Prioritizing program elements: A pre-testing effort to improve software quality

Mitrabinda Ray

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

Prioritizing program elements: A pre-testing effort to improve software quality

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

Doctor of Philosophy

in

Computer Science and Engineering

by

Mitrabinda Ray
(Roll: 508CS801)

under the guidance of

Prof. Durga Prasad Mohapatra
NIT Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India
January 2012

Department of Computer Science and Engineering
National Institute of Technology Rourkela, Rourkela.
Rourkela-769 008, Orissa, India.

December 17, 2012

Certificate

This is to certify that the work in the thesis entitled ***Prioritizing program elements: A pre-testing effort to improve software quality*** by ***Mitrabinda Ray*** is a record of an original research work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Doctor of Philosophy in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Dr. Durga Prasad Mohapatra
CSE department of NIT Rourkela, Rourkela.

Acknowledgment

"Life is God's novel. Let him write it."
Thank you God for all the blessings you have given me in every way...

I owe deep gratitude to the ones who have contributed greatly in completion of this thesis.

Foremost, I would like to express my sincere gratitude to my advisor, **Prof. Durga Prasad Mohapata** for providing me with a platform to work on challenging areas of softwar testing and slicing. His profound insights and attention to details have been true inspirations to my research.
My thanks goes to Prof. S. K. Rath for painstakingly reading my report and helping me with his insightful comments on my work. I am grateful to Prof. S. K. Jena and Prof. B. Majhi for giving me valuable suggestions toward enhancing the quality of the work in shaping this thesis.
I am grateful to all the faculty members of the CSE Department for their many helpful comments and constant encouragement. I wish to thank the Software Laboratory staff and all the secretarial staff of the CSE Department for their sympathetic cooperation. I would like to thank my lab-mates Swati Vipsita, Suresh, Jayadeep and Madhumita for their encouragement and understanding. Their help can never be penned with words.
Most importantly, none of this would have been possible without the love and patience of my family. Without the constant support and encouragement of my husband, Jyoti Prakash, I could hardly have completed this work. His unending patience, encouragement and understanding have made it all possible, and meaningful. I wish to appreciate and thank my daughter, **Gudia**, for bearing me to stay away from home for the most of the time.

My mother to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to her.

Mitrabinda Ray

Abstract

Test effort prioritization is a powerful technique that enables the tester to effectively utilize the test resources by streamlining the test effort. The distribution of test effort is important to test organization. We address prioritization-based testing strategies in order to do the best possible job with limited test resources. Our proposed techniques give benefit to the tester, when applied in the case of looming deadlines and limited resources. Some parts of a system are more critical and sensitive to bugs than others, and thus should be tested thoroughly. The rationale behind this thesis is to estimate the criticality of various parts within a system and prioritize the parts for testing according to their estimated criticality. We propose several prioritization techniques at different phases of Software Development Life Cycle (SDLC). Different chapters of the thesis aim at setting test priority based on various factors of the system. The purpose is to identify and focus on the critical and strategic areas and detect the important defects as early as possible, before the product release. Focusing on the critical and strategic areas helps to improve the reliability of the system within the available resources.

We present code-based and architecture-based techniques to prioritize the testing tasks. In these techniques, we analyze the criticality of a component within a system using a combination of its internal and external factors. We have conducted a set of experiments on the case studies and observed that the proposed techniques are efficient and address the challenge of prioritization.

We propose a novel idea of calculating the influence of a component, where influence refers to the contribution or usage of the component at every execution step. This influence value serves as a metric in test effort prioritization. We first calculate the influence through static analysis of the source code and then, refine our work by calculating it through dynamic analysis. We have experimentally proved that decreasing the reliability of an element with high influence value drastically increases the failure rate of the system, which is not true in case of an element with low influence value. We estimate the criticality of a component within a system by considering its both internal and external factors such as influence value, average execution time, structural complexity, severity and business value. We prioritize the components for testing according to their estimated criticality. We have compared our approach with a related approach, in which the components were prioritized on the basis of

their structural complexity only. From the experimental results, we observed that our approach helps to reduce the failure rate at the operational environment. The consequence of the observed failures were also low compared to the related approach. Priority should be established by order of importance or urgency. As the importance of a component may vary at different points of the testing phase, we propose a multi cycle-based test effort prioritization approach, in which we assign different priorities to the same component at different test cycles.

Test effort prioritization at the initial phase of SDLC has a greater impact than that made at a later phase. As the analysis and design stage is critical compared to other stages, detecting and correcting errors at this stage is less costly compared to later stages of SDLC. Designing metrics at this stage help the test manager in decision making for allocating resources. We propose a technique to estimate the criticality of a use case at the design level. The criticality is computed on the basis of complexity and business value. We evaluated the complexity of a use case analytically through a set of data collected at the design level. We experimentally observed that assigning test effort to various use cases according to their estimated criticality improves the reliability of a system under test.

Test effort prioritization based on risk is a powerful technique for streamlining the test effort. The tester can exploit the relationship between risk and testing effort. We proposed a technique to estimate the risk associated with various states at the component level and risk associated with use case scenarios at the system level. The estimated risks are used for enhancing the resource allocation decision. An intermediate graph called *Inter-Component State-Dependence graph (ISDG)* is introduced for getting the complexity for a state of a component, which is used for risk estimation. We empirically evaluated the estimated risks. We assigned test priority to the components / scenarios within a system according to their estimated risks. We performed an experimental comparative analysis and observed that the testing team guided by our technique achieved high test efficiency compared to a related approach.

Contents

List of Figures **x**

List of Tables **xii**

Abbreviations **xiv**

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Objective . . . 5

1.3 Overview . . . 6

1.4 Focus and Contribution of the Thesis . . . 7

1.5 Organization of the Thesis . . . 10

2 Background **12**

2.1 Object-Oriented Technology and Software Testing . . . 12

2.2 McCabes Cyclomatic Complexity . . . 13

2.3 Halstead Complexity Metric . . . 14

2.4 Program Slice . . . 15

2.4.1 Categories of program slicing . . . 15

2.4.2 Applications of program slicing . . . 17

2.5 Program Representation . . . 18

2.5.1 Program Dependence Graph (PDG) . . . 18

2.5.2 System Dependence Graph (SDG) . . . 18

2.5.3 Extended System Dependence Graph (ESDG) . . . 19

2.6 Unified Modeling Language (UML) . . . 20

2.7 CK Metrics . . . 21

2.8 Value-based Testing . . . 22

2.9 Operational Profile . . . 24

2.10 Risk-based Testing . . . 25

2.11 Summary . . . 25

3 Related Work **27**

3.1 Pre-testing Effort . . . 27

3.1.1 Code prioritization . . . 28

3.1.2 Fault-prone based testing . . . 29

3.2 On-testing Effort . . . 30

3.3 Empirical Work on Reliability Analysis . . . 33

3.4 Early Test Effort Estimation Methods based on Use Cases . . . 36

3.5 Risk Analysis for Testing . . . 37

3.6 Summary . . . 38

4 Prioritizing Source Code for Testing **39**

4.1 Our Approach . . . 41

4.1.1 Influence of a method . . . 42

4.1.2 Influence of a class . . . 44

4.1.3 Average execution time of a class within a system . . . 44

4.1.4 Computation of criticality . . . 45

4.2 Experimental Studies . . . 45

4.2.1 Sensitivity analysis . . . 46

4.2.2 Comparison with Musa's approach . . . 48

4.2.3 Threats to validity of results . . . 50

4.2.4 Limitation of our approach . . . 51

4.3 Summary . . . 52

5 Criticality Estimation **53**

5.1 Our Approach . . . 55

5.1.1 Analyzing the structural complexity . . . 56

5.1.2 Severity analysis . . . 58

5.1.3 Business value estimation . . . 64

5.1.4 Criticality computation . . . 68

5.2 Experimental Studies . . . 68

5.2.1 Result analysis . . . 73

5.3 Summary . . . 74

6 Multi Cycle-based Test Effort Prioritization Approach **76**

6.1 Our Approach . . . 78

6.1.1 First test cycle . . . 79

6.1.2 Second test cycle . . . 92

6.1.3 Third test cycle 93
6.2 Experimental Studies 95
6.2.1 Result analysis and discussion 99
6.3 Summary 102
7 Ranking Use Cases for Testing 104
7.1 Complexity Factors 107
7.1.1 Sum of complexities of Linearly Independent Paths within a SD 108
7.1.2 Number of Test Paths generated within a SD 112
7.1.3 Number of Critical Messages transmitted within a SD 116
7.1.4 Number of Operational Variables used 118
7.1.5 Length of the longest Maximum Message Sequence (MMS) . . 118
7.1.6 Number of External Links used in a SD 120
7.1.7 Number of Polymorphic Calls within a SD 121
7.1.8 Architectural Dependencies among use cases 123
7.2 Computing Complexity and Test Priority 124
7.2.1 Computing the complexity of a use case 125
7.2.2 Computing test priority 125
7.3 Experimental Studies 128
7.4 Summary 135
8 Analyzing Risk at Architectural Level for Testing 136
8.1 Risk Estimation Method 138
8.1.1 Quantifying the complexity for a state of a component 139
8.1.2 Severity analysis 143
8.1.3 Risk computation 147
8.1.4 Complexity analysis of risk estimation approach 153
8.2 Experimental Validation 154
8.2.1 Experiment 1 154
8.2.2 Experiment 2 (Comparison with related work) 156
8.2.3 Applicability 160
8.3 Summary 161
9 Conclusions and Future Direction 162
9.1 Contribution 162
9.1.1 Computing the influence of a component 162
9.1.2 Computing the criticality of a component 163

9.1.3 Conducting multi cycle-based testing 163
9.1.4 Estimating the criticality of a use case 164
9.1.5 Estimating risk at the architectural level for testing 164
9.2 Future Work 165

Bibliography **166**

Dissemination **175**

List of Figures

1.1 Outline of the thesis 10

2.1 A program with its CFG 13
2.2 An example program 17
2.3 A program with its ESDG 20
2.4 Risk structure 25

4.1 Use case diagrams of the case studies 47
4.2 Failure rate of an application based on class reliabilities (one at a time) 49

5.1 An example program with its CDD 65
5.2 Priority of each component after slicing S_1 and S_2 67
5.3 Test execution details for the component CashDispenser . . . 72

6.1 Communication diagram for *withdraw* use case 79
6.2 Program with its Control Dependence Graph 87
6.3 Business Importance estimation 95
6.4 Complexity metrics of ATM obtained through JaBUTi 97
6.5 Test Case execution result and coverage report of component *Transaction* in ATM 98
6.6 Two sample sequence diagrams 101
6.7 A minor failure in ATM 102

7.1 SDs with *asynchronous* and *parallel* messages 109
7.2 CCFG of Figure 7.1b 110
7.3 SD for use case Remove Title 112
7.4 CCFG of Figure 7.3 112
7.5 SD for use case Issue Item 113
7.6 CCFG of the SD of Issue Item use case 114

7.7 **State chart diagrams of various objects of LMS** 115

7.8 **Possible state transitions of objects in *Issue Item* use case** . 116

7.9 **An example of a SD showing message criticality in a Fire Controller system** 117

7.10 **An example of a deployment diagram** 121

7.11 **An example of a Class Diagram** 122

7.12 **An example of polymorphic calls** 123

7.13 **Preceded use cases of use case *Issue Item*** 124

7.14 **High level design of CFE Tool** 127

7.15 **Priority distribution among use cases of LMS case study** . . 129

7.16 **Defect detection rate by various testing methods** 131

8.1 **ISDG of LMS** 140

8.2 **An overview of the severity analysis method** 145

8.3 **Fault tree with its XML form for a hazard of *Issue Item* use case** 146

8.4 **SCOTEM for scenario S_x** 149

8.5 **Interaction Overview Diagram of LMS** 152

List of Tables

2.1 Value assignment 24

4.1 Test Priority Calculation 45
4.2 Brief summary of our case studies 46

5.1 Structural Complexity of various entity classes within LMS 58
5.2 Structural complexity of various entity classes within SMA 58
5.3 Possible error conditions within a class/module 60
5.4 SFMEA at method level for some components within the *Withdraw* Scenario 63
5.5 Business values associated with various components of ATM system . 68
5.6 Criticality computation for *Transfer* component of ATM system . . . 68
5.7 Mutation Score by two testing methods 73
5.8 Failure observation at the time of release 74

6.1 Failure mode of *dispenseCash()* of component *CashDispenser* 89
6.2 Execution Probability of various use cases of ATM 91
6.3 Normalized criticality of various components within *withdraw* scenario 91
6.4 Various types of mutants applied to our case studies 96
6.5 Mutants killed by the two different testing approaches 97
6.6 Test cases designed to test various use cases of ATM case study . . . 100
6.7 Reliability assessment based on two different testing strategies 100

7.1 Possible transitions of various objects in Issue Item use case 115
7.2 Decision table for use case *establish session* 119
7.3 Decision table for use case *Issue Item* 119
7.4 Execution probabilities of use cases with their business values in LMS 126
7.5 Complexity estimation for use case *Issue Item* 127
7.6 Priority calculation 128
7.7 Testing results of three testing methods 130

7.8 Reliability assessment based on two different testing strategies 134

8.1 Complexity for each state of Book component 143
8.2 Estimated risk for various states of *Borrower* component within *Issue Item* use case 148
8.3 Possible transitions by various objects within *Issue Item* use case . . 150
8.4 Risk estimated for various use cases of LMS 151
8.5 Experimental result for various use cases of LMS 156
8.6 Comparison of our work with the existing work 157
8.7 Bugs seeded to LMS 158
8.8 Mutants killed 159
8.9 Failure observation at the time of release 160

Abbreviations

- SDLC: Software Development Life Cycle
- UML: Unified Modeling Language
- CFG: Control Flow Graph
- CCFG: Concurrent Control Flow Graph
- PDG: Program Dependence Graph
- SDG: System Dependence Graph
- ESDG: Extended System Dependence Graph
- SCOTEM: State COllaboration TEst Model
- ISDG: Inter-component State Dependence Graph
- IOD: Interaction Overview Diagram
- CRT: Compatible Reference Type
- CON: Constructor
- OVM: Overriding Method
- AMC: Access Modifier Changes

Chapter 1

Introduction

Testing is the process of exercising a program with the intent of detecting bugs. The basic aim is to increase the confidence in the developed software. Testing enhances the software quality in terms of the total number of test runs, bugs revealed and the percentage of code coverage. Verification, validation and defect finding are the major tasks under software testing.

In software testing literature, four terms are commonly used. These are (i) failure (ii) error (iii) fault (iv) defect. Though, they have related meaning, they differ at some points. An error made by a programmer results in a defect (fault or bug) in the program. The execution of a defect may cause one or more failures. As per the IEEE standard, failure is the inability of a system or a component to perform its required functions within the specified requirements. A failure in a system is observed by the user externally. There are two main goals in software testing: (i) to achieve the adequate quality in which the objective is to search the bugs within a software (ii) to assess the existing quality of the system in which the objective is to assess the reliability of a software system. Based on the testing strategy, the software testing approaches are classified into two types such as *code based testing* and *usage based testing*. The aim of code based testing is to execute each and every statement in a program at least once, during the test [2, 3]. It attempts to cover each reachable elements in the software, within the available test budget. In the code based testing methodologies such as statement, branch and path coverage, each aspect of a program is treated with equal importance [2]. The main aim is to find as many bugs as possible. Usage based testing focuses on detecting bugs that are responsible for frequent failures of the system. Unlike the code based testing, the tester of usage based testing does not require any prior knowledge of the program. In code based testing, the aim is to execute each statement and conditional branch of the program to detect bugs, whereas in usage based testing, the aim is to detect the bugs in the frequently executed parts of the source code, at the early phase of testing.

Testing is an action of sampling. As it is expensive and also some times impossible to perform systematic testing with an adequate test suite due to an infinite state space, the tester needs to take 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 testing team follows prioritization-based testing techniques to solve this problem.

Prioritization-based Testing

The tester prioritizes the testing process with the hope to get the best possible chance to reveal the worst fault. At any instance of testing point, the tester feels that the tests that have been conducted are important than the tests that have not yet been conducted. Testing time is not certain. There is a chance of delay in all other activities before test execution or there is a pressure from market to release the product before scheduled time. The aim of prioritization-based testing is to ensure that the testing resources have been spent cost-effectively, whenever the testing process is terminated. Software industries conduct prioritization-based testing with a number of goals. For example:

- Detecting more bugs at the early phase of testing, when a regression test is conducted using the same test suite.
- Improving the code coverage within the available test resources.
- Improving the reliability of a system within the available test resources.
- Increasing the likelihood of detecting more bugs in the modified parts of the source code.
- Increasing the rate of detecting critical bugs at the early phase of testing process.

Test case prioritization and test case selection approaches have been discussed in software testing literature. A number of researchers [4–8] have considered several criteria for test case prioritization and test case selection. Some of the criteria are (i) coverage of statements, (ii) coverage of statements not yet covered (iii) coverage of functions (iv) coverage of functions not yet covered (v) potential for fault exposing (vi) probability of fault existence/exposure, adjusted to previous coverage etc.

All the existing techniques on test case prioritization and test case selection are purely code-based and require the information on previous usage of the system. Hence, these techniques are mainly used at the post-implementation phase and used only for regression testing. Among the objectives of test case prioritization, the most important one is to maximize the rate of fault detection. The aim is to detect the faults from the important parts of the source code at the early phase of the testing process. Other objectives include the ability to detect important faults and the ability to reveal faults associated with specific code changes or to achieve the target coverage or reliability level as early as possible.

The distribution of test efforts is important to test organization. In this thesis, prioritizition refers to *test effort prioritization* in which components[1]/scenarios are prioritized for testing according to their influence on the overall reliability of the system or severity of failures. Test effort prioritization is a research area under pre-testing effort i.e. before the generation of test cases. The software industry is really interested to save money on testing. As test resources are limited, a proper analysis is needed to decide how much test effort should be given to individual elements, within a system. The test manager should estimate the criticality associated with individual elements in order to decide which parts of the system should be tested thoroughly, within the available test budget. For estimating criticality, the test manager should consider various internal and external factors of a component such as complexity, dependability, severity and the business importance within the system.

1.1 Motivation

An efficient prioritization method can drastically reduce the inefficient effort and help to effectively utilize the test resources. Though, a great effort have been given on prioritization-based testing [4, 9–11], the proposed methods are not so much effective in reducing the failure rate of a system and improving the user's perception on the reliability of a system. Limitations of some prioritization-based testing methods and reason for their low productivity are described below.

The techniques used for code prioritization [11, 12] only find the percentage of

[1]A component refers either to a single item: an object, a class, or a procedure or to a complex item: a package of classes or procedures.

code coverage at the testing phase in a practical system. It cannot find the elements which have high impact on the overall reliability of the system. Testing methods based on operational profile [9, 13] alone did not consider the white-box approach for test effort prioritization. Though some researchers [14, 15] have considered the white-box approach along with operational profile, but they did not consider the data dependencies among components within a system.

Test effort prioritization at the early stage of development cycle makes the testing process effective. Several researchers [16–19] have proposed test effort estimation methods at the early phase, but to the best of our knowledge, no one has proposed a quantitative estimation of complexity for a use case. As, the complexity of a use case is a major input for test effort estimation and prioritization, there is a need to perform analytical complexity assessment at the architectural level, with little or no involvement of subjective measures from domain experts. Keeping these in view, we propose some approaches that attempt to overcome many of the limitations of the existing approaches highlighted above. Now, we discuss the motivations behind our research work.

- A bug in a critical element may cause frequent failures or severe failures of the system. The criticality of an element can be identified through the analysis of source code and the operational profile of the system.

- Some researchers [1, 20, 21] have observed that the return on investment on testing is increased through a *Value based software testing method*, where the business value that come from customer and market is considered as a testing factor. Similarly, there are some components which are executed rarely but a bug in that may cause catastrophic failures. To make the criticality computation process accurate and effective, the external factors of a component such as business value and severity associated with the failure modes should be considered along with its internal factors.

- It is possible to achieve a high quality software product in affordable cost. For this, software testing should be incorporated early into the software development process. It is desirable to identify the critical elements at the architectural level for an effective test resource distribution.

