Classification of Program Slicing Techniques. *Journal of Systems and Software 31*, 3 (1995), 197–214.

[107] Kaner, C., Falk, J., and Nguyen, H. Q. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

[108] Kayes, M. I. Test Case Prioritization for Regression Testing Based on Fault Dependency. In *Proceedings of 3rd International Conference on Electronics Computer Technology (ICECT)* (2011), vol. 5, IEEE, pp. 48–52.

[109] Khan, K., Lo, B., Skramstad, T., and Skramstad, T. Tasks and Methods for Software Maintenance: A Process-Oriented Framework. *Australasian Journal of Information Systems 9*, 1 (2007).

[110] Kim, S., Clark, J., and McDermid, J. Assessing Test Set Adequacy for Object-Oriented Programs using Class Mutation. In *28 JAIIO: Symposium on Software Technology* (1999).

[111] Kim, S., Clark, J. A., and McDermid, J. A. Class Mutation: Mutation Testing for Object-Oriented Programs. In *Proc. Net. ObjectDays* (2000), Citeseer, pp. 9–12.

[112] Klemola, T. A Cognitive Model for Complexity Metrics. In *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (2000), pp. 12–16.

[113] Korel, B., Koutsogiannakis, G., and Tahat, L. Application of System Models in Regression Test Suite Prioritization. In *Proceedings of the IEEE International Conference on Software Maintenance* (2008), pp. 247–256.

[114] Korel, B., Koutsogiannakis, G., and Tahat, L. H. Model-Based Test Prioritization Heuristic Methods and their Evaluation. In *Proceedings of the 3rd international workshop on Advances in model-based testing* (2007), ACM, pp. 34–43.

[115] Korel, B., Tahat, L. H., and Harman, M. Test Prioritization using System Models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, (ICSM'05)* (2005), IEEE, pp. 559–568.

[116] Kovács, G., Magyar, F., and Gyimóthy, T. Static slicing of java programs. In *University* (1996), Citeseer.

[117] Kozlov, D., Koskinen, J., and Sakkinen, M. Fault-Proneness of Open Source Software: Exploring its Relations to Internal Software Quality and Maintenance Process. *Open Software Engineering Journal 7* (2013), 1–23.

[118] Krinke, J. Statement-Level Cohesion Metrics and their Visualization. In *Proceedings of Seventh International Working Conference Source Code Analysis and Manipulation (SCAM)* (2007), IEEE, pp. 37–48.

[119] Krishnaswamy, A. Program Slicing: An Application of Object-Oriented Program Dependence Graphs. Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[120] Kumar, R., Panda, S., and Mohapatra, D. P. Analysis of Java Programs using Joana and Java SDG API. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (2015), IEEE, pp. 2402–2408.

[121] Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C. Change Impact Identification in Object-Oriented Software Maintenance. In *Proceedings of the International Conference on Software Maintenance* (1994), IEEE, pp. 202–211.

[122] Lanubile, F., and Visaggio, G. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering 23* (1997), 246–259.

[123] Larsen, L., and Harrold, M. J. Slicing Object-Oriented Software. In *Proceedings of the 18th IEEE International Conference on Software Engineering* (1996), pp. 495–505.

[124] Larsen, L., and Harrold, M. J. Slicing Object-Oriented Software. In *Proceedings of the 18th International Conference on Software Engineering* (1996), IEEE, pp. 495–505.

[125] Latha, T. J., and Suganthi, L. An Empirical Study on creating Software Product Value in India - An Analytic Hierarchy Process Approach. *International Journal of Business Information Systems 18*, 1 (2015), 26–43.

[126] Lehnert, S. A Taxonomy for Software Change Impact Analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution* (2011), ACM, pp. 41–50.

[127] Leung, H., and White, L. Insights into Regression Testing Selection. In *Proceedings of the Conference on Software Maintenance* (1989), pp. 60–69.

[128] Li, B. *A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement.* Citeseer, 2001.

[129] Li, B., Sun, X., Leung, H., and Zhang, S. A Survey of Code-Based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability 23*, 8 (2013), 613–646.