- Risk assessment at the early stage helps to achieve a high level of confidence in a software system. A software system is generally state based. A system behaves differently to the same event, when it is in different states. The state of a system at any time is the composition of the states of the various interacting components (objects) within the system at that time. Hence, we are motivated by the need to develop a methodology to estimate the risk for a state of a component within a scenario and use it to estimate the risk for the scenario and for the system.

With this motivation, we concentrate upon identifying critical elements both at the implementation and architectural level. In the next section, we identify the major objectives of the thesis.

1.2 Objective

Our aim is to estimate the criticality of an element at various phases of software development life cycle by considering various internal and external factors of the element. To address this broad objective, we identify the following goals based on the motivations outlined in the previous section.

- To develop various metrics through static and dynamic analysis of source code and identify the sensible elements within a system.
- To expose the critical elements within a system that have a high influence on the overall reliability of the system or a bug in that component is responsible for severe failures of the system.
- To set different test objectives at different instances of testing phase.
- To get the complexity of a high level function at the early phase of development life cycle based on some quantitative metrics that are analytical in nature rather than subjective measures from domain experts and distribute the test efforts accordingly for an effective testing.
- To estimate the risk for a state of a component within a scenario and use it to compute the risk for the scenario. To generate a list of components within a scenario and a list of scenarios within the whole system ranked by their estimated risks, so that test effort can be distributed accordingly for an effective testing.

1.3 Overview

In order to save time and cost in the Software Development Life Cycle (SDLC), there is a requirement of an effective decision-making for allocating resources to various parts of the software system. In this thesis, we explore some test effort prioritization issues at various phases of software development life cycle. We propose a set of techniques to prioritize the components/use case scenarios for testing at the code level and also, at the design level. At the code level, the potential of a program element to cause failures is measured with the metric *Influence Metric*. Based on a graph-based representation, the effected part of classes are determined. Within a system, we consider the internal and external factors such as the class influence, average execution time, structural complexity, severity and business value for ranking of the importance of a class for testing. We propose a novel approach for reliability improvement that involves the analysis of the dynamic influence and severity of various components within a software system.

A software product can be lunched in due time with sufficient testing, if a test plan is prepared early. As the analysis and design stage is critical compared to other stages of SDLC, detecting and correcting errors at this stage is less costly than later stages. We aim to leverage the architectural complexity and business importance information to assign test priority to use cases. We first analyze the factors that have an effect on the complexity of a use case and then, give a framework to compute test priority. The stakeholders and developers feel that the measurement of the quality of a software system through risk is more significant than other factors such as expected number of residual bugs or failure rate etc. Risk assessment framework takes into account the arguments about the benefits as well as the hazards[2] associated with a system. It helps to take a valuable decision on investment at an early stage. We propose a technique to estimate the reliability-based risk at the design level. Reliability-based risk is estimated based on two factors (i) the probability of the failure of the software product within the operational environment and (ii) the adversity of that failure. We propose a technique to assess the risk of a component at various states within a system, which is used as the basis for establishing the test priority.

A set of experiments are conducted to compare our test effort prioritization techniques to different solutions. Through the experimental results, we observed that our

[2]A hazard is an accident waiting to happen. It is due to faults or failures which occur in a particular context.

proposed techniques guide the tester to expose the critical elements that are getting less attention in terms of testing. In addition to that, our approaches also help to improve the reliability of the system within the available test resources.

1.4 Focus and Contribution of the Thesis

Specifically, the thesis makes the following contributions:

- We propose a framework to compute the criticality of a component within a system and prioritize the components for testing according to their estimated criticality. For this, we introduce a new metric called *Influence Metric* using forward slicing technique to compute the *influence value* of a component towards system failures. It is based on static analysis of the program. We have experimentally proved that decreasing the reliability of a component with high criticality drastically increases the failure rate of the system, whereas it is not true in case of a component with low criticality. In this work, we have not considered the impact of external factors while doing prioritization.

- Though, the *influence value* of a component affects the reliability of a system, this factor alone is not sufficient to estimate the criticality of the component. The reliability calculation only counts the number of failures observed after the testing phase. It does not consider the impact of those failures on the system. The impacts of different failures are different. Some are minor whereas, some are major. Similarly, each high level function does not provide equal benefit to the customer. For criticality estimation, we extend our previous work by adding some internal and external factors such as the average execution time, structural complexity, and severity of failures in the component as well as the component's perceived business value. We have conducted a set of experiments and observed that our approach is effective in guiding test effort as it is linked to both external measure of defect severity and business value, and internal measure of frequency and complexity. Through the experimental results, we observed that our approach helps to improve the reliability of a system within the available test resources. In addition to that, our approach also helps to reduce the post-release failures that have a negative impact on the system.

  In both the approaches, we have prioritized the program elements based on static analysis of the source code. We have not considered the dynamic aspects.

In our next work, we prioritize the program elements based on dynamic analysis of source code.

- In our previous work, we have assigned different priority values to different components, but the priority values remained constant throughout the testing phase. As priority is established by order of importance, a component does not get equal priority throughout the testing phase. We propose a multi cycle-based test effort prioritization technique, in which the priority values of various components change between two test cycles within a system under test. In the first test cycle, we estimate the criticality of a component and assign test priority to the component based on its criticality. Unlike the previous work, we estimate the criticality based on dynamic *Influence Metric*. In a static influence metric, only the information regarding how many other classes request services from a given class is obtained, but in dynamic influence metric, the information regarding how often these requests are executed within a scenario is obtained. In the second test cycle, we assign test priority to a component based on its failure rate in the previous test cycle. We include a Value-based testing approach in the third test cycle. The effectiveness of our proposed testing approach has been validated by applying it to three moderate sized case studies.

  The proposed techniques can be used by testers in software industry for prioritizing the test efforts, where the source codes are available. Since in many cases, the source code may not be available, in our next work, we develop a technique for prioritization of elements at the design level. The technique can be used by tester in software industry, where source code are not available and/or test planing is required much early in the SDLC.

- Planning at high level enhances the decision on resource allocation. Estimating the criticality of an architectural element and performing test effort prioritization based on criticality at high level helps both the system analyst and the test manager in planing suitable provision for the critical elements. If the critical elements will be detected at the early phase of SDLC then, it will be useful in allotting resources in afterward development phase. Keeping this in mind, we propose a technique to rank the use cases within a system for testing based on their internal criteria- *architectural complexity* and external criteria- *business*

value. We first, analyze the factors that have an effect on the complexity of a use case and then, give a framework to compute test priority. The complexity of a use case is computed analytically through a collection of data at the architectural level with little or no involvement of subjective measures from domain experts. In our approach, a high-ranked use case may be more fault-prone or it may add value to the organization. Hence, the failure of a high-ranked use case may create a great loss to the organization.
In all the above work, we have not considered the risk associated with a system. In real practice, risks are associated with every system. Resolving risks at the analysis and design level will improve the quality of the system, within the available resources. In our next work, we develop an approach at the design level for prioritization of elements for testing, considering the risk associated with a system.

- Test effort prioritization based on risk is a powerful technique for streamlining the test effort and delivering the software product with right quality level in limited resources. The tester feels that he is doing the best possible job with the limited resources by exploiting the relationship between risk and testing effort. Risk assessment at an early stage helps to achieve a high level of confidence in the system. We propose an analytical approach for risk assessment of a software system at the design stage. First, we propose a method to estimate the risk for various states of a component within a scenario and then, estimate the risk for the whole scenario. In our previous work, we have assessed the severity at the code level, but in this work, we assess the severity at the design level. We estimate the risk of the overall system based on two inputs: scenarios risks and Interaction Overview Diagram (IOD) of the system. Our risk analysis approach ranks the components/scenarios within a system for testing according to their estimated risks. We performed an experimental comparative analysis and observed that the testing team guided by our risk assessment approach achieves high test efficiency compared to a related approach.

The relationships among the contribution is shown in Figure 1.1. As shown in the figure, the contribution on test effort prioritization is broadly divided into two parts. The first part deals with the analysis of the source code and the second part deals with the analysis of the design model.

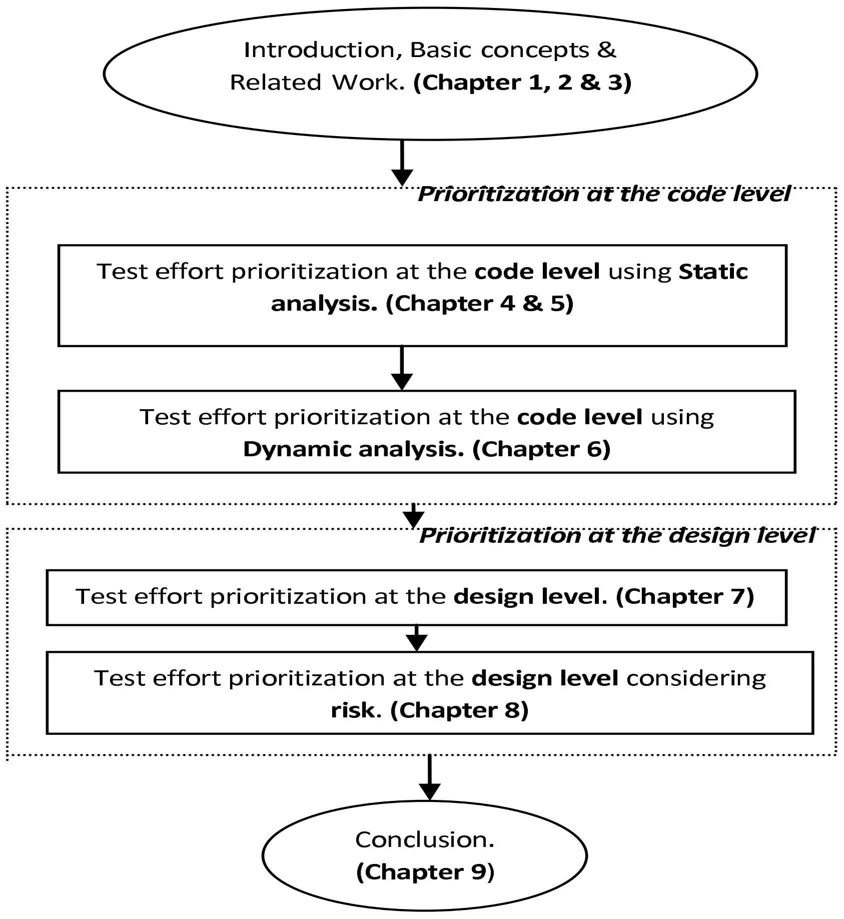

Introduction, Basic concepts & Related Work. (Chapter 1, 2 & 3)
Prioritization at the code level
Test effort prioritization at the code level using Static analysis. (Chapter 4 & 5)
Test effort prioritization at the code level using Dynamic analysis. (Chapter 6)
Prioritization at the design level
Test effort prioritization at the design level. (Chapter 7)
Test effort prioritization at the design level considering risk. (Chapter 8)
Conclusion. (Chapter 9)

Figure 1.1: **Outline of the thesis**

Our proposed prioritization techniques can be used in software industries for analyzing and identifying the important components / scenarios of a software system. Based on the results of the analysis, appropriate test effort can be allocated to different components of the system and the quality of the software can be improved within the available test resources. The proposed severity analysis used for prioritization, will help detect the important errors at the early phase of testing, thus reducing the total test effort. Our risk based testing approach can be used in safety critical systems such as Pace Maker, Nuclear power plant, Air traffic control system etc., for identifying the risks associated with various components / scenarios and the whole system and allocating the test effort accordingly.

1.5 Organization of the Thesis

The rest of this thesis is organized into chapters as follows.

1. *Chapter 2* discusses the background concepts used in the thesis.

2. *Chapter 3* provides a brief review of the related work relevant to our contribution.

3. *Chapter 4* presents a novel approach to get the influence of a component towards system failures. We propose a metric, called *Influence Metric*, through static analysis of source code and use it as a factor for prioritizing program elements at the code level.

4. *Chapter 5* presents a novel approach to prioritize classes according to their potential to cause failures and severity of those failures. This is a very important and interesting problem for software testing. This chapter extends the work in Chapter 4 by adding some contributing factors- structural complexity, severity and business value- for test effort prioritization.

5. *Chapter 6* presents a multi cycle-based test effort prioritization approach to improve the reliability of a system within the available test resources, through the dynamic analysis of source code.

6. *Chapter 7* presents an approach to estimate the test effort based on the prioritization of use cases in the design level of software development life cycle. Our approach quantifies a method for estimating the test effort of a software system based on use cases. It provides experimental results that appear to substantiate the method.

7. *Chapter 8* presents a risk estimation approach of software system at the architectural level. The main idea consists in using UML sequence and state diagrams, in order to calculate an overall risk factor associated to a selected architecture.

8. *Chapter 9* concludes the thesis with a summary of our contributions. We also briefly discuss the possible future extensions to our work.

Chapter 2

Background

This chapter provides a general idea of the background used in the rest of the thesis. For the sake of conciseness, we do not discuss a detailed description of the background theory. We just highlight the basic concepts and definitions by providing a short introduction. The basic concepts and definitions are used in subsequent chapters of this thesis. Section 2.1 gives an introduction to software testing. Section 2.2 presents the concept of McCabes cyclomatic complexity. Section 2.3 presents the concept of Halstead complexity metrics. Section 2.4 contains the basic concept of program slicing which will be used later in our Influence Metric computation algorithms. Section 2.5 provides the intermediate program representation that is used for extracting slices of a program. Section 2.6 gives an overview of Unified Modeling Language (UML) and its advantages. Section 2.7 gives introduction of a metric called Chidamber & Kemerer Suite of Metrics (CK metrics) to analyze the complexity of an object-oriented program. Section 2.8 gives an introduction to Value based testing technique. Section 2.9 presents the basic concepts on Operational Profile of a system which is used in various testing approaches for achieving and assessing the reliability of a system. Section 2.10 briefly discusses the concepts of risk-based testing. Section 2.11 summarizes this chapter.

2.1 Object-Oriented Technology and Software Testing

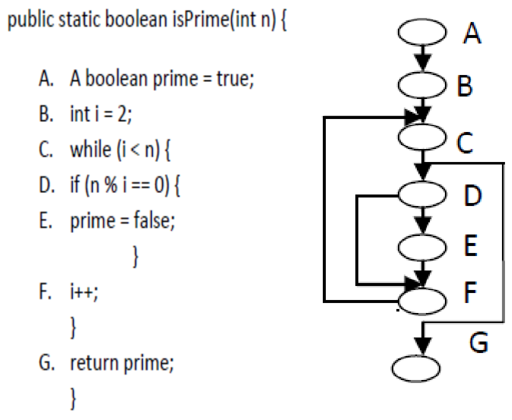
It is widely accepted that the object-oriented (O-O) paradigm will significantly increase the software reusability, extendibility, inter-operability, and reliability. This is also true for high assurance systems engineering, provided that the systems are tested adequately. Object-oriented software testing (OOST) [22] is an important software quality assurance activity to ensure that the benefits of object-oriented (O-O) programming will be realized. Below, we discuss different levels of testing associated with object-oriented programs.

1. Intra-method testing: Tests designed for individual methods. This is equivalent to unit testing of conventional programs.
2. Inter-method testing: Tests are constructed for pairs of method within the same class. In other words, tests are designed to test interactions of the methods.
3. Intra-class testing: Tests are constructed for a single entire class, usually as sequences of calls to methods within the class.
4. Inter-class testing: It is meant to test a number of classes at the same time. It is equivalent to integration testing.

The first three variations are of unit and module testing type, whereas inter-class testing is a type of integration testing. The overall strategy for object-oriented software testing is identical to the one applied for conventional software testing but differs in the approach it uses. We begin testing in small and work towards testing in the large. As classes are integrated into an object-oriented architecture, the system as a whole is tested to ensure that errors in requirements are uncovered.

2.2 McCabes Cyclomatic Complexity

Cyclomatic Complexity (v(G)) [23] is a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and therefore, the



```
public static boolean isPrime(int n) {
    A.  A boolean prime = true;
    B.  int i = 2;
    C.  while (i < n) {
    D.  if (n % i == 0) {
    E.  prime = false;
                }
    F.  i++;
        }
    G.  return prime;
        }
```

(a) A program (b) CFG

Figure 2.1: **A program with its CFG**

minimum number of paths that should be tested. If the structure of source code is complex, it is hard to understand, to change and to reuse. The cyclometic complexity measures the number of linearly independent paths through the Control Flow Graph (CFG) of the program. v(F) = e - n + 2, where F is the CFG of the program, n the number of vertices and e the number of edges. We present a program with its CFG in Figure 2.1. In the program, n=7, e=8. So, the cyclomatic complexity for the program is: 8-7+2=3.

2.3 Halstead Complexity Metric

Any programming language is defined by declarative instructions definitions, executable instructions. The operators and operands are handled within expressions. The programs are made up of instructions, written in sequences, without taking into account the running order. Halstead [24] makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. The metrics proposed by Halstead are computed through the static analysis of the source code. He estimated the programming effort. The measurable and countable properties are:

- n1 = number of unique or distinct operators appearing in the source code.
- n2 = number of unique or distinct operands appearing in that source code.
- N1 = total usage of all of the operators appearing in that source code.
- N2 = total usage of all of the operands appearing in that source code.

The number of unique operators and operands (n1 and n2) as well as the total number of operators and operands (N1 and N2) are calculated by collecting the frequencies of each operator and operand token of the source program. Halstead defines:

- The program length (N) is the sum of the total number of operators and operands in the program, $N = N1 + N2$
- The vocabulary size (n) is the sum of the number of unique operators and operands in the program, $n = n1 + n2$.
- The program volume (V) is the information contents of the program, $V = N * log_2(n)$

- The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program. D is also proportional to the ration between the total number of operands and the number of unique operands in the program, $D = (n1/2) * (N2/n2)$

2.4 Program Slice

Program slicing is a program analysis technique. It is used to extract the statements of a program that are relevant to a given computation. A program slice consists of the parts or components of a program that (potentially) affect the values computed at some point of interest. Program slices are computed with respect to a slicing criterion. For a statement s and variable v, the slice of a program P with respect to the slicing criterion $< s, v >$ includes only those statements of P that are needed to *capture* the *behavior* of v at s [25]. According to Weiser [25], a program slice is a reduced and executable program obtained from a program by removing statements, such that the slice replicates part of the behavior of the program.

Slicing object-oriented programs presents new challenges which are not encountered in traditional program slicing [26]. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism need to be considered carefully [27]. Larson and Harrold were the first to consider these aspects in their work [28]. To address these object-oriented features, they enhanced the system dependence graphs (SDG) [29] to represent object-oriented software. After the SDG is constructed, the two phase algorithm of Horwitz et al. [29] is used with minor medications for computing static slices. Larson and Harrold [28] have reported only a static slicing technique for object-oriented programs, and did not address dynamic slicing aspects. The dynamic slicing aspects have been reported by Song et al. [30] and Xu et al. [31].

2.4.1 Categories of program slicing

Several categories of program slicing as well as methods to compute them are found in literature. The main reason for the existence of so many categories of slicing is the fact that different applications require different types of slices.

Static Slicing and Dynamic Slicing: Slicing can be static or dynamic. Static slicing technique uses static analysis to derive slicing. That is, the source code of the program is analyzed and the slices are computed for all possible input values.

No assumptions are made about the input values. It is static in the sense that the slice is independent of the input values to the program. Since, the predicates may evaluate either to true or false for different values, conservative assumptions have to be made, which may lead to relatively large slices. So, a static slice may contain statements that might not be executed during an actual run of a program, whereas dynamic slicing makes use of the information about a particular execution of a program. The execution of a program is monitored and the dynamic slices are computed with respect to execution history. A dynamic slice with respect to a slicing criterion $< s, v >$, for a particular execution, contains only those statements that actually affect the slicing criterion in the particular execution. Dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing. A major goal of any dynamic slicing technique is efficiency since results are normally used during interactive applications such as program debugging [32]. Efficiency is an especially important concern in slicing object-oriented programs, since the size of practical object-oriented programs is often very large. The response time of an inefficient dynamic slicer may be unacceptably large for such programs. In all slicing techniques, the source code is first analyzed to produce a graph representation called an *intermediate program representation*. Then the intermediate program representation is analyzed by using an algorithm to compute the slice. So, the efficiency of a slicing technique depends on how suitably the program is represented by an intermediate representation and how much efficient the slicing algorithm is.

Consider the C++ example program given in Figure 2.2. The static slice with respect to the slicing criterion $< 11; sum >$ is the set of statements {4, 5, 6, 8, 9}. Consider a particular execution of the program with the input value i = 15. The dynamic slice with respect to the slicing criterion $< 11; sum >$ for the particular execution of the program is the statement {5}.

Backward and Forward slicing: Slices can be backward or forward. A *backward slice* contains all parts of the program that might directly or indirectly affect the slicing criterion but, a *forward slice* with respect to a slicing criterion $< s, v >$ contains all the parts of the program that might be affected by the variables in v used or defined at the program points. A forward slice provides the answer to the question: “which statements will be affected by the slicing criterion?” whereas, a backward slice provides the answer to the question: “which statements affect the

```
main()
{
int i,sum;
cin>>i;
sum=0;
while(i<=10)
{
sum=sum+i;
++i;
}
cout<<sum;
cout<<i;
}
```

Figure 2.2: **An example program**

slicing criterion?” [33].

Intra-procedural Slicing and Inter-procedural Slicing: Intra-procedural slicing computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, inter-procedural slicing can be used to derive slices that span multiple procedures [29]. For object-oriented programs, intra-procedural slicing is meaning less as practical object-oriented programs contain more than one method. So, for object-oriented programs, inter-procedural slicing is more useful.

2.4.2 Applications of program slicing

Slicing is used by both developer and tester, before the execution of the code and during execution. The developer uses slicing tool to understand the source code and to reduce the size of a program. Sometimes a programmer has to read a lot of code before finding what he is actually looking for. Programmer uses the slicing tool to improve the productivity. The tool helps the programmer in reducing the amount of code that need to read. The tool is used by the developer for debugging. Some variables may show unexpected values at some point in the program. To know the exact cause of these values is difficult and also time taking. The slicing tool helps a lot in this case. The tester uses the slicing tool for analyzing the test coverage of the test suite [7,34]. The dynamic slice is created for each test case of a test suite and the union of these slices are computed to get an idea of code coverage by the test suite. Recently, Qusef et al. [35] proposed a novel approach to maintain the traceability links between unit tests and tested classes based on dynamic slicing.

2.5 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. In the context of program slicing, program representations are used to support automation of slicing. Various representation schemes have resulted from the search for ever more complete and efficient slicing techniques.

2.5.1 Program Dependence Graph (PDG)

The program dependence graph [36] 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 (or data) dependence edges (m, n) indicates that n is control (or data) dependent on m. Note that the PDG of a program P is the union of a pair of graphs: Data dependence graph and control flow graph of P.

2.5.2 System Dependence Graph (SDG)

The PDG cannot handle procedure calls. Horwitz et al. [29] introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures. 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 dependence edges which link the various subgraphs together.

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

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

Control dependence edges and data dependence edges are used to link an individual PDG in an SDG. The additional edges that are used to link a PDG 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.
- Summary edges connects an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex may affect the value in actual-out vertex. It represents the transitive dependencies that arise due to procedure calls.

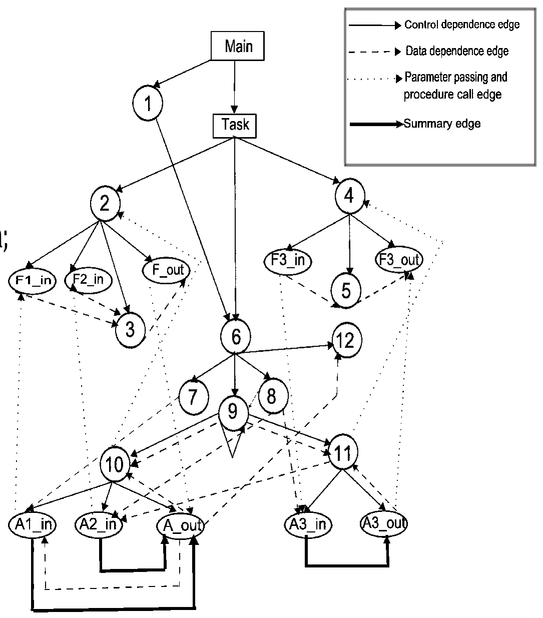
2.5.3 Extended System Dependence Graph (ESDG)

ESDG models the main program with all other methods. Each class in a given program is represented by a class dependence graph. Each method in a class dependence graph is represented by procedure dependence graph. Each method has method entry vertex that represent the entry in the method. The class dependence graph contains a class entry vertex that is connected with the method entry vertex of each method in the class by a special edge known as class member edge. To model parameter passing, the class dependence graph associates each method entry vertex with formal-in and formal-out vertices.

The class dependence graph uses a call vertex to represent a method call. At each call vertex, there are actual-in and actual-out vertices to match with the formal-in and formal-out vertices present at the entry to the called method. If the actual-in vertices affect the actual-out vertices then summary edges are added at the call-site, from actual-in vertices to actual-out vertices to represent the transitive dependencies. To represent inheritance, we construct representations for each new method defined by the derived class, and reuse the representations of all other methods that are inherited from the base class. To represent the polymorphic method call, the ESDG uses a polymorphic vertex. A polymorphic vertex represents the dynamic choice among the possible destinations. The detailed procedure for constructing an ESDG is found in [28]. Each node can be a simple statement or a call statement or a class entry or a method entry. An example of an object-oriented program with its ESDG is shown in Figure 2.3. Several researchers [4,37,38] have proposed different types of

```
Class Task {                    9  while (i < 11) {
      public:                   10 sum=add (sum, i);
2  int add (int x, int y) {     11 i = incr (i);}
3  return ( x + y);}            12 cout<< "SUM="<<sum;
4  int incr (int i) {           }
5  i=i+1;}                      main ( ) {
6  void fun ( ) {                     Task ob;
7  int sum=0;                   1     ob.fun ( );
8  int i = 1;                         }
```

(a) An Object-Oriented Program



(b) Its ESDG

Figure 2.3: **A program with its ESDG**

intermediate representation for object-oriented software. Rothermel and Harrold [4] extended Program Dependence Graph (PDG) and proposed Class Dependence Graph (ClDG) for use in regression testing. Larsen and Harrold [28] extended the System Dependence Graph (SDG) by representing a class with a ClDG, and proposed Extended System Dependence Graph (ESDG) for object-oriented software. The basic aim of designing ESDG was to get a slice of an object-oriented program on the basis of graph reachability. Liang and Harrold [38] proposed extensions to ESDG for the purpose of object-slicing. Malloy et al. [37] also proposed a layered representation, the Object-Oriented Program Dependency Graph (OPDG), by adapting the basic concepts of PDG. Out of these, we consider the ESDG by Larsen and Harrold [28] in our work because, our main aim is to get a forward slice of a method-entry vertex through the process of graph reachability. Throughout the thesis, we use the terms node and vertex interchangeably.

2.6 Unified Modeling Language (UML)

Models are the intermediate artifacts between requirement specification and source code. Models preserve the essential information from requirement specification and are base for the final implementation. UML has emerged as an industrial standard for modeling software systems [39]. It is a visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. UML can be used to describe different aspects of a system including static, dynamic and use case views of a system. UML supports object-oriented features at the core. It accomplish the visualization of software at early stage of development cycle, which helps in many ways like confidence of both developer and the end user on the system, earlier error detection through proper analysis of design and etc. UML also helps in making the proper documentation of the software and so maintains the consistency in between the specification and design document. UML diagrams can be divided into two broad categories: structural and behavioral diagrams. The UML structural diagrams are used to model the static organization of the different elements in the system, whereas behavioral diagrams focus on the dynamic aspects of the system. Our approaches use information present in three behavioral diagrams, namely use case, sequence and state chart diagrams.

Use case diagrams represent the high level functionalities (called use cases) of a system from the perspective of the users. It is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. A use case comprises different possible sequence of interactions between the user and the computer. Each specific sequence of interactions in a use case is called a scenario. Use case diagrams are mainly used for requirement based testing and high level test design [40]. Sequence diagram describes how a set of objects interact with each other to achieve a behavioral goal. It captures time dependent sequences of interactions down between objects. It shows the chronological sequence of the messages, their names and responses and their possible arguments. State chart diagrams capture the dynamic behavior of class instances. It describes object state transition behavior. Typically, it is used for describing the behavior of class instances.

2.7 CK Metrics

CK metrics [41] were designed to measure the complexity of the design of object-oriented system. CK metrics measured from the source code have been related to: fault-proneness, productivity, rework effort, design effort and maintenance. It helps in taking managerial decisions, such as re-designing and/or assigning extra or higher skilled resources to develop, to test and to maintain the software. The set of metrics are:

1. WMC (Weighted Methods per Class): It is the sum of the complexity of the methods of a class.
   WMC = Number of Methods (NOM), when all methods complexity are considered unity. It is a predictor of how much time and effort is required to develop and to maintain the class.

2. DIT (Depth of Inheritance Tree): The maximum length from the node to the root of the tree. DIT with high value makes complex to predict the behaviour of the class.

3. NOC (Number of Children): Number of immediate subclasses subordinated to a class in the class hierarchy. NOC with high value increases the requirements of method's testing in that class.

4. CBO (Coupling Between Objects): It is a count of the number of other classes to which it is coupled. CBO with low value improves modularity and promote encapsulation, indicates independence in the class and makes easier to maintain and test a class.

5. RFC (Response for Class): It is the number of methods of the class plus the number of methods called by any of those methods. RFC with high value makes complex the testing and maintenance of the class.

6. LCOM (Lack of Cohesion of Methods): Measures the dissimilarity of methods in a class via instanced variables. LCOM with high value does not promotes encapsulation and implies classes should probably be split into two or more subclasses.

2.8 Value-based Testing

In Value-neutral testing method, each use case is considered equally important and hence, the test effort for a use case is linear to the factor complexity. Value-based testing method focuses the test effort on the features (use cases) that provide a high system value [1, 20, 21, 42]. The addition of Value (say, business value) helps to maximize the returns on investment on the resources allocated to testing [43]. Boehm [42] had considered some case studies and found that 20% test cases cover 80% business value. He had pointed that the main reason for majority of software crises is due to generating value-neutral test data. He pointed that Value-based testing provides more net value and hence, test data generator based on business value cut the test costs in half.

For a developer, it is a difficult task to guess which high level functions are important to the customer. A customer also cannot estimate the cost and technical difficulties in implementing a specific high level function. The requirements are classified into three categories: (i) must have (ii) important to have (iii) nice but unnecessary. The domain experts first collect a list of requirements, which are important for the customer and the end-user and then, prioritize the requirements based on the business value that come from market and customer. From a business point of view, test effort distribution based on the return on investment will be more effective. It is because, the failure of a scenario may cause a great loss to the stake holder and to the organization.

The prioritizing requirements model proposed in [44] is used for getting Value for different requirements. It consists of eight steps and it includes a number of participants involved with the system such as project manager, key customer representatives and development representatives. The Value for a use case is assessed by considering both the benefit and penalty due to the presence and absence of the use case. The following steps show a simple method adopted in various software industries for estimating the business value associated with high level functions [43].

1. The relative benefit that each feature provides to the customer or business. It is estimated on a scale from 1 to 9, where 1 and 9 indicates the minimum benefit and the maximum possible benefit respectively. The best people to judge these benefits are the domain experts and the customer representatives.

2. The relative penalty by not including a feature is also estimated. It represents how much the customer or business would suffer, if the feature is not included within the system. For this penalty, a scale from 1 to 9 is also used, where 1 stands for no penalty and 9 represents the highest penalty.

3. The sum of the relative benefit and penalty gives the total business value called Value. By default, benefit and penalty are weighted equally. The weights for these two factors can be changed. We have rated the benefit twice as heavily as the penalty ratings as defined in [21, 42].

For example, the business values for various use cases of Automatic Teller Machine (ATM) system are shown in Table 2.1. We consider only the use cases that are used by the customer. *start-up* and *shut-down* use cases are not considered as they are the basic use cases to run the system.

Table 2.1: Value assignment

Relative_Weights	2	1	-	-
$Usecase$	$Benefit$	$Penality$	$Total - Value$	$Value\%$
withdraw	8	9	25	20
deposit	7	5	19	24
transfer money	9	5	23	27
inquiry balance	9	9	27	29
SUM	-	-	94	100

2.9 Operational Profile

According to Musa [9], a profile is a collection of disjoint (only one can occur at a time) alternatives with some probability assigned for each occurrence. An *operational profile* simply consists of a set of operations that a system is designed to perform along with its probabilities of occurrence. It predicts the possible use of the system in the operational environment in a quantitative manner. It is widely used in the field of software reliability engineering.

An operational profile assigns probability values to various high level functions (use cases) according to their probability of use by various users within a system [45–47]. Suppose, we have drawn a use case diagram consisting of m types of users and n number of use cases for a system. Each user type has been assigned a probability of using the system. Let u_i be the probability assigned to i-th user type for accessing the system such that $\sum_{i=1}^{m} u_i = 1$. Let q_{ij} be the probability of requesting the functionality of j-th use case (j=1...n) by i-th type user (i=1..m) such that $\sum_{j=1}^{n} q_{ij} = 1$. Then, the probability of a use case x denotes the likelihood of the use case being executed by an average user is given by:

$$p(x) = \sum_{j=1}^{m} u_j * q_{jx} \tag{2.1}$$

We consider that the functionality of any system can be modeled through a set of scenarios derived from use cases [40]. A use case consists of one main scenario and a number of alternative scenarios. As per the domain knowledge, a scenario of a use case is assigned some frequency based on its execution in the operational environment. Let $f_i(j)$ be the frequency of j-th scenario of i-th use case such that $\sum_{j=1}^{nos_i} f_i(j) = 1$ where, nos_i is the total number of scenarios of i-th use case. Then, the probability of execution of k-th scenario of i-th use case, $p(k_i)$ is given by:

$$p(k_i) = p(i) * f_i(k) \tag{2.2}$$

2.10 Risk-based Testing

In order to save time and cost in the software development life cycle, there is a requirement of an effective decision-making for allocating resources to various high level requirements. For this, there is a need to assess quantitatively all possible types of risks associated with high level requirements as early as possible. Risk is the combination of damage that occur due to failure and probability of failure in the operational environment, as shown in Figure 2.4. Risk analysis is important for a critical real-time application and it is basically done to assess the damage during use, frequency of use, and to decide the probability of failure by looking at defect [48]. There are several types of risks such as reliability-based risk, availability-based risk, acceptance-based risk, performance-based risk, cost-based risk, and schedule-based risk. We are mainly concerned with reliability-based risk. It is the probability that the software product will fail in the operational environment and the adversity of that failure. Risk assessment framework takes into account arguments about benefits as

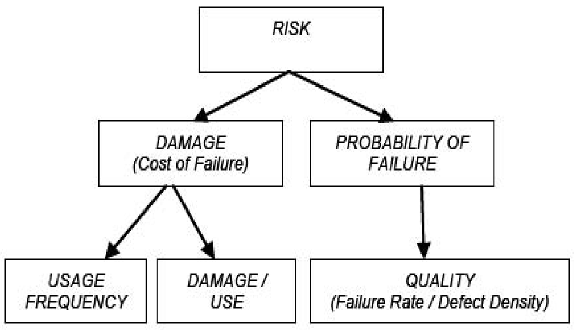


Figure 2.4: **Risk structure**

well as hazards. It helps to take a valuable decision on investment at an early stage.

2.11 Summary

In this chapter, we have discussed the slicing concept and intermediate program representation that will be used later in our thesis. We have discussed the estimation of business value with an example. We have also given an introduction of risk associated with a system under test.

Chapter 3

Related Work

In this chapter, we review the literature and present a brief summary of the work done related to prioritization-based testing at both the implementation and architectural level. The different approaches proposed in this direction by different researchers can be broadly categorized into two types: *pre-testing effort* prioritization (before the construction of test cases) and *on-testing effort* prioritization (at the time of test case selection in a test suite). Pre-testing effort prioritization methods help to prioritize the *program elements* for testing whereas, on-testing effort prioritization methods prioritize the *test cases* within a test suite. We first discuss the reported work on *pre-testing effort* prioritization methods in Section 3.1 followed by *on-testing effort* prioritization methods in Section 3.2. As our aim is to improve the reliability of a system under test through test effort prioritization, we discuss a number of reliability models for assessing and achieving the reliability of a software system in Section 3.3. We propose a test effort prioritization method at the architectural level to rank the use cases within a system for testing. For this, we discuss some early effort estimation and prioritization methods (development effort and testing effort) based on use cases in Section 3.4. For achieving a better reliability through testing, it is required to estimate the reliability-based risk for various elements within a system at the early phase and prioritize the elements according to their estimated risk. We present a brief summary of the work done on risk assessment in Section 3.5. Finally, we present the summary of the chapter in Section 3.6.

3.1 Pre-testing Effort

The basic aim of a pre-testing effort prioritization method is to prioritize the test effort of an application based on its test objectives. With a prioritized test effort and focused test architecture, test cases are created and executed. The research areas coming under this category are: code prioritization for improving test coverage, test effort prioritization based on fault-proneness and usage-based testing.

3.1.1 Code prioritization

Code prioritization is a testing technique which is used for improving the code coverage in a coverage-based testing. Code coverage is a metric that represents how much of the source code for an application run when the unit tests for the application are run. It is basically used for measuring the thoroughness of software testing.

Li. [11] proposed a priority calculation method that prioritizes and highlights the important parts of the source code based on *dominator analysis*, that need to be tested first to quickly improve the code coverage. His approach consists of two major contributions: (i) considers the impact of calling relationship among methods/functions of complex software and takes a global view of the execution of a program being tested (ii) relaxes the guaranteed condition of traditional dominator analysis to be at least relationship among dominating nodes. Relaxing the guaranteed condition makes dominator calculation much simpler without losing its accuracy. His approach expands this modified dominator analysis to include global impact of code coverage, i.e. the coverage of the entire software other than just the current function.

Before test construction, Li's method decides which line of code will be tested first to quickly improve code coverage. According to his approach, first the intermediate representation of the source code, known as Control Flow Graph (CFG) is constructed. Then, a node[1] of the CFG is prioritized based on measuring quantitatively how much lines of code are covered by testing that node. A weight is calculated for each node considering only the coverage information. It does not take into account, for instance, the complexity or the criticality of a given part of the program. A test case covering the highest weight node will increase the coverage faster[2]. There are two kinds of code coverage such as control flow based and data flow based. Li's work focuses on control flow coverage.

Li et al. [12] presented a methodology for code coverage-based path selection and test data generation, based on Li's previous work [11]. They [12] proposed a path selection technique that considers the program priority and call relationships among

[1]A node is either a statement or a method or a basic block in the source code

[2]The tester, based on his/her experience may desire to cover first a node with a lower weight but that has a higher complexity or criticality

class methods to identify a set of paths through the code, which has high priority code unit. Then, constraint analysis method is used to find object attributes and method parameter values for generating tests to traverse through the selected sequence of paths. It helps to automatically generate tests to cover high priority points and minimize the cost of unit testing.

Code coverage is a sensible and practical measure of test effectiveness [49]. It helps the developers and vendors to indicate the confidence level in the readiness of their software but, the limitation is that it gives equal importance to the discovery of each fault. So, no information is gained on how much it affects the reliability of a system by detecting and eliminating a fault during the testing process, as different faults have different contribution to the reliability of a system.

3.1.2 Fault-prone based testing

Fault-prone based testing approach identifies the faulty components in a system and test effort prioritization is done accordingly. It estimates the probability of the presence of faults within a component, which helps to take a valuable decision on testing. There has been significant amount of research [50–54] in software industry to identify the fault-prone components within a system and prioritizing the test accordingly. Different authors have focused on different characteristics associated with a component for counting faults.

Eaddy et al. [50] experimentally proved that concern-oriented metrics[3] are more appropriate predictors of software quality than structural complexity measures and there is a strong relationship between scattering and defects.

Czerwonka et al. [54] discussed the application of CRANE tool set on a large scale software product, *Windows Vista*, to expose the required information such as code churn, code complexity, dependencies, pre-release bugs with the purpose to make a decision for failure prediction, change analysis and test prioritization to minimize risks of further problems in changed code.