[130] Li, B. X., Fan, X. C., Pang, J., and Zhao, J. J. Model for Slicing Java Programs Hierarchically. *Journal of Computer Science and Technology 19*, 6 (2004), 848–858.

[131] Li, D., Jin, Y., Sahin, C., Clause, J., and Halfond, W. G. Integrated Energy-Directed Test Suite Optimization. In *Proceedings of International Symposium on Software Testing and Analysis* (2014), ACM, pp. 339–350.

[132] Li, Z., Harman, M., and Hierons, R. M. Search Algorithms for Regression Test Case Prioritization. *Software Engineering, IEEE Transactions on 33*, 4 (2007), 225–237.

[133] Liang, D., and Larson, L. Slicing Objects using System Dependence Graphs. In *Proceedings of International Conference on Software Maintenance (ICSM)* (November 1998), pp. 358–367.

[134] Lin, I.-W., Huang, C.-Y., and Lin, C.-T. Test Suite Reduction Analysis with Enhanced Tie-Breaking Techniques. In *Proceedings of 4th IEEE International Conference on Management of Innovation and Technology (ICMIT)* (2008), IEEE, pp. 1228–1233.

[135] Lippert, M., and Lopes, V. C. A Study on Exception Detection and Handling using Aspect-Oriented Programming. In *In Proceedings of the International Conference on Software Engineering (ICSE)* (2000), ACM Press, p. 418âĂŞ427.

[136] Lisper, B., Masud, A. N., and Khanfar, H. Static Backward Demand-Driven Slicing. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation* (2015), ACM, pp. 115–126.

[137] Lossing, N., Guillou, P., Amini, M., and Irigoin, F. From Data to Effects Dependence Graphs: Source-to-Source Transformations for C. In *the 18th International Workshop on Compilers for Parallel Computing (CPC'15).*

[138] Lou, Y., Hao, D., and Zhang, L. Mutation-Based Test-Case Prioritization in Software Evolution. In *proceedings of 26th International Symposium on Software Reliability Engineering (ISSRE)* (2015), IEEE, pp. 46–57.

[139] Lyle, J. R., and Weiser, M. D. Automatic Program Bug Location by Program Slicing. In *Proceedings of the second International Conference on Computers and Applications, Peking, China* (1987), pp. 877–882.

[140] Ma, Y.-S., Offutt, J., and Kwon, Y. R. MuJava: An Automated Class Mutation System. *Software Testing, Verification and Reliability 15*, 2 (2005), 97–133.

[141] Majumdar, D., Kanjilal, A., and Bhattacharya, S. Separation of Scattered Concerns: A Graph Based Approach for Aspect Mining. *ACM SIGSOFT Software Engineering Notes 36*, 2 (2011), 1–11.

[142] Malishevsky, A., Ruthruff, J., Rothermel, G., and Elbaum, S. Cost-cognizant Test Case Prioritization. Technical Report TRUNL-CSE-2006-0004, 2006.

[143] Mall, R. *Fundamentals of Software Engineering*, 3rd ed. PHI Learning Pvt. Ltd., 2010, pp. 159–160.

[144] Manna, Z., and R, W. The Logic of Complete Programming. *IEEE Transactions on Software Engineering 4*, 1 (1978), 199–229.

[145] Mansour, N., and El-Fakih, K. Natural Optimization Algorithms for Optimal Regression Testing. In *Proceedings of Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)* (1997), IEEE, pp. 511–514.

[146] Mansour, N., and El-Fakih, K. Simulated Annealing and Genetic Algorithms for Optimal Regression Testing. *Journal of Software Maintenance 11*, 1 (1999), 19–34.

[147] Marinescu, R. Measurement and Quality in Object-Oriented Design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), IEEE, pp. 701–704.

[148] Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., and Rothermel, G. A Static Approach to Prioritizing Junit Test Cases. *IEEE Transactions on Software Engineering 38*, 6 (2012), 1258–1275.

[149] Meyers, T. M., and Binkley, D. Slice-Based Cohesion Metrics and Software Intervention. In *Proceedings of 11th Working Conference on Reverse Engineering* (2004), IEEE, pp. 256–265.