Ostrand et.al. [51] proposed a novel approach to identify the faulty files at the time of next release of an application. For prioritizing testing efforts, their approach considers the factors that are obtained from the modification requests and the version control system. These factors are (i) the file size (ii) file status (whether the file was new to the system) (iii) fault status in previous release (iv) number of changes

[3]A concern is anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design constraints.

made. For some initial releases, the models were customized based on the above observed factors. Based on the experimental results, the authors concluded that their methodology can be implemented in the real world without extensive statistical expertise or modeling effort.

Ostrand et. al [53] proposed a negative binomial regression model. The binomial model is used to predict the expected number of faults in each file of the next release of a system. The predictions are based on the code of the file in the current release, and fault and modification history of the file from previous releases. Similarly, Emam et al. [52] found that a class having high export coupling value is more fault-prone. A complex program might contains more faults compared to a simple program [55]. As the factor complexity is the most important defect generator, the *complexity metric* is used as a parameter for testing [56, 57].

We present some existing work on prediction of faulty components through design metric. Researchers [56,58] related the structural complexity metric obtained through CK metric suite [41] to the fault-proneness of a system. It is observed that the *estimated defect density* that is computed through *static analysis* and the *pre-release defect density* that is computed through *testing* are strongly correlated. Emam et al. [52] experimentally proved that inheritance and external coupling metrics are strongly associated with fault-proneness.

3.2 On-testing Effort

The basic job of on-testing effort is to identify the important test cases within an existing test suite with the aim to reduce the test cost. In this section, we briefly present the work done on two sub areas: *test case prioritization* and *test case selection*, which are under the main research area, on-testing efforts. A meaningful prioritization or selection of test cases from a test suite can enhance the effectiveness of testing, without increasing the test effort [4]. Test case selection and test case prioritization are both interlinked. The basic difference between these two techniques is as follows.

In a test case selection technique, a subset of the test suite is selected in which, the test coverage of the selected subset is same as the original test suite. However, in a test case prioritization technique, the test cases of a test suite are ranked for testing according to their estimated priority. Within a test suite, a test case with the highest priority is executed first and a test case with the lowest priority is executed last. Test case prioritization and test case selection approaches have been discussed in software testing literature. A number of researchers [4–8] have considered several criteria for test case prioritization and test case selection. Some of the criteria are:

1. Coverage of statements.
2. Coverage of statements not yet covered.
3. Coverage of functions.
4. Coverage of functions not yet covered.
5. Fault exposing potential.
6. Probability of fault existence/exposure, adjusted to previous coverage.
7. Relevant slices of outputs.

An empirical research work [4–6] proposed on test case prioritization in which, statement-level and function-level coverage techniques are used for test case prioritization. The basic aim of these approaches is to improve a test suite's fault detection rate and to reduce the cost of regression testing based on total requirement and additional requirement coverage.

Elbaum et al. [5] proposed a metric named as *Average of the Percentage of Faults Detected* (APFD). The metric is used to measure the ability of rate of fault detection of a sequence of test cases according to a prioritization technique. They used *greedy* strategy for selecting the test cases from a test suite in regression testing. In the greedy strategy, a test case with the highest statement coverage was selected first. Each time, after the selection of the best test case (test case with the highest statement coverage) for execution, the remaining test cases were again ordered based on the criteria; *the coverage of un-covered statements* (the statements that are not yet covered by the already executed test cases). They did not consider the statements which were already covered by the executed test cases. Test case selection and execution, which is iterative in nature was continued in a test suite, till the coverage of each statement by at least one test case. Though, this scheme helps the tester in achieving full statement coverage within a program by using as few test cases as possible, it does not ensure in the improvement of the reliability of a software product using a fixed size test suite. The limitation with the test case prioritization approaches is that the metric, APFD, used for prioritization, gives equal importance to each detected fault. In this technique, the assumption make that all detected faults are of equal severity and all test cases have equal costs, which is not true for a practical application.

To solve this problem, Elbaum et al. [6] extended their previous work [5] by adding two major attributes: (i) test cost and (ii) fault severity. They adapted their previously proposed *APFD* metric and proposed a new cost-cognizant metric, *APFDc*. In their approach, the cost of a test case might be measured in terms of test execution, setup and test validation. The cost of a test case might be also measured in terms of hardware costs or the cost of hiring tester. They measured the severity of a fault was measured in any one of the two ways: (i) the time required to locate the fault and correct the fault (ii) the impact of failures that are caused by the faults.

Jeffrey and Gupta [10] also proposed a new approach for prioritization of test cases for early detection of faults in the regression testing process. They considered the statements of a program, which are influencing or may influence the output through the consideration of *relevant slicing* on the output of a program.

Recently, Bryce et al. [8] proposed various criteria for prioritizing test cases. These are (i) Parameter-value interaction coverage-based, (ii) Count-based and (iii) Frequency-based. They applied these criteria to some stand-alone GUI and Web-based applications and found that the fault detection rate is increasing over random ordering of test cases.

All these discussed test case prioritization techniques are purely code-based and require the information on previous usage of the system. These techniques are mainly used at the post-implementation phase and used mainly for regression testing.

Test cases can also be prioritized based on the design model. Kundu et al. [59] had proposed a technique called *System Testing for Object-Oriented systems with test case Prioritization, STOOP*, to generate test cases from UML 2.0 Sequence Diagrams for system testing. They had prioritized those test cases based on three prioritization metrics: (i) sum of message weights (ii) average weighted path length and (iii) code weight. Their prioritization technique is applied at the development phase and helps to increase the confidence in reliability of the system at a faster rate.

A lot of research [7, 60–62] have been done on test case selection technique for regression testing. Harrold et al. [61] had proposed data flow testing method for reducing test suite whereas, program slicing technique is used for reducing test suite in [7]. Similarly, Leon et al. [60] had used information flow through both data and control dependency for test case selection. They had proposed two approaches such as coverage-based and profile-distribution-based technique for this. An improved version of test case selection technique is proposed by Jeffery and Gupta [62], in which additional coverage technique is used that adds some extra test cases which are redundant with respect to test suite minimization criteria.

3.3 Empirical Work on Reliability Analysis

A number of reliability models [9, 14, 15, 57, 63] have been proposed for assessing and improving the reliability of a software system over several decades. Some researchers have considered system as a black box whereas, others have included the architecture of the system in their analysis.

Musa [9] is recognized for his work in the field of test suite design using *Operational Profile*. *Operational Profile* is a quantitative characterization of how a system will be used. An operational profile is used to guide testing. If testing is terminated and the product is shipped due to crucial schedule constraints, the tester is ensured that the most-used operations will have received the most testing effort. According to Musa, the reliability of a software product depends on how the product will be used by a customer. The testing should be conducted as if the product is in the field. The chance of failure is high in a module with high execution probability. If a module is executed more frequently, then the probability of activation of any residing error in that module is high, which may cause frequent failures. Based on this idea, he had proposed a technique to prioritize input-domains or *fault-regions* on the basis of their impact on the overall reliability of the system. His proposed testing method is for both assessing and enhancing the reliability of a system according to user's point of view.

Testing based on operational profile is efficient and effective in revealing bugs (compared to coverage-based testing) that influence the reliability of a system at the operational environment [13]. Cobb et. al [13] had experimentally proved that in terms of Mean Time Between Failure (MTBF), operational profile based testing improves the perceived reliability during operation 21 times greater than coverage-based testing. They also proved that even if the operational profile is not accurate during testing, there is a high probability that MTBF at the operation time will be much higher than that obtained with coverage-based testing or other black box testing approaches.

Not only at the testing time, but also during software inspection, Usage-Based Reading (UBR) [64] technique helps reviewers to find quickly the faults that have the most negative impact on the user's perception on system quality. The use cases are prioritized based on their execution probability and handed over to the reviewers for inspection. UBR guides the reviewers to focus the software parts that are most important for a user. The limitation with these discussed methods is that failure regions for a practical program was decided only considering a black-box approach. It is a challenging job.

Reliability prediction will be more accurate, if internal structure (interaction among components) of the system will be considered along with the operational profile of the system. Goseva-Popstojanova et al. [65] proposed that, there are broadly two categories of architecture-based analysis such as state-based [14, 15, 66] and path-based [67–70]. In a state-based analysis, the probabilistic control flow graph is mapped to a state space model and transition probability between components is decided based on Markov property and operational profile [9]. Cheung [14] has proposed a user-oriented software reliability model, which measures the quality of service that a program provides to a user. His Markov reliability model uses a program flow graph to represent the structure of the system. The flow graph structure is obtained by analyzing the code. It uses the functional modules as the basic components whose reliabilities can be independently measured. It uses branching and function-calling characteristics among the modules, that are measured in the operational environment. Similar structural models have been proposed by Littlewood [15] and Booth [71], to analyze the failure rates of a program. Lyu [66] proposed a structural model for estimating the reliability of component-based programs where the software components are heterogeneous and the transfer of control between components follows a discrete time Markov process. It is assumed that time spent in each state is exponentially distributed.

In a path-based analysis, reliability of each path from singly entry node to single exit node in a control flow graph is computed and average of path reliability is computed for reliability estimation of whole system. Sometimes path-based analysis gives incorrect result due to infinite paths caused by loop. This problem is solved to some extent by Krishnamurty et al. [68]. They have proposed a two phase approach. In the first phase, the reliability for each component was predicted based on code coverage of that component. In the second phase the reliability for the whole system was obtained by integrating the reliability of components achieved in the first phase. They have resolved the problem of *intra component dependency* due to loop. Multiple executions of the same component in a loop were collapsed into k occurrences, where k is defined as the degree of independence(DOI). The major problem in reliability analysis is that it is not properly measured how the fault detection and removal at the component level influence the reliability of whole system. They mentioned that the reliability of a component is predicted based on code coverage of that component.

The development of a probabilistic technique for reliability analysis that is applicable at the analysis and design-level is cost effective. It saves the effort at the actual development and system integration phases. Cortellessa et.al. [69] have proposed an early estimation of time distribution for components from UML model. Their approach on system reliability prediction is based on component and connector failure rates. Three different types of UML diagrams: Use Case, Sequence and Deployment diagrams are used for reliability analysis. For estimating the time spent in each component, they have counted the number of times a class is busy in a scenario. Both component failure and connector failure probabilities are considered.

Similar to this, Yacoub et al. [70] also proposed a path-based approach to get the early reliability of a system at the analysis phase. They proposed an algorithm named Scenario-Based Reliability Analysis (SBRA). SBRA is used to identify critical components and critical component interfaces, and to investigate the sensitivity of the application reliability to changes in the reliabilities of components and their interfaces. The technique is suitable for systems whose analysis is based on valid scenarios with timed sequence diagrams. The execution profiles of these scenarios are assumed to be available. Component Dependency Graphs (CDG) are derived to establish a probabilistic model upon which the reliability analysis technique is based. Time spent in each component is based on execution probability of each scenario and execution time of the component within that scenario. It is assumed that the reliability of each component is already given. For estimating the reliability of an individual component, Lyu et al. [57] stated that *complexity metric* should be a parameter. It helps to estimate the failure rate for initial software fault density. The advantage of these early reliability estimation techniques [69, 70] is that it can extract valuable components at the analysis phase.

3.4 Early Test Effort Estimation Methods based on Use Cases

The work on early effort estimation based on use cases was first proposed by Karner [19]. He defined a metric called Use Case Point (UCP) based on use cases to estimate the effort of an application. From that day onwards, a continuous research is going on UML based effort estimation [19,72,73]. A number of technical complexity factors such as distributed system, response, end-user efficiency, easy to install, easy to use etc. are considered along with some environmental factors to adjust the *Use Case Point*(UCP). There is a mapping from use case to test case generation. Currently, the number of test cases for a use case is estimated based on UCP.

In the above discussed work [19, 72, 73], though the complexity of a use case is considered as a major attribute for effort estimation both for development and testing, the complexity is roughly categorized as simple, average or complex based on the number of transactions or number of scenarios only. As the factor "complexity" plays a major role in estimating the fault proneness of a system, it is directly related to testing and development effort. Hence, the architectural details of a use case should be analyzed for getting the complexity in a quantitative form. Another limitation with these existing work is that these estimation techniques contain a lot of involvement of subjective measures from domain experts and hence, the accuracy of these approaches are doubtful.

Kim et al. [74] could be able to solve these limitations to some extent by proposing an effort estimation approach, in which the UML model is analyzed to get the complexity of a system. It collects data at the analysis stage and considers the use-case diagram, class diagram, interaction & state diagrams. The project effort is estimated based on UML points, where UML points are calculated through UCPs and Class Points (CP). Inheritance, uses, realize relationships of use-case diagrams and number of parameters, number of classes of class diagrams (To estimate CPs) are used to get the structural complexity of a system in whole. The accuracy level of their approach is less as it is based on a lot of subjective matters.

Robiolo and Orocosco [75] proposed an effort estimation method through use-case diagram. In their approach, the size of a project is estimated based on two factors: total number of use-case transactions and total number of entity objects. Finally the effort is estimated through mean productivity value. First a use-case is converted to a textual description and then, basic elements such as function and data are identified. Size of the application is estimated based on the number of module entity objects. They have not considered the architectural details of a use-case and could not estimate the effort required for an individual function.

Similar to the discussed UCP-based testing effort estimation method, Zhu et al. [76] proposed a method to predict the number of test cases for the system from use cases. They considered number of transactions, number of entity objects and some special requirements which are not covered by transactions to estimate the number of test cases for a use case. These effort estimation methods only considered the estimation of high level effort. Furthermore, these testing effort estimation methods [16, 17, 77] estimate test cases for the whole system not for each individual task unit(use case). These are too abstract for estimation.

3.5 Risk Analysis for Testing

Amland [78] has proposed a risk-based testing for large projects. The risk factor of a function is calculated based on probability of failure and cost of failure in the function. The probability of failure is decided based on four parameters such as new functionality, design quality, size and complexity. The cost of failure is decided based on both supplier cost and customer cost. For all indicators, only three values are used: low (1), medium (2) and high (3). The limitation of this approach is that the complexity assessment is error-prone as it is decided in an informal way based on subjective judgment of domain experts.

Some risk assessment methods [79, 80] have been proposed at the requirement stage based on multiple experts knowledge. These two methods first identify possible mode of failures for a high-level requirement and then try to estimate the impact of these failures on the requirement. Unlike our approach, these risk assessment methodologies do not take any architectural level information and therefore are purely subjective. Hence, these methods are more error-prone due to only human intensive.

Some techniques are available for reliability-based risk assessment based on formal design model [81, 82]. Yacoub and Ammar [81] first proposed a risk assessment method at the architectural level using UML models. They have proposed heuristic risk factor associated with a component and with a connector based on dynamic metrics (dynamic complexity and dynamic coupling metrics to estimate the complexity factor of a component and a connector). Then, the risk factor of the system is assessed based on two inputs: abstract intermediate representation of the system called

Component Dependence Graph (CDG) [70] and risk factor of individual components.

Goseva-Popstojanova et al. [82] have also proposed a similar approach for risk assessment. They have estimated the risk factor for a scenario by the help of component and connector risk factors and *Discrete Time Markov Chain* (DTMC) with a transition probability matrix $P^x = |p_{ij}|^x$, where $|p_{ij}|^x$ is the conditional probability that the program will next execute component j, given that it has just completed the execution of component i. They also introduced multi-failure states that represent failure modes with different severities. These two approaches [81, 82] are purely analytical and do not take any input from domain experts.

Appukkuty et. al [83] have proposed a risk assessment method by considering possible failure modes of a scenario and computing the complexity of the scenario in each failure mode. Their proposed method is for risk assessment at the requirement level. Similar to our approach, Cortessela et al. [84] have proposed a risk assessment method based on UML models, but their method is assessing performance-based risk, whereas our method is assessing reliability-based risk from UML models.

Smidts et al. [85] have added *safety* as a characteristic for reliability estimation and define the software reliability is the probability that the software-based digital system will successfully perform its intended *safety* function, for all conditions under which it is expected to respond, upon demand, and with no unintended functions that might affect system safety. They have collected various software engineering measures at different stages of the software development life cycle and proposed a number of methods to estimate the reliability for safety critical digital systems at different phases of the life cycle.

3.6 Summary

We have discussed the work on code prioritization to improve the coverage-based testing within the available test resources. We have also presented the recently reported literature on identification of fault-prone components at the code level and architectural level. We have discussed various reliability analysis techniques both at the code level and architectural level. Finally, we have discussed some existing risk analysis techniques in which the components, scenarios and use cases are ranked relative to their estimated risks.

Chapter 4

Prioritizing Source Code for Testing

A moderate size application generally consists of a number of components. The components interact among themselves through a number of operations. It is not always feasible to test all components and operations thoroughly within the available test budget. Prior work has shown that often, a small number of bugs within a system account for the majority of the reported failures; and often, most of the bugs are found in a small portion of the source code of a system. However, exactly identifying those parts is a big challenge.

The user's view on the reliability of a system is improved, when the occurrence of bugs are reduced from the frequently executed parts of the software [9, 13, 86, 87]. According to Musa [9], removing faults from the frequently executed parts of the source code helps the test manager to achieve high reliability with low cost. However, the length of time a part of the source code is executed does not wholly determine the importance of the part in the perceived reliability of the system. It is possible that the result produced by an element[1] which is executed only for a small duration is saved and extensively used by many other elements. Sometimes, the produced result of a rarely executed element is saved and widely used by a number of frequently executed elements. Hence, an element on which many number of other elements are dependent would have a high impact on the reliability of the system, even though it is itself getting executed only for a small duration.

The degree of coupling is correlated with the criticality of a system [88]. An element[2] which provides a number of services is reusable as it is independent. An

[1]An element may be as small as a statement and as elaborate as a class or couple of related classes.

[2]We have considered a class as an element throughout our thesis. Class and element are written interchangeably.

element importing services may be difficult to reuse in another context because it depends on many other elements. Coupling is also related to change proneness [89]. An element which is providing services to many number of other elements is likely to change, because it has to adjust to the evolving needs of the dependent elements [88]. The same element is also reusable and changeable[3]. So, extra test resources is required for the element, which is providing services to many number of elements because, bugs in that may be infected highly.

Assuming that all elements are approximately of similar size and complexity, the failure rate of a software product would be disproportionately influenced by the presence or absence of bugs in some elements. These elements either get executed frequently than others during the normal operation of the software or the results produced by these are used extensively by a large number of other elements. Hence, we estimate the criticality of an element on the basis of its two important characteristics such as execution probability and influence toward system failures. The first one is determined through *operational profile* [9] of the system and the later one by the help of *coupling* [90].

We introduce a new metric called *Influence Metric* for an element within a system. It shows the number of elements in the system that are using the produced result of the given element, directly or indirectly. Our proposed Influence Metric provides the detailed information at the statement level by marking the statements within a program that are influenced by a given element. The Influence Metric that generates $influence\_value$ for an element is used as the measure for criticality computation. As the analysis is performed at code level, our proposed method marks the nodes (statements) in the source code that are dependent on a given element, directly or indirectly. First, we propose an algorithm to compute the influence_value of a method and then, we use it to compute the influence_value of a class. We compute the criticality of a class on the basis of its $influence\_value$, which shows how many nodes are dependent on it and *average execution time*, which shows how often these dependencies are executed at run time. We prioritize the elements within a system according to their estimated criticality. First, prioritizing the elements within a system and then conducting testing, will promote efficient testing of software by revealing important bugs at the early phase of testing.

The rest of the chapter is organized as follows: Our proposed Influence Metric is

[3]The element which is depending on a number of elements is also changeable due to change in determinate elements.

discussed in Section 4.1. In the section, we discuss the test priority assignment using Influence Metric. The experimental results are given in Section 4.2 and the chapter summary is given in Section 4.3.

4.1 Our Approach

An object-oriented program comprised of a set of classes. A class consists of number of methods. We have proposed an algorithm named *MethodInfluence* by using forward slicing approach [25] to compute the influence_value of a method within a system. Influence_value of an element shows the influence of the element towards system failure. We have taken the intermediate representation of the program called *Extended System Dependence Graph (ESDG)* [28] as an input to our algorithm, *MethodInfluence*. Our algorithm is applied on each method-entry vertex v of a class. The algorithm marks the vertices of ESDG that are dependent on v, directly or indirectly. We get the influence_set(m) for a method that contains the set of vertices that are using the results produced by the method m. Combination of influence_set of all relevant methods of a class is the influence_set for the class. From the influence_set of the class, we get its influence_value. This approach statically computes the influence of a class within a whole program. Execution of the program is not necessary. Though, the influence_set of a class shows all possible requests to the class for service, but it is unable to show how often these requests are executed in the operational environment.

The reliability of a system is not related to the number of existing faults in the system under test. It is only related to the probability that a fault leads to a failure that occurs during software execution [14]. It is because, the data input supplied by the user decides which parts of the source code will be executed. A bug existing in the non-executed parts will not affect the output. So, it is not sufficient for a class to know how many other classes are requesting services from that class. It is also required to know how often these requests are executed at the run time. For this, we are extracting the average execution time of a class within a system. It is obtained through the *operational profile* of the system. Operational profile is the probability with which different high-level functions (or use cases) are executed during a typical use of a software. Once both the factors of a class within a system, *influence_value* and *average execution time* are obtained, we compute the criticality for the class within the system. Test Priority (TP) is assigned to a class according to its criticality. TP of an element shows its intensity of testing requirements. Higher is the TP value of an element, more is the test resource required to reduce the system failure rate.

4.1.1 Influence of a method

First, we represent the input program by an intermediate representation called ESDG. Then, we apply our proposed algorithm on the ESDG to compute the influence of a method in the program. Our algorithm counts the number of nodes marked as influenced by a method m in a program from the data dependent set of that method's formal parameter-out nodes.

The influence_value of a method m is expressed as:

$$influence\_value(m) = \frac{\text{\#nodes influenced in ESDG by } m}{\text{Total \# nodes in ESDG}} \qquad (4.1)$$

In this section, we present our algorithm *MethodInfluence* in pseudo code form to compute the *influence_value* of a method. The notations used in our algorithm are presented below.

visited[i]: It is a Boolean variable which is set to TRUE upon visiting node i.

$influence[i]$: It is a Boolean variable which is set to TRUE when node i is marked as influenced.

*queue*1: It is a queue that contains the nodes which are to be processed next.

*queue*2: It is a queue that contains the nodes which are to be marked as influenced.

insertQueue: It is a function that adds nodes to a queue.

deleteQueue: It is a function that deletes nodes from a queue.

Type(n): It is a function that returns the type of node n out of all possible types in ESDG. The algorithm maintains two queues, *queue*1 and *queue*2. *queue*1 maintains the node that are to be traversed next. It contains the nodes that are in the end of control dependence edges, data dependence edges or parameter-in edges of the visiting node. *queue*2 maintains the nodes that are to be marked as influenced. It contains the nodes that are in the end of parameter-out edges of the visiting node.

Working of the Algorithm

To illustrate how to compute the static influence of a method within a program, we consider the program and its ESDG shown in Figure 2.3a and Figure 2.3b respectively. Consider the method *add* of class *Task* in the program shown in Figure 2.3a. Now, our proposed algorithm starts execution from the method-entry vertex *2*. Our algorithm

Algorithm 1 MethodInfluence(ESDG,V_{me})

```
Require: $ESDG$: Intermediate representation of the program
    $N$: Total number of nodes in $ESDG$
    $V_{me}$: Method-entry vertex of method $M_i$
1: return $Inf(M_i)$: Influence_value of the method $M_i$.
2: for $i \leftarrow 1$ to $N$ do
3:     $visited[i] \leftarrow$ FALSE
4:     $influence[i] \leftarrow$ FALSE
5: end for
6: Queue $queue1 \leftarrow \emptyset$
7: Queue $queue2 \leftarrow \emptyset$
8: insertQueue($queue1$,$V_{me}$)
9: while $queue1 \neq \emptyset$ do
10:    $n \leftarrow$ deleteQueue($queue1$)
11:    if Type($n$)==*method-entry vertex* then
12:        Traverse only its control-edges and parameter-edges
13:    else
14:        if Type($n$)==*call vertex* then
15:            Traverse its adjacent nodes
16:        end if
17:    else
18:        if Type($n$)==*polymorphic vertex* then
19:            Traverse only its polymorphic-edges
               {each adjacent node of a polymorphic-edge is a method-entry vertex}
20:        end if
21:    else
22:        Traverse only its outgoing data dependence edges and control dependence
           edges.
23:    end if
24:    for each adjacent not-visited node $w$ do
25:        $visited[w] \leftarrow$ TRUE
26:        if Type($w$)==*parameter-out vertex* then
27:            insertQueue($queue2$, $w$)
28:        else
29:            insertQueue($queue1$, $w$)
30:        end if
31:    end for
32: end while
33: while $queue2 \neq \emptyset$ do
34:    $n \leftarrow$ deleteQueue($queue2$).
35:    $influence[n] \leftarrow$ TRUE and add node $n$ to influence_set of the input method.
36:    Traverse the adjacent nodes of $n$ through its all types of edges except the
       control dependence edges
37:    for each not-visited node $w$ do
38:        $visited[w] \leftarrow$ TRUE
39:        insertQueue($queue2$, $w$)
40:    end for
41: end while
42: Calculate $Inf(M_i)$ using expression (1)
43: return $Inf(M_i)$
```

traverses each control dependence edge from the given method-entry vertex and adds the nodes $F1\_in$**,** $F2\_in$**, 3** in *queue1* and F_out in *queue2*. Now, the algorithm will delete the first element $F1\_in$ from *queue1* and checks all its outgoing edges to find any depending node not traversed. Then, it will delete $F2\_in$ and **3** from *queue1*. The process is continued till *queue1* becomes empty. Once, *queue1* becomes empty, our algorithm will start deletion from *queue2*. It deletes the node F_out from *queue2*, marks it as influenced and traverse all the *parameter-out* edges of the node only. Then, the algorithm adds all the not-visited nodes in *queue2*. Now, *queue2* contains the node A_out. Deleting A_out and marking it as influenced next, *queue2* will contain nodes 10, $A1\_in$ and 12. In the similar way, other relevant nodes are inserted in *queue2* and deleted from it. At the end, the nodes F_out, A_out, $A1\_in$, 10 and 12 are marked as influenced. It shows the contribution of *add* method to the rest of the source code.

Complexity Analysis

If N number of nodes are created in the intermediate graph ESDG for representing the object-oriented program, at each node there can be maximum $N - 1$ number of edges.
So, the worst case space complexity will be $N \times (N - 1) = O(N^2)$.
Similarly, in the ESDG any edge is visited at most once. So, the time complexity= $O(E)$, where E is the total number of edges.

4.1.2 Influence of a class

The nodes in the set $influence\_set(c)$ for the class c is the union of all the sets $influence\_set(m_i)$, where m_i is the i-th method of class c.

$$influence\_set(c) = \cup_{i=1}^{k} influence\_set(m_i)$$

where, k is the total number of methods in class c. From the influence_set, we get influence_value by applying Equation 4.1.

4.1.3 Average execution time of a class within a system

A scenario within a system is implemented by the interaction among a set of classes. The average execution time of a class c_i, denoted as $ET(c_i)$, in the system is given by:

$$ET(c_i) = \sum_{j=1}^{nos} p(j) * (Time(c_i)^j),$$

where nos is the total number of scenarios in a system under test, $p(j)$ is the probability of the execution of j-th scenario within a system and $Time(c_i)^j$ is the total activation time of class c_i within j-th scenario.

4.1.4 Computation of criticality

Test effort is assigned to a class according to its criticality. We combine both *influence_value* and *average execution time* of a class to get the criticality of that class. *Criticality* for a class is computed by applying the following formula.

$$Criticality(c_i) = Influence\_value(c_i) \times ET(c_i) \tag{4.2}$$

where, $influence\_val(c_i)$ is the estimated influence of class c_i towards system failures and $ET(c_i)$ is the *average execution time* for class c_i within a system. *Test Priority (TP)* is assigned to a class according to its criticality. A class with high TP is critical and hence, requires extra test effort.

TP for various classes of a small program is computed using Equation 4.2 and is shown in Table 4.1.

Table 4.1: Test Priority Calculation

c_i	$Influence\_value$	$ET(c_i)$	Total TP	$TP\%$
1	22	55	1210	22
2	12	75	900	16
3	15	38	570	11
4	45	58	2610	49
5	06	15	90	02
Sum	100	-	5380	100

4.2 Experimental Studies

We have implemented *MethodInfluence* algorithm for the calculation of Influence Metric for Java programs. The intermediate graph, ESDG, used in the algorithm is obtained using ANTLR in the ECLIPSE framework. We have considered three case studies- *Library Management System (LMS)*, *Super Market Automation System (SMA)* and *Automatic Teller Machine (ATM)* throughout our thesis. The case study, LMS, basically provides the facilities such as login, register, add/remove title, add/remove book, search/issue book, return book and collect fine etc. The case study, SMA, is implemented in a large supermarket that provides a number of services such as find product, specify the required quantity, specify fulfillment of the product, record customer details, take payment, conform order and print invoice and picking. These are well explained in [2]. Our third case study, ATM, is an electronic banking outlet, which allows customers to complete basic transactions without the aid of a branch representative or teller. This is an example of a commercial application system.

We present a brief summary of these case studies in Table 4.2, so that the size of each can be well understood. In Table 4.2, *Object-point* shown in Column 6 is estimated based on a number of factors such as how many individual *screens* are displayed, how many *reports* are produced and how many *3GL modules* are developed in the system etc. [91, 92]. Classes shown in Column 5 represent the number of user classes. System classes are not considered here. These case studies are neither very small nor very large, but of moderate size. For a better understanding of the above case studies, use case diagrams of the case studies are shown in Figure 4.1.

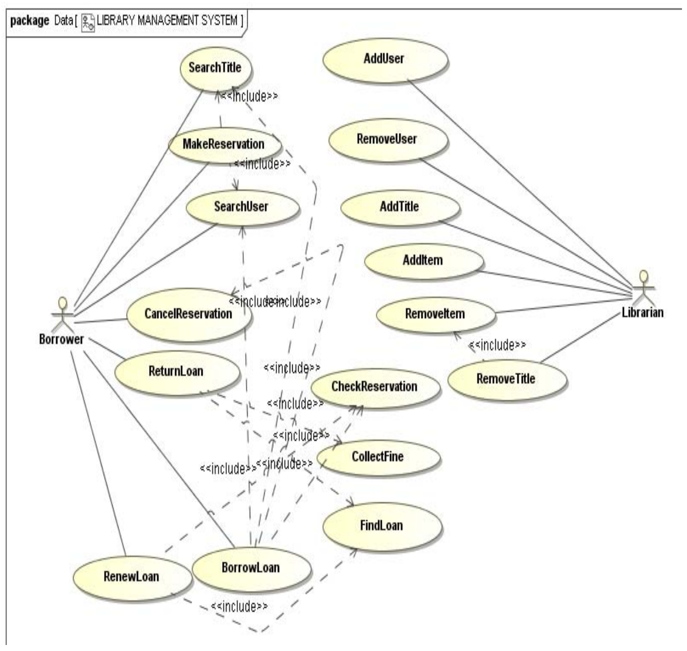
Table 4.2: Brief summary of our case studies

$System$	$LOCs$	$UseCases\#$	$Scenarios\#$	$Classes\#$	$Object-points\#$
LMS	2486	16	56	18	153
SMA	1137	09	23	10	31
ATM	4217	12	30	22	82

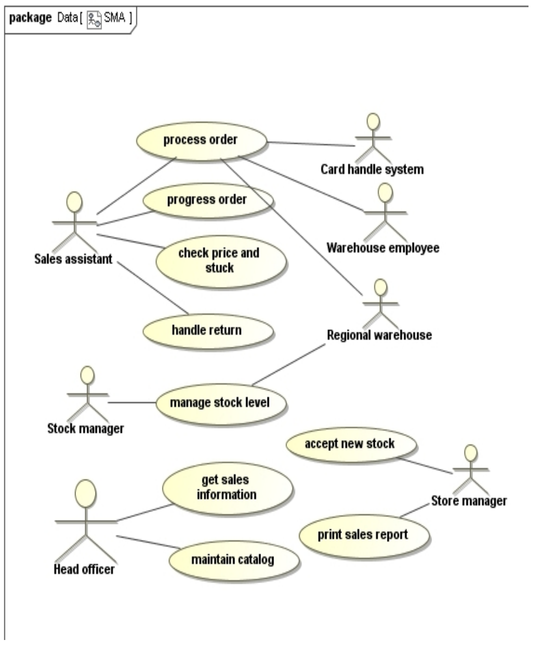
We have conducted a number of experiments to examine the sensitivity of various classes toward system failures in Section 4.2.1. We have also conducted a number of experiments to check the effectiveness of our approach compared to an existing approach in Section 4.2.2.

4.2.1 Sensitivity analysis

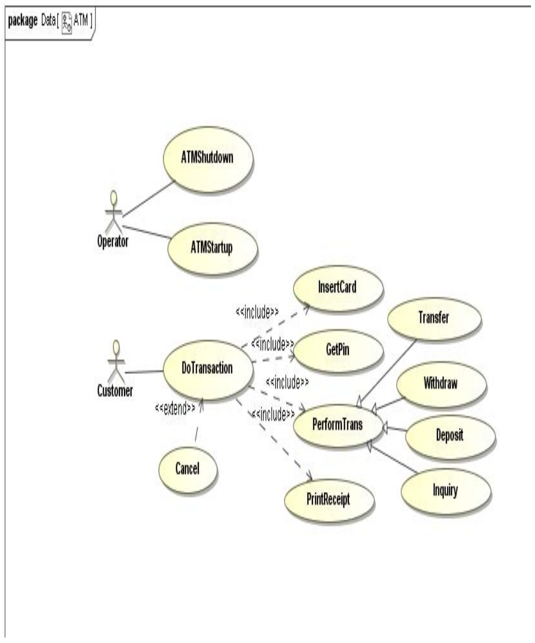
Using our criticality estimation method, we have investigated the failure rate of an application based on the failure of individual classes with different criticality. We have done it in three phases. In the first phase, we selected the highest priority class



(a) Use case diagram for LMS



(b) Use case diagram for SMA



(c) Use case diagram for ATM

Figure 4.1: **Use case diagrams of the case studies**

from a case study and decreased its reliability[4], while fixing the reliabilities of other classes to 1.0. To observe the failure rate of the application, we selected randomly *100* numbers of test cases (randomly selected scenarios) based on operational profile. A

[4]Techniques for class reliability estimation is a step wise procedure that includes fault injection, testing and retrospective analysis. We are assuming an estimate is available, this is used as a parameter for observing the failure rate to analyze the sensitivity of the application.

test case is responsible for the execution of one scenario[5]. We continued our process by slowly decreasing the reliability of a selected class in a step wise manner and observed the failure rate of the system under test at each reliability point of that class for the same set of test cases. Same process and same test cases were also applied to a class with medium priority and the class with the lowest priority. As the observed failure rates[6] were varied for the same set of test cases, at each reliability point of selected classes (one at a time), we could analyze the sensitivity of a class towards system failure rate. The graphs shown in Figure 4.2 show the failure rates of LMS, SMA and ATM case studies. We obtained the graphs by decreasing the reliability of the highest priority class, some medium priority classes and the lowest priority class (one at a time) of each case study, in a step wise manner. We have considered six classes of each case study including the highest and lowest priority class.

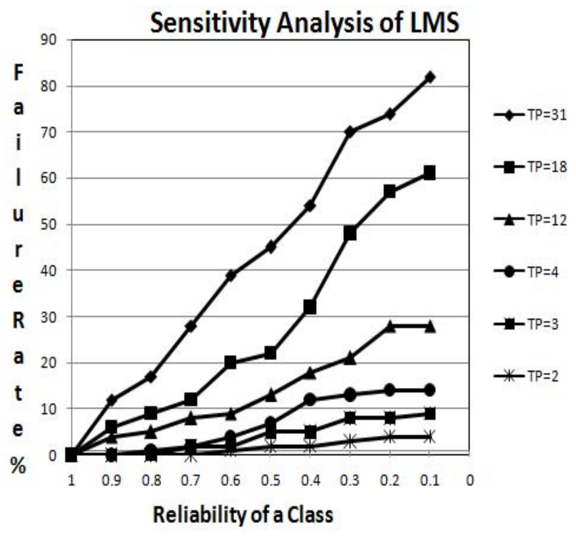
In Figure 4.2, it is clearly shown that, when the reliability decreases for a class with high TP value, the system failure rate increases at a higher rate, but this is not true for a class with low TP value.

4.2.2 Comparison with Musa's approach

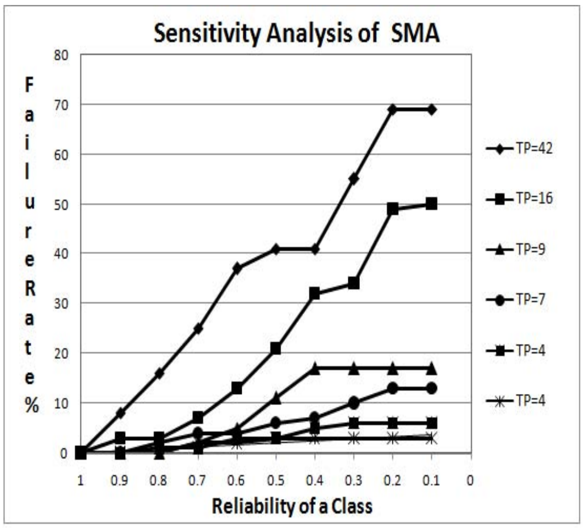
We have argued that the classes with high tendency towards system failures are not only identified by their execution time but also by their influence_values. Like *average execution time*, a class with high *influence_value* is also responsible for a high failure rate of the overall system. To validate our claim, we have conducted two experiments on each case study (LMS, SMA and ATM). In the first experiment, Experiment 1, we checked the impact of *execution time* on system failure rate and in the second experiment, Experiment 2, we checked the impact of *influence_value* on system failure rate.

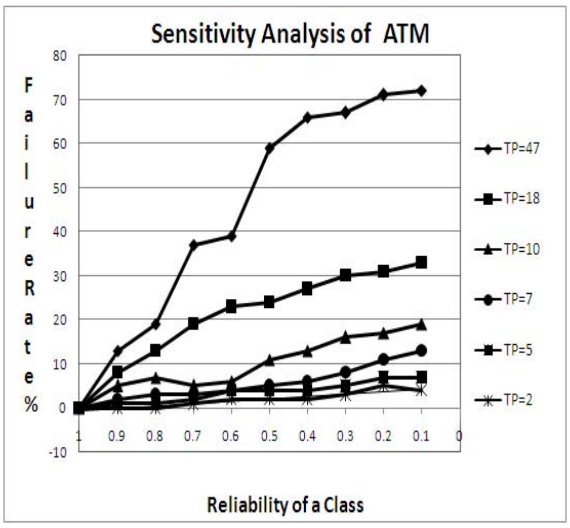
Experiment 1: (Extended Musa's Approach)
We have extended the existing Musa's approach [9] to class level, for sensitivity analysis. For each case study, we have taken first five classes from a set of classes arranged in descending order according to their *average execution times.* In this experiment, we have ignored the *influence_value* of a class. Then, we applied the same technique and the same data set to the selected five classes as discussed in Section 4.2.1.

[5]A scenario may be executed a number of times for different test values.
[6]$Failure\ rate = number\ of\ test\ cases\ failed \div number\ of\ test\ cases\ executed.$



(a) LMS





(b) SMA

(c) ATM

Figure 4.2: **Failure rate of an application based on class reliabilities (one at a time)**

Experiment 2: (Checking the impact of newly introduced factor: *influence_value*)
In this experiment, our aim is to prove that a rarely executed class is also responsible for increasing the system failure rate, if a number of classes are dependent on it, directly or indirectly. Hence, we check the impact of *influence_value* of a class on system failure rate. For conducting sensitivity analysis, we have selected five classes with low *average execution time* and high *influence_value* from the case studies, LMS, SMA and ATM. We have applied the same technique and same data set as in *Experiment* 1. We checked the tendency of each selected class towards the overall failure of the application. For simplicity, we have considered only five classes in both the experiments.