[150] Meyers, T. M., and Binkley, D. An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM) 17*, 1 (2007), 2.

[151] Meyers, T. M., and Binkley, D. An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM) 17*, 1 (2007), 2.

[152] Mohapatra, D. P., Mall, R., and Kumar, R. An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems* (2004), IEEE Computer Society, pp. 60–65.

[153] Mohapatra, D. P., Mall, R., and Kumar, R. Computing Dynamic Slices of Concurrent Object-Oriented Programs. *Information and software technology 47*, 12 (2005), 805–817.

[154] Mohapatra, D. P., Mall, R., and Kumar, R. An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica (Slovenia) 30*, 2 (2006), 253–277.

[155] Morell, L. J. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering 16*, 8 (1990), 844–857.

[156] Murgia, A., Tonelli, R., Marchesi, M., Concas, G., Counsell, S., and Swift, S. System Performance Analyses through Object-Oriented Fault and Coupling Prisms. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering* (2014), ACM, pp. 233–238.

[157] Najumudheen, E., Mall, R., and Samanta, D. A Dependence Graph-Based Representation for Test Coverage Analysis of Object-Oriented Programs. *ACM SIGSOFT Software Engineering Notes 34*, 2 (2009), 1–8.

[158] Ngo, M. N., and Tan, H. B. K. Heuristics-Based Infeasible Path Detection for Dynamic Test Data Generation. *Information and Software Technology 50*, 7 (2008), 641–655.

[159] Offutt, J., Alexander, R., Wu, Y., Xiao, Q., and Hutchinson, C. A Fault Model for Subtype Inheritance and Polymorphism. In *Proceedings of 12th International Symposium on Software Reliability Engineering, ISSRE.* (2001), IEEE, pp. 84–93.

[160] Offutt, J., Ma, Y.-S., and Kwon, Y.-R. The Class-Level Mutants of MuJava. In *Proceedings of the international workshop on Automation of software test* (2006), ACM, pp. 78–84.

[161] Ottenstein, J. K., and Ottenstein, M. L. The Program Depenedence Graph in a Software Development Environment. *ACM SIGPLAN Notices 19*, 5 (1984), 177–184.

[162] Panigrahi, C. R., and Mall, R. An Approach to Prioritize the Regression Test Cases of Object-Oriented Programs. *CSI Transactions on ICT 1*, 2 (2013), 159–173.

[163] Panigrahi, C. R., and Mall, R. A Heuristic-Based Regression Test Case Prioritization Approach for Object-Oriented Programs. *Innovations in Systems and Software Engineering 10*, 3 (2014), 155–163.

[164] Panigrahi, C. R., and Mall, R. Regression test size reduction using improved precision slices. *Innovations in Systems and Software Engineering* (2015), 1–7.

[165] Park, H., Ryu, H., and Baik, J. Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing. In *Proceedings of Second International Conference on Secure System Integration and Reliability Improvement (SSIRI'08)* (2008), IEEE, pp. 39–46.

[166] Qi, X., and Xu, B. Dependence Analysis of Concurrent Programs Based on Rechability Graph and it's Applications. In *Proceedings of International Conference on Computational Science* (2004), pp. 405–408.

[167] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., and Binkley, D. Recovering Test-to-Code Traceability using Slicing and Textual Analysis. *Journal of Systems and Software 88*, 1 (2014), 147–168.

[168] Rajlich, V. A Model for Change Propagation Based on Graph Rewriting. In *Proceedings of the International Conference on Software Maintenance* (1997), IEEE, pp. 84–91.

[169] Ren, X., Chesley, O. C., and Ryder, B. G. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. *IEEE Transactions on Software Engineering 32*, 9 (2006), 718–732.

[170] Reps, T., and Rosay, G. Precise Interprocedural Chopping. In *Proceedings of Third ACM Symposium on the Foundations of Software Engineering* (October 1995), pp. 41–52.

[171] Rilling, J., and Klemola, T. Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics. In *11th IEEE International Workshop on Program Comprehension* (2003), IEEE, pp. 115–124.

[172] Rothermel, G., and Harrold, M. J. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering 22*, 8 (1996), 529–551.