Result Analysis and Discussion

For LMS, SMA and ATM case studies, the failure rate of the overall system was 58% and 47% and 72% respectively in the first experiment, Experiment 1, when the reliability of the first class, the class with the highest execution rate, was decreased from 1 to 0.5. In the second experiment, we found that the system failure rate was near about 54%, 51% and 69% for LMS, SMA and ATM case studies respectively, when the reliability of some classes out of the selected five classes (one at a time) was decreased from 1 to 0.5. We found that the overall failure rate of ATM case study was the highest in both the experiments, Experiment 1 and Experiment 2. In the case study ATM, we found that the class Withdrawal has the highest execution rate and also has high influence_value. So, the failure rate was increased in a high rate, when the reliability of Withdrawal class was decreased.

From Experiment 1, we observed that the failure rate of a system was increased, when the reliability decreased for a class with high *average execution time*. From Experiment 2, we observed that a class with low *average execution time* but high *influence_value* was also responsible for increasing the failure rate of the system. From both the experiments, we concluded that the newly introduced factor *influence_value* in our proposed method is also playing a major role in identifying the failure-prone classes whereas, Musa [9] stated that only the frequently executed classes should get extra test resources on the testing phase as they are more failure-prone. As our approach considers both the factors: *average execution time* and *influence_value*, it exposes the failure-prone classes that are exposed by Musa's approach [9]. Further, our method identifies new failure-prone classes through the newly introduced factor *influence_value* that are neglected by Musa's approach due to low execution time. It is because, some wrongly produced output by a rarely executed class may be used by some frequently executed classes, that makes the failure rate of the system high.

4.2.3 Threats to validity of results

In order to justify the validity of the results of our experimental studies, we identified the following list of threats:

- Biased test set design and influencing results.
- Seeding biased errors in various classes of each case study.
- Testing only for selected failures and loosing generality of results.

- Using testing methods which may only be suitable for some particular bugs while may not reveal other common and frequent bugs.

Measures taken to overcome the threats

In order to overcome the above mentioned threats and validate the results for most common and real life cases, we have taken the following corrective measures:

- We used same test set in each reliability point of a class for observing failures.
- We used same type of seeded bugs in the classes of each case study.
- We took care that the seeded bugs match with commonly occurring bugs.
- We inserted class mutation operators to seed bugs. Using mutation operators, we can ensure that a wide variety of faults are systematically inserted in a somewhat impartial and random fashion. While traditional mutation operators are restricted to a unit level, class mutation operators [93] for object-oriented programs have impact on cluster level.
- We considered the failures that provide a base to the user to decide how much they can trust the software.

4.2.4 Limitation of our approach

It is not sufficient to assign test priority to an element on the basis of its influence_value and its average execution time. A single bug in a class with low test priority value may cause catastrophic failure. As, some classes usually provide exception handling of rare but critical conditions, it is necessary to consider the severity associated with each class by checking the effect of its failure to the system operation. For efficient testing, the test priority computation should also include the severity associated with the failure of a class. The limitation of our approach is that we have not considered the severity associated with each class.

Another limitation is that though ESDG is simple for representing small and moderate programs, but for a large real life program, ESDG may become too large and complex to manage [26]. Obviously, the storage requirement will also be very high. For large programs, $influence\_value$ of a method may be computed by using traditional *fan-in* and *fan-out* metrics [94] in place of ESDG. However, the advantage of using ESDG over the traditional fan-in and fan-out method is that our proposed influence metric will improve the accuracy. It is because, ESDG shows the details regarding the statements that are really affected in the source code, when a method is producing incorrect result. It is because, ESDG shows the dependencies at statement level, whereas fan-in and fan-out show the higher level dependencies at function level/module level.

4.3 Summary

We have proposed a new metric called *Influence Metric* to identify the criticality of an element in the source code. It is based on static analysis of the source code. The average execution time of a component within a system was estimated based on the *operational profile* of the system. Criticality for a component within a system is computed on the basis of its *influence_value* and *average execution time.* Test priority is assigned to the components according to their criticality.

We have experimentally proved that decreasing the reliability of a high priority class drastically increases the failure rate of the application, whereas, it is not true in case of a low priority class. So, the intensity with which each element should be tested is proportionate to its test priority value. It helps the test manager to expose the critical elements before test case generation that are getting less attention in terms of testing. The limitation with our criticality estimation method is that it does not consider any external factor. Our proposed test effort prioritization method will be effective, if the severity associated with various failure modes of an element could be considered. So, we aim at considering the severity in our next work.

Chapter 5

Criticality Estimation

First, prioritizing the program elements within a system according to their criticality and then, conducting the testing process will promote efficient testing of a software product by revealing important bugs at the early phase of testing. In Chapter 4, we have considered two factors of a component[1] (i) *influence_value* and (ii) *average execution time* and computed the criticality of the component within a software system. We have experimentally proved that when the reliability decreases for a high critical component, system failure rate increases at a higher rate whereas, this is not true for a low critical component. Though, the failure rate of a system is heavily influenced by the influence_value of a component, but it is not a sufficient factor for criticality estimation. In a real life application, each failure does not carry equal weight. Our aim is not only to improve the reliability of a system, but also to minimize the post-release failures that have a negative impact on the user or on the system. For this, we consider the impact of a failure within a scenario as another factor for criticality computation. We consider two important external factors such as the severity of failures and the business value associated with a component for criticality computation. Severity checks the impact of failure of a component within a system. The aim of consideration of business value is that a technical staff cannot guess which high level functions are important to the customer and also a customer cannot estimate the cost and technical difficulties in implementing a specific high level function. The adoption of Value-based testing [21, 42] increases the return on investment on testing.

[1]In an object-oriented program, a component may be a class which is the smallest executable unit or it may be elaborated as a collection of classes in a package. However, a component may be a collection of many other things. For example, a mixture of some source code templates with related documents might be called as a component. After all, everything is a class in an object-oriented program, every component is a class too. In our approach, we take a class as a component and write class and component interchangeably throughout the chapter.

We compute the criticality of a component on the basis of the following factors:

1. Average execution time of a component within a system.
2. Influence_value of a component within a system.
3. Structural Complexity of a component: Response for a Class (RFC); Weighted Methods in a Class (WMC).
4. All possible types of system failures for which the component is responsible and the severity associated with each failure.
5. Business value associated with a component.

User's perception is an indicator on the acceptance of a system. User's view on the reliability of a system is improved and almost cheaper, when faults which occur in the most frequently used parts of the software are almost removed [9, 13, 86, 87]. The idea behind the consideration of average execution time for a class as a parameter is that when, a class is executed for longer time, there is a high probability that any existing errors in the class will be executed during the run. It will cause the frequent failure of the system.

In Chapter 4, we have introduced a metric called *Influence Metric* for an element, that shows the degree of influence of that element toward system failures.

The idea of including structural complexity is to estimate the probability of presence of faults within a component. The case studies discussed in [95] show that the residual bugs location is strongly correlated with module size and complexity. For evaluating the structural complexity, Chidamber and Kemerer [41] have proposed six metrics. We found that considering all the six metrics at a time is complicated, time consuming and also sometimes not useful for a particular purpose. At the same time, a single metric is also not sufficient for complexity estimation. At least the use of two or three CK metrics give a proper estimation of potential problems [58]. For our purpose, we are using two CK metrics: RFC and WMC. RFC gives an idea about the longest sequence call of methods and WMC provides the Cyclomatic complexity of each method implemented in a class. Our approach shows the complexity of a class within a program. In addition to that, it also shows the likelihood of the class to fail in the operational environment due to the consideration of the factor, *average execution time.*

There are some components which exist within a system with low complexity, but the failure of any one of those components may have a catastrophic impact on the system. For example, a critical code may be called in case of an emergency, which happens infrequently but can have catastrophic impact, if an error occurs in that part. The impact of the failure may cause severe damage to the system or a huge financial loss. So, for computing the criticality of a component, we consider the severity of the damage caused by the failure of the component within a scenario. The severity factor is dependent on the nature of the application. Hence, it is a subjective matter and is basically assessed by domain analyst, who has the knowledge of the environment in which the software will be used. The basic input for severity assessment is the costs of various failure modes. Detailed procedure of severity estimation is addressed in [96].

There is also a close relationship between testing and business value that comes from market or from customers [43]. Each use case of a system should not be treated with equal importance [97]. Once the business values for various use cases are decided through the interaction with domain experts, our approach estimates the business value for a component by checking its interaction within various use cases. Based on the Values of use cases, a component's Value is estimated.

Once the criticality of a component is estimated through our approach, exhaustive testing has to be carried out to minimize bugs in high critical components. The total test effort is distributed among various components within a system according to their criticality. The component with high criticality will get high priority for testing. As a result, not only the post-release failures will be minimized but also, the severed types of post-release failures will also be minimized within the available test budget.

The rest of the chapter is organized as follows: Section 5.1 discusses the proposed methodology for estimating the criticality of a component within a system. The experimental studies are conducted to test the effectiveness of our approach. The experimental results are shown in Section 5.2. The summary of the chapter is discussed in Section 5.3.

5.1 Our Approach

Our proposed methodology on criticality computation of a component consists of the following steps:

1. Computing the *influence_value* of a component within a system as discussed in

Chapter 4.

2. Estimating the average execution time of a component by executing the test data based on operational profile (discussed in Chapter 4).

3. Analyzing the structural complexity for a component (Section 5.1.1).

4. Analyzing the severity associated with a component through simulation runs (Section 5.1.2).

5. Estimating the business value associated with a component (Section 5.1.3).

6. Performing criticality estimation and prioritizing the components according to their criticality (Section 5.1.4).

5.1.1 Analyzing the structural complexity

Our aim is to find the complexity associated with a component by analyzing the complexity of various services provided by the component. We consider only two CK metrics (RFC and WMC), out of six metrics proposed in [41]. It is experimentally proved that a component with high RFC and high WMC is fault-prone [98]. Hence, these two chosen metrics (RFC, WMC) are used as inputs to derive the complexity of a component for our purpose. RFC contains a set of member functions directly or indirectly called by the class, whereas WMC is checking the complexity associated with all member functions of a class using Cyclomatic complexity.

RFC metric measures the cardinality of a set of methods that can potentially be executed in response to a message received by an object of that class [41]. In RFC, the basic unit is a *method*, which refers to the message passing concept in O-O programming. The RFC value for a class is given by

$$|RS| = \sum_{i=1}^{M} R_i,$$

where RS, M and R_i represent the response set for the class, number of methods in the class and the set of methods called by i-th method of the class, respectively. A class with high value of RFC indicates that the complexity of services provided by the class is high and hence, the understandability is less. When a larger number of methods are invoked from a class through messages, it complicates the testing and debugging process and also it is difficult to change a class due to the potential for a ripple effect. As testing and maintenance is complicated, the chance of getting bug increases. We have derived R_i using the intermediate representation, ESDG, of the source code. Our algorithm starts traversing from each *method-entry* vertex of a class and traverses only the *call-edges* in a forward direction and generates a set of nodes called by each method of a class. This process is repeated for each method of a class and finally, the sets are merged to get the response set, RS, for the class.

Luke [99] argued that there is really no way to know a software failure rate at any given point in time because the defects have not yet been discovered. According to his statement, the design complexity is positively linearly correlated to defect rate. Hence, the occurrence of software defects should be estimated based on McCabe's complexity value or Halstead's complexity measure [99]. We consider WMC metric that gives a rough estimation of total complexity associated with a class. WMC metric is correlated with defect rates [58]. It counts local methods and calculate the sum of the internal complexities of all local methods in a class [41]. The internal complexity of each method is decided through Cyclomatic Complexity. WMC value for a class c is given by

$$WMC = \sum_{i=1}^{M} W_i$$

where, M and W_i represents the number of methods in a class and Cyclometic complexity of i-th method, respectively. It helps to evaluate the minimum number of test cases needed for each method and hence, is used as a guideline by test manager to estimate how much time and effort is required to develop and maintain a class.

We estimate the probability of faults in a class based on two parameters: RFC and WMC. First, we assign a threshold value to each metric as defined by Rosenberg et. al. [100]. For each parameter, we use only three weights: low **(0.3)**, medium **(0.5)** and high **(1)** [100]. The assignment of points to the three weights is a rough guideline. The following threshold values are assigned to the two parameters as stated in [100].

1. Weighted Methods per Class (WMC): $\lessapprox$ 25 preferred, $\lessapprox$ 40 acceptable .

2. Response for Class (RFC): $\lessapprox$ 40 preferred, $\lessapprox$ 50 acceptable. It has been observed that very few classes with RFC over 50 exist within a system.

The complexity information for LMS and SMA case study are shown in Table 5.1 and 5.2, respectively. The classes that are within preferred (acceptable) limit are low (medium) in complexity and the classes which exceed the acceptable limits are high in complexity. Out of these two parameters, if one parameter is in low range

Table 5.1: Structural Complexity of various entity classes within LMS

Class	*WMC*	*RFC*	*Complexity*
Borrower	29(medium)	42(medium)	medium
Title	18(low)	39(low)	low
Item	29(medium)	33(low)	medium
Loan	32(medium)	52(high)	high
Reserve	15(low)	30(low)	low

Table 5.2: Structural complexity of various entity classes within SMA

Class	*WMC*	*RFC*	*Complexity*
ProductInfo	25(low)	18(low)	low
Category-Mgr	28(medium)	37(Low)	medium
OrderHandler	33(medium)	58(high)	high
InventryMgr	27(medium)	46(medium)	medium

and the other one is in medium range, we are accepting the whole complexity of the class is medium, the higher one between the two factors. The intent of computing the structural complexity of a component is to show the probability of existence of faults within the component whereas, the intent of computing the influence_value of a component is to show that how many other components will be affected by the faulty behavior of the component.

5.1.2 Severity analysis

Severity is a rating which is applied to the effect of a failure. It shows the seriousness/impact of the effects of a failure within a system. Severity of a failure within a system decides how a bug within a component affects the whole system. We have inserted some bugs in various components within a system and executed the system for some duration in the operational environment. We observed that similar types of bugs in different components cause failures with different severity. Hence, we use the severity factor of a component as a measure to the overall quality of the product. We consider that a component is critical, if the failure of the component causes severe effect on the whole system. In our proposed criticality evaluation method, our aim is to first reveal bugs from a high critical component and then, reveal bugs from a low critical component. If there is an urgency to release the system before time or the testing time is shortened due to some unavoidable circumstances then, the test manger should ensure that the bugs responsible for severe type of failures are revealed and fixed.

Though, testing focus should be given to the parts of the code that are executed frequently [9, 13, 63], however, there is also a need for severity analysis for better quality of a system. Some parts of the source code are executed in case of an emergency. Though these parts execute rarely, the existence of a bug with them may cause a severe failure. For example, let us consider a component which is providing exception handling of rare but critical conditions. In this case, the component is executed rarely. The influence of the component toward failure is low and the structural complexity of the component is also low but, a bug in that component could cause catastrophic failure. Therefore, we have included the severity of a component as an important factor for criticality computation.

We estimate the severity of a component within a system through *Failure Mode Effect Analysis (FMEA)* [101], which is a bottom-up approach. FMEA is applied to a component to get the deficiency and hidden design defects. It focuses on two points: (i) analyze the potential failure modes of a component (how a component fails) and (ii) determine the effect of the failure modes on the system as a whole (consequences of failures). For a hardware component (electrical/mechanical), the failure modes are well known, but this case is not true for a software component. Nowadays, a number of system functions are implemented on software level and hence, there is a need to apply FMEA methodology on software based systems for determining the severity factor of a component. For a hardware component, the failure modes are wear and tear of machinery, design flaws and unintentional environmental phenomena. Sometimes, the component manufacturing company discloses the possible failure modes and also the estimated frequencies of failures for their products. This is not possible in case of a software component. For a software component, the analyst decides the failure modes based on the design and development process. Software is not a physical entity, it is a logical construct. The analyst identifies the system level hazards both at the analysis, design and implementation phase and translates it into software terms. Now, we discuss about FMEA for a software system.

Software Failure Mode and Effect Analysis (SFMEA)

The detailed SFMEA focuses on the classes or modules in which several error conditions are checked. Table 5.3 shows various types of errors that may occur within a software module/class at the design or coding stage.

Ozarin [102] has discussed the advantages of performing SFMEA at various levels:

Table 5.3: Possible error conditions within a class/module

Error Condition	*Examples*
	Error in computation
Wrong Algorithm	The module may carry out estimations wrongly due to faulty requirements or wrong coding of requirements.
Calculation underflow or overflow	The algorithm may produce in a divide by zero state
	Error in data
Unacceptable data	The module may accept out of range or wrong input data, no data, wrong data type or size, or premature data; produce wrong or no output data; or both.
Input data trapped at some value	A sensor may read zero, one, or some other value.
Bulky data rates	The module may not be able to handle a vast amounts of data or many input requests simultaneously.
	Error in logic
Wrong or unpredicted commands	The module may receive improper data but continue to execute a process. It may be intended to do the proper thing under improper situation/state.
Failed to issue a command	The module may not call a routine under certain circumstance.

(i) Method-level analysis (ii) Class-level analysis (iii) Module level analysis and (iv) Package-level analysis. According to him, SFMEA process is accurate and effective at the Method-level, which is the lowest level analysis. The authors of [103] have considered that a method within a software system is equivalent to a part of hardware system in which, there is a chance of failure under certain conditions. It is because, if a method within a class does not perform according to its pre-defined specification then, there is a chance of failure of the whole system under some conditions.

At the time of testing, the debugger analyzes the root cause of a bug and extracts the method within a class and specifically the instruction within a method, which one is the source of bug. If any failure occurs at the testing phase then, significant amount of searching is conducted to find the exact faulty parts in the source code and specifically the search is conducted to find the exact faulty instructions of a method.

As the source code is available in this stage, we conduct the operation level or method level SFMEA. During the execution of a scenario, a number of objects communicate through message passing. The message passing mechanism is implemented through method calls. A method within a class may or may not has formal parameters and may or may not has return value. To identify the severity of a class within a system, we have to identify the various types of failure modes within a method of a class and also we have to estimate the severity of each failure mode by seeding some bugs, observing the failures and estimating the impact of failures. To estimate the severity level of a failure mode, we take the views of domain experts.

Method level failure modes and effect analysis

A method performing important tasks is generally viewed as an agent, which has to fulfill a contract to perform its operation. There may or may not be any formal parameter in a method and a method may or may not return any value. A method maintains some pre-conditions and post-conditions that explicitly state the agreement of a method for performing a task. A pre-condition is the entry condition to perform a task and a post-condition is a condition that must be true after the completion of the task. Similarly, a *class invariant* states some constraints that must be true for its objects, at each instance of time during the life time of an object. A method's job is divided into two parts: (i) constraint checking part and (ii) actual logic to perform a task. We assume that there is no time constraint when, a method is performing its task. In this chapter, we consider four failure modes of a method as defined in [103]. These are:

1. Pre-condition Violation Failure Modes (F_1): There are two sub-failure modes: (i) pre-condition is not satisfied but its corresponding exception is not raised, $F_{1.1}$ and (ii) pre-condition is satisfied but its corresponding exception is raised, $F_{1.2}$.

2. Parametric Failure Modes (F_2): Any failures regarding to formal parameter declaration within a method comes under this category. The constraints on parameter values are checked in this failure mode. The type of a parameter is not considered. If any parameter constraint is already stated in the pre-condition of the method then, that failure is not included under this failure mode. For any other parameter constraint, the response of the method is checked. If any alarm is raised by the method in the form of exception then, two failure modes are considered for two individual cases. (i) the constraint is false but its corresponding exception is not raised $F_{2.1}$ and (ii) the constraint is true but its corresponding exception is raised, $F_{2.2}$.

3. Method Call or Invoke Failure Modes, F_3: It consists of two sub-cases.

   (a) A method *m1* invokes method *m2* of the same class or super class then, there is a possibility of the following failure modes in the list of failure modes of *m1*, $F_{3.1}$.

   i. *m1* invokes *m2* in the wrong order (when the invocation of *m2* is condition based), $F_{3.1.1}$.

   ii. *m1* invokes *m2* by wrong parameters (when *m2* contains parameters. We consider only parameter's value not the type), $F_{3.1.2}$.

(b) A method *m1* of class *A* invokes method *m2* of class *B* then, there is a possibility of the following failure modes in the list of failure modes of *m1*, $F_{3.2}$.

   i. *m1* fails to invoke *m2* (because of lack of instance of object of class B), $F_{3.2.1}$.

   ii. *m1* invokes *m2* in wrong order (when the invocation of *m2* is condition based), $F_{3.2.2}$

   iii. *m1* invokes *m2* by wrong parameters (when *m2* contains parameters. We consider only parameter's value not the type), $F_{3.2.3}$.

4. Post-condition Violation Failure Modes, (F_4). It consists of the failures when a method is unable to satisfy the post-conditions. The failure mode checks the conditions that a method must satisfy after its execution is completed. Any violation in the postcondition indicates an error within the method whereas, any violation of a precondition indicates an error with the client: the client is using the method wrongly. In this case, the method is performing the task in invalid manner.

We first obtain the FMEA of a component within a scenario and then, we assign severity through FMEA and hazard analysis. Severity of a component within a scenario shows how its failure affects the execution of the scenario. Domain experts play a vital role in hazard analysis and estimate the severity level of a component within a scenario. We rate the severity of a component within a scenario based on the worst effect of the failure of providing services by the component within that scenario.

According to [101], severity is classified as:

1. Catastrophic: A failure may cause death or total system loss.

2. Major: A failure may cause very serious effects. The system may loose functionality, security concerns etc.

3. Marginal: A failure may cause minor injury, minor property damage, minor system damage, or delay or minor loss of production, like loosing some data.

4. Minor: Defects that can cause small or negligible consequences for the system, e.g. displaying results in some different format.

We assign severity weights of 0.25, 0.50, 0.75, and 0.95 to Minor, Marginal, Major, and Catastrophic severity classes, respectively as defined in [81, 82]. The damage may be classified to different classes as mentioned above or it may be quantified into money value, whatever the analyst feels better. For example, if a large volume data to be sent by mail are wrong, then the cost of re-mailing will be horrible.

Tables 5.4 shows a part of SFMEA at the method level for some components within withdraw scenario of ATM system. The column *Triggered Hazard* shows the occurrence of failure when an event is triggered and some action is performed. The column *Component* shows the component in which there is an occurrence of fault. In the table, failure mode is any one failure mode out of the four failure modes (F_1, F_2, F_3, F_4) discussed above. The column *Effect* shows the effect of the failure mode on the system. Severity is any one severity out of the four severities-Catastrophic, Major, Marginal and Minor- discussed above.

Table 5.4: SFMEA at method level for some components within the *Withdraw* Scenario

Triggered Hazard	Component	Failure mode	Effect	Severity
A fault in dispensing cash	CashDispenser	CashDispenser is empty but not raising any exception	Money will be deducted from the account immediately though the customer is not able to withdraw the said amount. As every transactions are maintained in the Log, the account will be updated by the banker later on.	Major
A fault in performing transaction	Withdrawal	The object of Withdrawal component fails to check the sufficiency of cash in the customer account	ATM gives out more money than it is available in the corresponding account.	Catastrophic
A fault in completing transaction	Withdrawal	The object of Withdrawal component fails to create a new receipt	Receipt will not be printed	Minor
A fault in reading menu choice from the screen	Withdrawal	Failed to call the readMenuChoice method of an object of component CustomerConsole	Transaction can't be performed.	Marginal

There can be a couple of severity level for a component within a scenario. For example, the component *Withdrawal* has two severity levels (Minor, Marginal) as shown in Table 5.4. We consider only the worst-case consequence of a failure as the severity level for the component. For the component Withdrawal, we consider the Marginal severity level within the *withdraw* scenario.

5.1.3 Business value estimation

For ATM, the main use cases are *deposit*, *withdraw*, *inquiry balance* and *transfer money*. The business value (Value) for ATM is estimated in Table 2.1. We consider only the use cases that are used by the customer. *start-up* and *shut-down* use cases are not considered as they are the basic use cases to run the system. We use that Values to estimate the Values of various components of ATM system. A component often serves many different use cases. So, we calculate the importance of a component on the basis of its involvement with various use cases. The proposed methodology consists of the following steps:

1. Constructing the Component Dependence Diagram (CDD) from the source code.

2. Extracting slices of various scenarios from the CDD and using the slices for estimating the business value for a component.

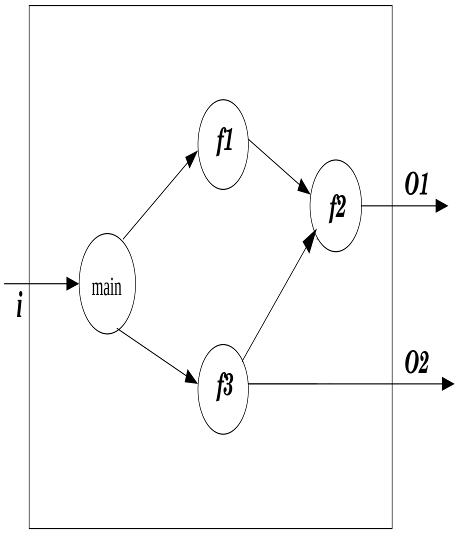
Component Dependence Diagram

A Component Dependence Diagram (CDD) is a directed graph that is used as an intermediate representation of a program. Each node of a CDD corresponds to a component of the program. A component is a basic executable unit. In a procedure-oriented program, a component can be a function whereas, it is as simple as a class in an object-oriented program. The edges of the graph represent either control dependency or data dependency among the nodes. These dependencies are represented by directed arrows. We do not use different symbols to represent these two types of dependencies since our method checks for any type of dependency. If both data and control dependencies exist between two components, we draw only one arrow between the corresponding nodes. Figure 5.1 shows an example program and its CDD. In the example program shown in Figure 5.1a, the statement *store (d)* in function $f2$ indicates saving of latest value of the variable d to a memory location. Similarly, the statement *read (d)* in function $f3$ indicates reading of the value of variable d that was last saved by function $f2$.

We view a CDD as a simplified form of a *System Dependence Graph* (SDG) [29] but, a CDD does not have as many types of edges as SDG [29]. Unlike SDG, a CDD does not represent the individual statements of a program because, inclusion of individual statements makes the graph unnecessarily complicated. In Extended Control Flow Graph (ECFG) [104], a node refers to a method of an object-oriented program whereas, in CDD, a node refers to a component. The aim of referring a node of CDD to a component instead of a method in an object-oriented program is to make the graph simple and easily understandable. We compare CDD with the *Component Dependence Graph* (CDG) proposed by Yacoub et al. [70] where, nodes refer to components. They have adapted control flow graph principle to represent the dependency between two components and possible execution paths. Unlike our approach, Yacoub et.al. [70] have considered only control dependency between components. In their approach, the components are assumed to be independent. The existence of bug in one component is not responsible for the failure of another component. As we have considered the data dependency between two components, a bug in one component may have an effect on other components.

```
int d[10];

main(){
int i;
i=readUserOption();
if(i== 1) f1();
else      f3();

}
f1(){
      .
      .
       f2();
       }

f2(){
.
.
.
store(d);
print(o1);
}
f3(){
read(d);
.
.
print(o2);
}
```

(a) An example program



(b) Component Dependence Diagram of program P

Figure 5.1: **An example program with its CDD**

The CDD, generated in our approach, satisfies all the following constraints:

- No node is isolated.

- All use cases put together cover all nodes.
- No self loops.
- The node at which a use case starts execution is not control dependent on other nodes of the graph.
- The nodes tested by any one test case are a subset of nodes belonging to slice of a scenario.

Once the intermediate graph, CDD, is constructed, we use it to extract slices with respect to various scenarios for prioritizing the components within a system.

Extracting slices of CDD with respect to various scenarios and estimating the Value of a component

Each use case has one main scenario and a number of alternative scenarios. We only consider the main scenario and do not consider the alternative scenarios. The Value of a use case is same as the Value of its main scenario.

We compute the slice S_i of the CDD with respect to scenario S_i and represent it as $Slice(CDD, S_i)$. The slice contains the set of components that are either executed during the execution of the scenario S_i or the results of the components which are saved in different variables, used during the execution of S_i.

Value estimation scheme

Once, the business values for all scenarios of a system are determined, we estimate the business value, $Value(C_i)$, for a component C_i, as follows.

$$Value(C_i) = \sum_{j=1}^{nos} q_j, \tag{5.1}$$

Where, $q_j = Value_j, \ if \ C_i \in slice(S_j) \ else, \ q_j = 0$. In Equation 5.1, nos is the number of scenarios within a system, $Value_j$ is the probability of j-th scenario and $slice(S_j)$ is the slice of the CDD with respect to j-th scenario. The priority value of a component intuitively indicates the priority of its being used during an actual operation of the program. We now algorithmically present our business value estimation scheme for various components within a system.

Value estimation Algorithm

```
1. Construct the CDD of the program.
```

2. `Determine the business value` $Value_i$ `for each scenario` S_i `within a system.`

3. `For each component` C_j `of CDD do,` $Value(C_j)$`=0.`

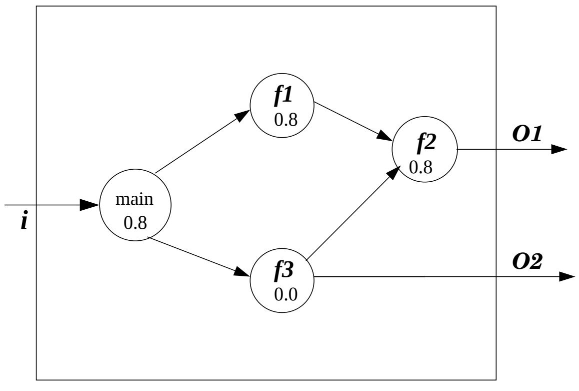
4. `For each scenario` S_i `of the program do`

   (a) `Compute the slice of scenario` S_i `from the CDD.`

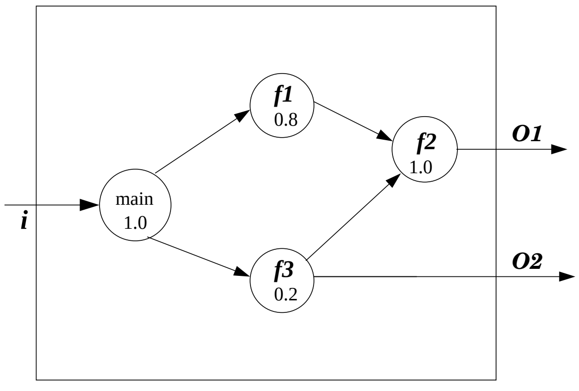
   (b) `for each component` C_j `in` $slice(S_i)$ `do,` $Value(C_j)$ `=` $Value(C_j)$ `+` $Value_i$`.`

5. `Print the Value computed for each component.`

We now explain our Value estimation method using a simple example. Let us assume that the program shown in Figure 5.1a has two use cases: U_1 and U_2. Each use case has only one scenario. Let, the business values associated with scenario S_1 and S_2 be 0.8 and 0.2, respectively. Figure 5.2a shows the Values obtained in various functions after $Slice_1$, the slice of CDD with respect to scenario S_1. Each component that is executed within the slice $Slice_1$ is getting marked by 0.8, which is the Value of the scenario S_1. Next, the slice of CDD with respect to scenario S_2, $Slice_2$, is computed. The business values of various components after slicing the CDD with respect to both the scenarios (S_1 and S_2) are obtained, as shown in Figure 5.2b. In the said figure, the functions *main* and $f2$, have Value 1, each. This means that both the functions *main* and $f2$ are either getting executed or their results are used by both the scenarios S_1 and S_2. It may be noted that the business value of 1 for a function indicates that it is required for all scenarios of the system. Table 5.5 shows the business values associated with various components of ATM system.



(a) Values after slicing scenario S_1



(b) Values after slicing scenarios S_1 and S_2

Figure 5.2: **Priority of each component after slicing S_1 and S_2**

Table 5.5: Business values associated with various components of ATM system

Component	*Vlaue*
Session	0.93
Withdrawal	0.77
Deposit	0.69
Transfer	0.39
Inquary	0.58
Cardreader	1
EnvolopAccepter	0.69
CustomerConsole	1
Log	0.86

Table 5.6: Criticality computation for *Transfer* component of ATM system

Influence_value	*EEC*	SC	Severity	Val	Criticality
0.67	0.3	0.3	0.75	.39	2.41

5.1.4 Criticality computation

For assigning criticality, the commonly used method is to do a proper weight assignment and then, calculate a weighted sum for a class [105]. We assign a relative weight for each chosen factor of a class. For each factor, we assign equal weight. The weight may vary depending on the nature of the system. An example of criticality computation for a component is shown in Table 5.6. The headings used for different columns of the table are: EEC- Expected Execution Time, SC- Structural Complexity and Val- Business value associated with a component within a system. The estimated criticality of a component is normalized by dividing the criticality of the component with the sum of the total criticality of all components within the system.

There are a lot of technical, productivity and environmental complexity factors that exist within a system. For simplicity, we consider only five factors for complexity estimation. Consideration of a number of factors improve the accuracy of criticality estimation method but, it will make the process complicated and confusing. In Table 5.6, we assign weight of *1* to each complexity factor. It may vary from application to application and it is purely a subjective matter.

5.2 Experimental Studies

We have applied our proposed complexity estimation method and prioritized the test effort according to their estimated complexity on LMS, SMA and ATM case studies. These are implemented in Java and are introduced in Chapter 4.

First, we applied our proposed algorithm, *MethodInfluence* (discussed in Chapter 4) for each case study to get the influence of various classes within a system. Then, the average execution time of a class within a system was decided by executing the system 100 times and collecting the execution time of various classes in each run. 100 test cases were selected randomly based on operational profile data. We calculated WMC for a class manually through the source code. To get the RFC of a class, we have developed an algorithm that traverses the ESDG of the system and generates RFC for the class. The severity of a class within a system was decided by seeding different types of faults in the class and observing the system failures by executing the system a number of times in the operational environment. Finally, we have computed the criticality for each class as shown in Table 5.6 and ranked the classes according to their criticality. Once the criticality of various classes within a system are estimated, our aim is to validate our result. For this, we have conducted experiments to check how the faults in these classes are affecting the reliability of the system. The experiment is described below.

We have used fault seeding for evaluating the effectiveness of our proposed approach. It has been shown that, fault seeding is an effective practice for measuring the testing method efficiency [106]. We have carefully chosen some mutation operators to seed bugs randomly. The fault density is considered as a constant equal to 0.05 for each case study. This means that in a case study consisting of 1000 number of lines, 50 number of bugs were inserted. The seeded faults are either *class mutation operators* [93] or *interface mutation operators* [22, 107]. The class mutation operators are targeted at object-oriented specific features which Java provides such as class declaration and references, single inheritance, interface, information hiding and polymorphism. In this chapter, we have considered four class mutation operators to simulate the faults. These are:

- CRT operator-Type replacement: This operator replaces a reference type with all the compatible types (the name of other classes and interfaces) found from a cluster. There is a chance of subtle type errors by this mutant.

- CON operator- Initial states and object replacement: A Java class usually provides a number of constructors to capture the different ways of creating objects (constructor overloading). This operator replaces a constructor with other overloaded constructors. Some times, constructor of sub class may be replaced by constructor of super class by this operator. Object initialization error is related to this operator which happens frequently.

- OVM operator- Method replacement: This operator generates a mutant by deactivating the overriding method so that a reference to the overriding method actually goes to the overridden method. Actually a overriding method in a sub class has different functionality to the overridden method in a super class. So there is a chance of some semantic errors by this operator.

- AMC operator- Access mode replacement: This operator replaces a certain Java access mode with three other alternatives such as private, protected and public. For example, a field declared with a protected access mode can be mutated to private and public.

There is a number of interactions among components in an object-oriented application. Therefore, there are opportunities for integration/interface faults. Delamaro et. al. [107] have proposed Interface Mutation (IM) with the aim to test thoroughly the interactions among various units. Suppose, there are three functions f1, f2 and f3 within a system and to test the connection between f1 and f3, we insert mutants inside the component f3. In this case, these mutants may be identified through the test cases that execute calls to f3 from f2. As a result, the connection between f1-f3 cannot be tested. For this, there is a need to consider the proper place from where a function is called. Keeping this in view, we have carefully considered some *interface mutation operators* from the mutant set proposed in [107]. The IM are as follows.

1. Applying mutants within the called function: The mutants considered under this category are: *Direct Variable Replacement operator*, *Indirect Variable operator* and *Return Statement operators.*

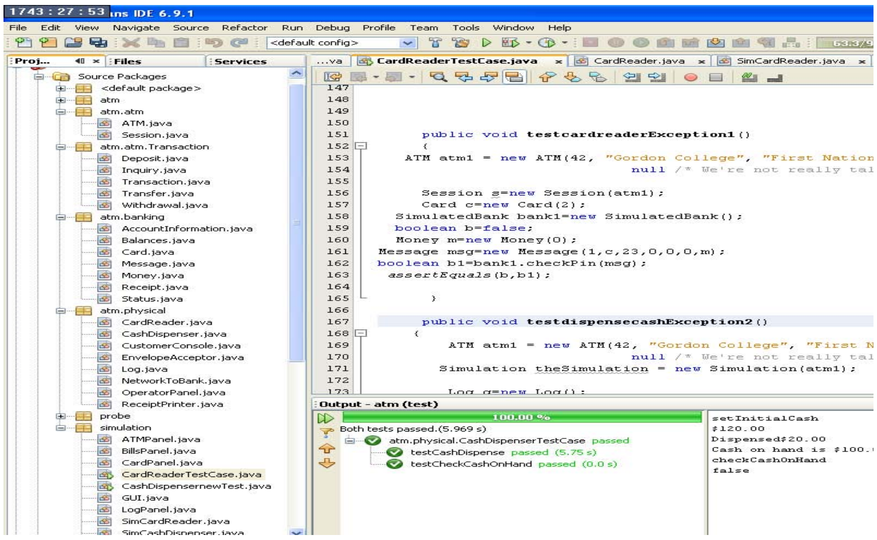
2. Applying mutants inside the calling function: It is applied to the call arguments. The mutants considered under this category are *Unary Operator Insertion* and *Function Call Deletion.* The last operator is a *missing transition.* It is not applied to the argument but to the whole function call. In a connection f1-f2, it deletes the call to the function f2. At the time of implementing the mutant inside an expression, special care is taken to replace it by an appropriate value, if the deleted function is returning any value.

First, the testing time for each case study was decided based on the number of classes, complexity of each class and number of object points [92] in the case studies.

Then, we made two copies of the source code of each case study and applied two different testing methods. In the *first testing method*, the components are prioritized according to their structural complexity [41] only, whereas in the *second testing method*, the components are prioritized according to their estimated complexity based on our proposed complexity estimation approach. The *first testing method* was applied to the *first copy* and the *second testing method* was applied to the *second copy* of each case study.

Same testing time was allocated for each copy of a case study. At this point, we emphasize the fact that our aim is not to achieve complete fault-coverage with the available test resources, but to check the efficiency of our proposed testing method. The number of test cases designed for a component at the unit level was decided according to their estimated criticality values. As a class with high $influence\_value$ provides services to others, a single bug in the class may cause interface bugs, which we cannot detect at the unit level. Interface bugs are detected at the integration level and interface testing assures that the classes have communicated correctly. So, at the integration level, we applied coupling based testing techniques [108], which is based on client-server concept. In the coupling based approach [108], when a client class calls another server class, first some method sequences of the client class are considered. These method sequences are subset of the set of method sequences decided at unit testing. Then, for each method sequence, the method sequences of the called class (server) are decided. At a time, one server class is considered for each client class. For one client method sequence, there can be number of server method sequences. In this level, the testing will be effective, if the method sequences of the client class will be complete. As we have tested thoroughly the classes with high criticality at the unit level in the second copy of a case study, we have considered the coupling-based integration testing [108] to cover all the possible interface faults of critical classes. We have taken the help of a coverage analysis testing tool *JaBUTi* [109] for getting the coverage report of a test case. The example of the coverage report by two test cases at the unit level through JaBUTi is shown in Figure 5.3.

At the unit and integration level, though testing time is same for the two copies of a case study, the test sets are different as the priority level of a component is different in different testing methods. After the completion of integration testing, we checked the mutation score of the test sets generated for two copies of a case study by two different testing methods.