[173] Rothermel, G., and Harrold, M. J. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology (TOSEM) 6*, 2 (1997), 173–210.

[174] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proceedings of the International Conference on Software Maintenance(ICSM)* (1998), IEEE, pp. 34–43.

[175] Rothermel, G., Untch, R., Chu, C., and Harrold, M. Prioritizing Test Cases for Regression Testing. *IEEE Transactions on Software Engineering 27*, 10 (2001), 924–948.

[176] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. Test Case Prioritization: An Empirical Study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on* (1999), IEEE, pp. 179–188.

[177] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. Prioritizing Test Cases for Regression Testing. *Software Engineering, IEEE Transactions on 27*, 10 (2001), 929–948.

[178] Ryder, B. G., and Tip, F. Change Impact Analysis for Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2001), ACM, pp. 46–53.

[179] Sengupta, S., and Bhattacharya, S. Functional Specifications of Object-Oriented Systems: A Model Driven Framework. In *31st Annual International Computer Software and Applications Conference (COMPSAC)* (2007), vol. 1, IEEE, pp. 667–672.

[180] Sherriff, M., and Williams, L. Empirical Software Change Impact Analysis using Singular Value Decomposition. In *1st International Conference on Software Testing, Verification, and Validation* (2008), IEEE, pp. 268–277.

[181] Shu, G., Sun, B., Henderson, T., Podgurski, A., et al. JavaPDG: A New Platform for Program Dependence Analysis. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)* (2013), IEEE, pp. 408–415.

[182] Silva, J. A Vocabulary of Program Slicing-Based Techniques. *ACM Computing Surveys (CSUR) 44*, 3 (2012), 1–41.

[183] Smith, B. H., and Williams, L. An Empirical Evaluation of the MuJava Mutation Operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007* (2007), IEEE, pp. 193–202.

[184] Srikanth, H., Banerjee, S., Williams, L., and Osborne, J. Towards the Prioritization of System Test Cases. *Software Testing, Verification and Reliability 24*, 4 (2014), 320–337.

[185] Sun, J., Liu, Y., Dong, J. S., and Pang, J. PAT: Towards Flexible Verification under Fairness. In *Computer Aided Verification* (2009), Springer, pp. 709–714.

[186] Sun, X., Li, B., Zhang, S., and Tao, C. HSM-Based Change Impact Analysis of Object-Oriented Java Programs. *Chinese of Journal Electronics 20*, 2 (2011), 247–251.

[187] Tahir, A., and MacDonell, S. G. A Systematic Mapping Study on Dynamic Metrics and Software Quality. In *28th IEEE International Conference on Software Maintenance (ICSM)* (2012), IEEE, pp. 326–335.

[188] Tao, C., Li, B., Sun, X., and Zhang, C. An Approach to Regression Test Selection Based on Hierarchical Slicing Technique. In *34th Annual IEEE Computer Software and Applications Conference Workshops* (2010), pp. 347–352.

[189] Tian, J., and Zelkowitz, M. V. A Formal Program Complexity Model and its Application. *Journal of Systems and Software 17*, 3 (1992), 253–266.

[190] Tip, F. A Survey of Program Slicing Techniques. *Journal of programming languages 3*, 3 (1995), 121–189.

[191] Tip, F. Infeasible Paths in Object-Oriented Programs. *Science of Computer Programming 97* (2015), 91–97.

[192] Tonella, P. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Transactions on Software Engineering 29*, 6 (2003), 495–509.

[193] Tonella, P., Antoniol, G., Fiutem, R., and Merlo, E. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *Proceedings of 19th International Conference on Software Engineering* (May 1997), pp. 433–443.

[194] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. SOOT-A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (1999), IBM Press, pp. 125–135.

[195] Venkatesh, G. A. The Semantic Approach to Program Slicing. *ACM SIGPLAN Notices 26*, 6 (1991), 107–119.

[196] Vincenzi, A., Wong, W., Delamaro, M., and Maldonado, J. JaBUTi:A Coverage Analysis Tool for Java Programs. *XVII SBES–Simpósio Brasileiro de Engenharia de Software* (2003), 79–84.

[197] Wagstaff, K., Cardie, C., Rogers, S., and Schrödl, S. Constrained K-means Clustering with Background Knowledge. In *International Conference on Machine Learning (ICML)* (2001), pp. 577–584.

[198] Walkinshaw, N., Roper, M., and Wood, M. The Java System Dependence Graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation* (2003), IEEE, pp. 55–64.

[199] Wang, D., Dong, M., and Zhan, W. An Input Data Related Behavior Extracting and Measuring Model. *International Journal of Applied Mathematics and Information Sciences 7*, 2 (2013), 683–689.

[200] Wang, T., and Roychoudhury, A. Using Compressed Bytecode Traces for Slicing Java Programs. In *26th International Conference on Software Engineering (ICSE'04)* (2004), ACM, pp. 512–521.

[201] Weiser, M. Program Slicing. In *Proceedings of the 5th International Conference on Software* (1981), San Diego, California, USA, pp. 439–449.

[202] Weiser, M. Programmers use Slices when Debugging. *Communications of the ACM 25*, 7 (1982), 446–452.

[203] Wen, W. Software Fault Localization based on Program Slicing Spectrum. In *In the Proceedings of the 2012 International Conference on Software Engineering* (2012), ACM, pp. 1511–1514.

[204] Weyuker, E. J. Evaluating Software Complexity Measures. *Software Engineering, IEEE Transactions on 14*, 9 (1988), 1357–1365.

[205] White, L. J., Narayanswamy, V., Friedman, T., Kirschenbaum, M., Piwowarski, P., and Oha, M. Test Manager: A Regression Testing Tool. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on* (1993), IEEE, pp. 338–347.

[206] Wibowo, B., and Sajeev, A. S. M. A Tool for Regression Testing. In *IASTED Conf. on Software Engineering* (2004), pp. 315–320.

[207] Wong, W. E., Horgan, J. R., London, S., and Agrawal, H. A Study of Effective Regression Testing in Practice. In *Proceedings of Eighth International Symposium on Software Reliability Engineering* (1997), IEEE, pp. 264–274.

[208] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P. Effect of Test Set Minimization on Fault Detection Effectiveness. In *17th International Conference on Software Engineering (ICSE).* (1995), IEEE, pp. 41–41.

[209] Yang, Y., Zhou, Y., Lu, H., Chen, L., Chen, Z., Xu, B., Leung, H., and Zhang, Z. Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study. *IEEE Transactions on Software Engineering 41*, 4 (2015), 331–357.

[210] Yoo, S., and Harman, M. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability 22*, 2 (2012), 67–120.

[211] Yoshida, N., Kinoshita, M., and Iida, H. A Cohesion Metric Approach to Dividing Source Code into Functional Segments to Improve Maintainability. In *Software Maintenance and Reengineering (CSMR), 16th European Conference on* (2012), IEEE, pp. 365–370.

[212] Young, M., and Taylor, R. N. Rethinking the Taxonomy of Fault Detection Techniques. In *Proceedings of the 11th International Conference on Software Engineering (ICSE)* (1989), ACM, pp. 53–62.

[213] Yu, P., Systa, T., and Muller, H. Predicting Fault-Proneness using OO Metrics. An Industrial Case Study. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on* (2002), IEEE, pp. 99–107.

[214] Zhang, L., Hao, D., Zhang, L., Rothermel, G., and Mei, H. Bridging the Gap between the Total and Additional Test-Case Prioritization Strategies. In *Proceedings of International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 192–201.

[215] Zhao, J. Applying Program Dependence Analysis to Java Software. In *Proceedings of Workshop on Software Engineering and Database Systems, International Computer Symposium* (1998), Citeseer, pp. 162–169.

[216] Zhao, J. Dynamic Slicing of Object-Oriented Programs. Technical Report SE-98-119, Information Processing Society of Japan, 1998.

[217] Zhao, J., Cheng, J., and Ushijima, K. A Dependence Based Representation for Concurrent Object-Oriented Software Maintenance. In *Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering* (1998), pp. 60–66.

[218] Zhou, Y., Wen, L., Wang, J., Chen, Y., Lu, H., and Xu, B. DRC: A Dependence Relationships Based Cohesion Measure for Classes. In *Proceedings of Tenth Asia-Pacific Software Engineering Conference (APSEC)* (2003), IEEE, pp. 215–223.

[219] Zhou, Y., Xu, B., Zhao, J., and Yang, H. ICBMC: An Improved Cohesion Measure for Classes. In *Proceedings. International Conference Software Maintenance* (2002), IEEE, pp. 44–53.

[220] Zhu, H., Hall, P. A., and May, J. H. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR) 29*, 4 (1997), 366–427.

Dissemination of Work

1. S. Panda, D. Munjal, D. P. Mohapatra, *A Slice-Based Change Impact Analysis for Regression Test Case Prioritization of Object-Oriented Programs*, Journal of Advances in Software Engineering, Hindawi Publishers, 2015 (Accepted).

2. S. Panda, D. P. Mohapatra, *ACCo: A Novel Approach to Measure Cohesion using Hierarchical Slicing of Java Programs*, Journal of Innovations in Systems and Software Engineering, Springer, Vol. 11, No. 4, pp: 243-260, 2015.

3. S. Panda, D. P. Mohapatra, *Hierarchical Regression Test Selection using Slicing*, International Journal of Computational Science and Engineering, Inderscience Publishers, 2015 (in Press).

4. S. Panda, D. P. Mohapatra, *A Framework to measure Coupling using Static Change Impact Analysis*, International Journal of Business Information Systems, Inderscience Publishers, 2015 (in Press).

5. S. Panda, D. P. Mohapatra, *Application of Hierarchical Slicing to Regression Test Selection of Java Programs*, In Journal of Model-based Software Engineering: Some Perspectives, Infosys Labs Briefings, 6th India Software Engineering Conference Workshop, Vol. 11, No. 2, pp: 3-19, 2013.

Biodata

Subhrakanta Panda

Department of Computer Science and Engineering,
National Institute of Technology Rourkela,
Rourkela – 769 008, Odisha, India.
`Mob: +91 94385 48432`
`Email: 511cs109@nitrkl.ac.in, subhrakanta11@gmail.com`

Permanent Address

Plot No. 934, Jyotivihar, Bidanasi, Cuttack 753 014, Odisha, India.

Qualification

- PhD (CSE) (*Continuing*)
  National Institute of Technology Rourkela
- M.Tech. (CSE)
  DRIEMS under BPUT, Rourkela, completed with 9.01 CGPA.
- B.Tech (CSE)
  Kalinga Institute of Technology and Science (KITS), KIIT, Bhubaneswar, completed with 67% of marks

Publications

- Journals:
  Published/Accepted: 4
  Communicated: 2
- Conferences: 9

Area of Interest

Programming Slicing, Change Impact Analysis, Software Testing, Regression Testing, Software Metrics, Aspect Mining, Semantic Analysis, Graph Theory Applications, Big Data Analytics.

Hobbies

Reading Books, Playing Chess, Solving Sudoku Puzzles.

Index

ACC, 126
ACCo, 101
all-edges criterion, 22
all-nodes criterion, 22
APFD, 121
ASG, 99

Backward Slice, 24
Black Box Testing, 15

CCG, 157
CFG, 29, 31
Change Impact Analysis, 154
CIG, 156
Clustering, 136
Cohesion, 100
Coupling, 123
CRG, 154
Crosscutting, 159

DOCF, 161
DOCS, 160
Dynamic Slice, 26

EOOSDG, 64

Forward Slice, 24

Grey Box, 15

HD Slicing, 73
Hierarchical Slicing, 62

Integer Linear Programming, 107

OPDG, 35

PDG, 32
Program Slicing, 23

Regression Testing, 18

Scattering, 159
SDG, 33
Selective Regression Testing, 19
slicing criterion, 62
Software Changes, 152
Software Testing, 13
Static Slice, 25

Tangling, 159
Test Case, 16
Test Case Prioritization, 20
Test Case Weight, 137
Test Suite, 16
Test Suite Minimization, 19

White Box Testing, 15