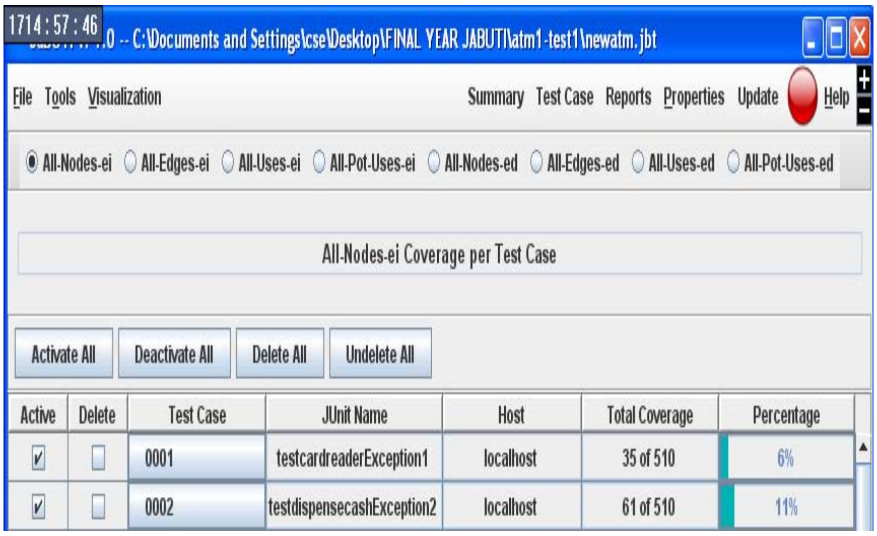


```
        public void testcardreaderException1()
        {
     ATM atm1 = new ATM(42, "Gordon College", "First Nation
                                        null /* We're not really tal

        Session s=new Session(atm1);
        Card c=new Card(2);
    SimulatedBank bank1=new SimulatedBank();
    boolean b=false;
    Money m=new Money(0);
  Message msg=new Message(1,c,23,0,0,0,m);
  boolean b1=bank1.checkPin(msg);
   assertEquals(b,b1);

         }

        public void testdispensecashException2()
       {
           ATM atm1 = new ATM(42, "Gordon College", "First N
                                        null /* We're not really tal
         Simulation theSimulation = new Simulation(atm1);

           Log g=new Log();
```

```
Output - atm (test)
100.00 %
Both tests passed.(5.969 s)
  atm.physical.CashDispenserTestCase passed
    testCashDispense passed (5.75 s)
    testCheckCashOnHand passed (0.0 s)
```

```
setInitialCash
$120.00
Dispensed$20.00
Cash on hand is $100.
checkCashOnHand
false
```

(a) Test cases executed for the component CashDispenser of ATM



C:\Documents and Settings\cse\Desktop\FINAL YEAR JABUTI\atm1-test1\newatm.jbt

All-Nodes-ei Coverage per Test Case

Active	Delete	Test Case	JUnit Name	Host	Total Coverage	Percentage
✓	☐	0001	testcardreaderException1	localhost	35 of 510	6%
✓	☐	0002	testdispensecashException2	localhost	61 of 510	11%

(b) Coverage shown by JaBUTi test tool

Figure 5.3: **Test execution details for the component CashDispenser**

The mutation score S, for a test set T, is defined as follows.

$$MutationScore(S,T) = \frac{\#dead\ mutants}{\#mutants\ seeded - \#equivalent\ mutants}$$

Table 5.7 shows the mutation score of generated test sets by two different testing methods. In Table 5.7, it is observed that MS_T and MS_P are nearly equal. In LMS case study, mutation score of the first testing method is high, whereas in SMA and ATM case studies, the mutation score of the second testing method is high, in which our method is applied. We observed that our method is also equally competent with the first testing method in finding mutants. As we consider average execution time, influence_value and severity for test effort prioritization in addition to

Table 5.7: Mutation Score by two testing methods

$Test$	$TC\#$	$Mu\#$	$EMu\#$	MS_T	MS_P
LMS	112	22	2	0.89	0.82
SMA	73	17	0	0.74	0.77
ATM	211	31	7	0.8	0.89

TC#:Number of test cases, Mu#: Number of mutants, $EMu\#$: Number of equivalent mutants, MS_T: Mutation score by first testing method in which the components were prioritized based on their structural complexity, MS_P: Mutation score by second testing method in which the components were prioritized based on our proposed approach.

structural complexity, we claim that our method exposes the important bugs, which are responsible for frequent failures or severe failures. We conducted another set of experiments to check the types of failures observed in the operational environment.

After resolving the detected bugs, we found that some residual bugs are existing in both the copies of the case studies. A few bugs were detected toward the end of testing, which could not be fixed due to the shortage of testing time. At this point, we again emphasize the fact that our aim is not to achieve complete fault-coverage with a minimal test suite size. We fixed a test budget for each case study before the testing phase and our aim is to ensure the efficiency of both testing methods within the available test budget. Therefore, after the completion of testing phase, we observed the effect of those residual bugs in both copies of each case study by invoking random services. For this, new system level test cases were randomly generated based on operational profile [9] for observing the behavior of the system at post-release stage. At this point, we did not fix any detected bug. Analytical comparison of the two testing methods were done by running the same input set on the results obtained by the discussed testing methods. The tested source code of each case study were again executed to test their behavior at the operational environment.

5.2.1 Result analysis

The results of our simulation studies are summarized in Table 5.8. The headings used for the different columns of the table are listed below.

$Test\#$ is number of test cases in the test set.

$Fail_t\#$ is the number of failures observed in the tested source code obtained through first testing method in which the components were prioritized for testing according to their structural complexity.

$Fail_p\#$ is the number of failures observed in the tested source code obtained through second testing method in which the components were prioritized for testing according to their estimated complexity obtained through our proposed method.

CS is a Case Study
F_{Ca}, F_{Cr}, F_{Ma} and F_M represent the number of catastrophic, Major, marginal and minor failures.
From Table 5.8, it is observed that the post-release failures are less in the second copy of a case study, $Fail_p\#$, to which our method is applied. Not only the number of failures are less, but also catastrophic and critical failures are rarely observed. Only some minor failures are observed in the copy tested by our approach, $Fail_p\#$, such as displaying results in some different format. This type of failures have very less effect on the system and also on customer. Some highly severed failures are observed in the first copy of each case study. It is because, some critical bugs were detected toward the end of test cycle, which were not fixed due to shortage of testing time, whereas these critical bugs were detected at the early stage of test cycle in the second copy of a case study to which our approach was applied.

Through a detailed analysis of the results of both testing methods, we conclude that our proposed test effort prioritization method helps to minimize the post-release failures of a system and also helps in minimizing the catastrophic and major types of failures at the operational environment. As a result of this, user's perception on overall reliability of the system is improved. The efficiency of our proposed method will be improved, if we run the software for long duration by taking a number of test cases based on operational profile.

Table 5.8: Failure observation at the time of release

CS	Test#	$Fail_t\#$				$Fail_p\#$			
		F_{Ca}	F_{Cr}	F_{Ma}	F_M	F_{Ca}	F_{Cr}	F_{Ma}	F_M
LMS	50	0	**2**	3	0	0	0	2	1
	100	0	**5**	3	4	0	0	3	3
	150	**1**	**6**	3	5	0	**2**	3	4
SMA	50	0	**1**	1	4	0	0	1	3
	100	**1**	**1**	1	4	0	0	2	4
	150	**1**	**2**	1	4	0	**1**	2	4
ATM	50	0	**2**	3	6	0	0	2	2
	100	**0**	**2**	4	6	0	0	4	2
	150	**0**	**2**	4	4	0	**0**	4	4

We observed that the performance rate is drastically increased by our method, when the system is executed for a long time, in all the three case studies.

5.3 Summary

We have proposed a criticality estimation method at the code level and prioritized the test effort for various elements within a system according to their estimated criticality. We have considered five important factors of a component: *influence_value*, *average execution time*, *structural complexity*, *severity* and *business value* for criticality computation. Our test effort prioritization method guides the tester to detect the important bugs at the early phase of testing that are responsible for frequent or severe failures. As a result, the user's perception on the reliability of the system is improved within the available test budget. Our approach helps to increase the test efficiency as it is linked to the measure for both internal and external factors of a program element.

The limitation with this approach is that once a priority value is assigned to a component, it is not changed throughout the testing phase. We observed that the importance of a component for testing varies at different instances of testing phase. To solve this problem, we propose a multi-cycle based test effort prioritization method in the next chapter, in which the priority of a component changes between two test cycles.

Chapter 6

Multi Cycle-based Test Effort Prioritization Approach

A moderate size application generally consists of a number of components. The components interact among themselves through a number of operations. Thorough testing of all components is often not feasible due to limited testing budgets. For achieving the desired level of reliability within the available test budget, a good test plan is required. A good test plan helps to monitor and improve the efficiency of testing. Sometimes, the testing team finds important bugs (important from the user's point of view) toward the end of the testing phase. As a result, the development team may not fix it due to shortage of time. Even if the detected bugs are fixed, the testing team may not be able to validate it within a short period.

Test effort prioritization technique helps the tester to do the best possible job within the available test resources [105]. The tester gets the best possible chance to reveal the important bugs. Important bugs are those that reside within critical functions and modules of the system

Our aim is to identify the criticality of a component before the testing phase and allocate test effort to the component according to its criticality. If bugs from the critical components are detected and fixed during testing, the post-release failure rate of the system will be reduced. The importance of a component may vary at different points of the testing phase. If a component has failed in past, then there is a possibility that it will fail in near future [54, 110]. Hence, we analyze the failure history of a component within a system and use it as a factor for estimating the test priority of the component in the next phase of testing.

We propose a multi cycle-based test effort prioritization approach to test the basic functionalities of the system. We institute three different test cycles meant to focus on different aspects of the quality of the system: (i) coverage of critical components,

(ii) coverage of fault-prone components and (iii) coverage of components with high business values.

In the existing prioritization-based testing methods [4–6], the priorities are assigned to the test cases and the priority assignment is done only once for the entire duration of the test. Unlike the existing approaches, we assign priorities to the program elements instead of assigning that to the test cases. We also assign different priorities to the same program element at different test cycles. A stipulated time period is set for a test cycle. The duration of a test cycle may vary under certain circumstances, but the duration of the entire testing time is fixed.

In the first test cycle, we estimate the criticality of a component within a system on the basis of its *influence_value*, *severity* and *execution probability.* In our previous work (Chapter 4), we presented a static metric to compute the influence_value of a class within a system. Dynamic metric captures the dynamic behavior of an application and helps the analyst to make a good test plan. In this chapter, we propose an algorithm to get the *influence_value* through dynamic analysis of source code. We assign test effort to the components according to their estimated criticality. Test priority of a component is set to its estimated criticality with the aim to reveal a number of bugs from high critical components. Though, the *influence_value* of a component affects the reliability of a system, this factor alone is not sufficient to estimate the criticality of a component. In addition to this, we consider another two factors: (i) severity and (ii) execution probability. Severity estimates the impact of failure of the component on the system. Our aim is not only to improve the reliability of a system, but also to reduce the post-release failures that have a high consequence on the system. The reliability calculation only counts the number of failures observed after the testing phase and it does not consider the impact of a failure (severity of a failure) on the system. We consider the execution probability of a component as a factor for criticality computation because the chance of failure is high for a component, which is executed for a number of times. Once the criticality of a component is computed within a system, exhaustive testing has to be carried out to minimize bugs in high critical components. This means, we cut down testing in less critical components to save the testing time for a high critical component. In this cycle, we conduct unit, integration and system testing.

In the second test cycle, the prioritization technique is different. At this stage, we assume that the critical components were tested thoroughly during previous test cycle. The components in which a number of faults were detected in the past are likely to be faulty in future [110]. Keeping this in mind, we set a goal in this cycle, which is different from that of the first cycle. Our aim is to allocate extra test effort to the components that have failed a number of times in the previous cycle. So, we assign test priority to the components on the basis of their failure history. We allocate extra testing effort to the components that have failed a number of times during previous cycle. In this cycle, we again conduct unit, integration and system testing. We match the coverage history of the components with their current estimated priority. There may be some components with different priory in different test cycles. If the desired code coverage level [11, 12] of a component in this cycle is already covered in the previous test cycle, we leave this as it is; otherwise, new test cases are executed for the component.

It is a difficult task for a developer to guess, which high level functions are important to the customer. To get the customer satisfaction and make the testing process effective within the available test budget, we consider the business value associated with a use case scenario for testing at the third test cycle. In the previous test cycles, the components were prioritized based on their criticality and failure history. In the third test cycle, the goal is to rigorously test the use cases that are important to the organization. Domain experts observe that the values of use cases follow a Pareto distribution, i.e., 20% of the requirements cover 80% of business value [20]. Hence, from a business point of view, test effort distribution based on the return on investment will be effective. In the third test cycle, we first prioritize the use case scenarios within a system based on their business values and then conduct only system testing.

We apply our proposed multi cycle-based test effort prioritization approach on LMS, SMA and ATM case studies. These are already introduced in Chapter 4. We illustrate our proposed approach through the case study ATM. Figure 6.1 shows the communication diagram of *withdraw* use case of ATM. We consider it as an running example in next section.

The rest of the chapter is organized as follows. We discuss our proposed multi cycle-based test effort prioritization approach in Section 6.1 and present the experimental studies in Section 6.2. We give a summary of the chapter in Section 6.3.

6.1 Our Approach

It consists of the following steps:

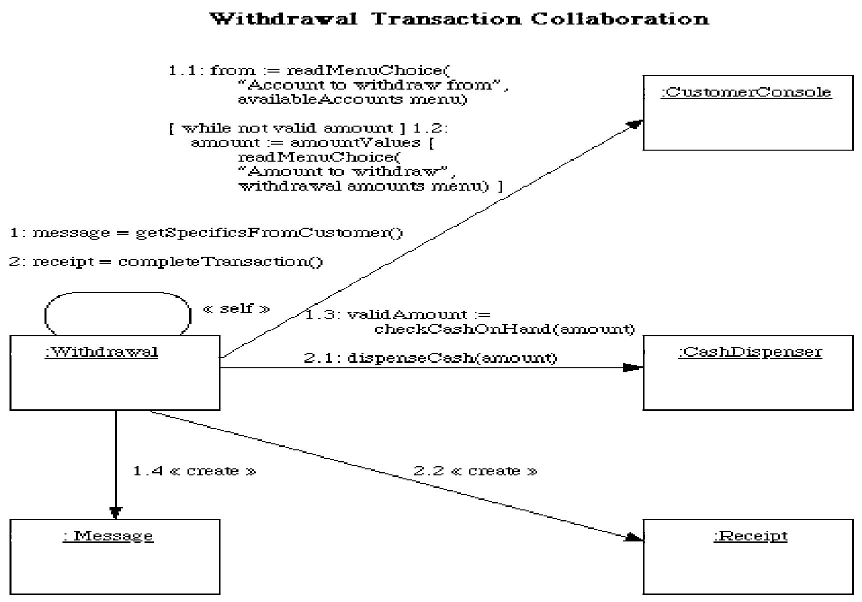

Withdrawal Transaction Collaboration
1.1: from := readMenuChoice("Account to withdraw from", availableAccounts menu)
[while not valid amount] 1.2: amount := amountValues [readMenuChoice("Amount to withdraw", withdrawal amounts menu)]
1: message = getSpecificsFromCustomer()
2: receipt = completeTransaction()
« self »
1.3: validAmount := checkCashOnHand(amount)
2.1: dispenseCash(amount)
1.4 « create »
2.2 « create »
:CustomerConsole
:Withdrawal
:CashDispenser
: Message
:Receipt

Figure 6.1: **Communication diagram for *withdraw* use case**

1. First test cycle: Computing the criticality of a component and prioritizing the components for testing according to their estimated criticality (Section 6.1.1).

2. Second test cycle: Collecting the failure and fault detection history of the components in the previous test cycle and prioritizing the components accordingly (Section 6.1.2).

3. Third test cycle: Computing the business value associated with each high level function (use case) and prioritizing the use case scenarios according to their estimated business values (Section 6.1.3).

Below, we discuss each step in detail.

6.1.1 First test cycle

First, we compute the criticality of a component within a scenario and then, compute the criticality of that component within the whole system. For criticality analysis, we consider two major inputs: dynamic *influence_value* and *severity* of a component. How to estimate the severity of a component at the code level is already discussed in Chapter 5. First, we discuss about the Dynamic Influence Metric and then present the approach for criticality computation. Once the criticality of a component within various scenarios of a system are estimated, we compute the criticality of the component within the whole system. Finally, we assign test effort to various components according to their estimated criticality.

Dynamic Influence Metric

We have already proposed a code-based algorithm to compute the influence_value of a component at the class level in Chapter 4. Testing the behavior of an object is an important task in testing the object-oriented program. The behavior of an object at any execution point can be tested by analyzing the slice of that object at that point during run time. If an object is executed for a long time then, there is a high probability of execution of any existing faults in that object. The occurrence of state transitions is also high in that object due to the invocation of various methods and modification of its attributes. According to Briand et al. [90], the existence of a bug within a class with high export coupling causes frequent failures as it is used by many number of classes. Keeping this in mind, we propose an execution-trace based metric called *Dynamic Influence Metric* and use it for computing the criticality of a component within a scenario.

To get the Dynamic Influence Metric, first we propose a new slicing technique to compute the slices of various interacting objects within a scenario. Then, we use the slices to get the influence_value of a given object within the scenario. In Chapter 4, we have already mentioned that the class level influence metric shows how many other classes are requesting the services from a class within a system, but in object level, using the influence metric, we can get how often these requests are executed within a scenario. Suppose there are two possibilities within a scenario: (i) class c_1 is requesting services from another class c_2 five times (ii) class c_1 is requesting services from five different classes. The first one shows the *number of service invocations* whereas the later one shows the *number of distinct services invoked* within a scenario. The first one is collected at run time and is used to compute the dynamic influence of an object. In object level, our algorithm shows an object is providing services to which objects and how many times to each object within a scenario. In this level, we check how many objects are using the given object, directly or indirectly, within a scenario. At statement level, our algorithm shows how many statements are affected by the given object out of the total number of statements executed by the test case. For simplicity, we assume that one use case consists of one scenario; Only the main scenario is considered, the alternate scenarios are not considered. At run time, our approach on Dynamic Influence Metric maintains all dynamic informations such as the occurrence of object creations, deletions, invocation of various methods, attribute references etc.

The successful execution of a method is dependent on the corresponding state of its object. For any unpredictable behavior of a method, it is required to check the consistency of its corresponding object's state. Our object slicing approach helps the tester to check the state space before and after the execution of a method through its data members. Our approach acts as an active monitor and reports the objects which are responsible for changing the state of the corresponding object within a scenario. Our dynamic slicing approach overcomes some limitations of the existing graph reachability methods for slicing [28, 31, 38]. The main limitation in these existing slicing methods is that when the slicing criteria changes, we have to again start from the slicing point. The slices for different variables at different nodes are obtained by traversing the graph several times starting from the slice point. The advantage of our slicing approach is that, the previous results that are saved in memory can be reused instead of starting from the beginning every time. The dynamic slice of an object at any execution point is the combination of dynamic slices of its data members. Mund and Mall [111] have proposed an inter procedural dynamic slicing algorithm to compute the dynamic slice of procedural programs. The advantage of their method is that, the previous results that are saved in memory are reused instead of starting from the beginning every time. They have not considered the object-orientation aspects. We have extended their work to get the dynamic slice of object-oriented programs. With this new dynamic slicing approach, we compute the Dynamic Influence Metric that gives the influence_value of an object by checking its contribution at every execution step.

We propose an algorithm called *Influence Through Dynamic Slice (ITDS)* to compute the *influence_value* of an object within a scenario. The rationale behind this algorithm is to prioritize the regions of the source code for testing because, some components of a program are more critical and sensitive to bugs than others, and thus should be tested thoroughly. In this section, we first provide the definitions used in our algorithm and then, present our proposed algorithm, *ITDS*. We also explain the working of our proposed algorithm through an example.

Definitions used in the algorithm

Before presenting our proposed algorithm, we first introduce a few definitions that are used in the algorithm. $Def(var)$ and $Use(var)$ represent the set of nodes in the intermediate graph that are used for defining and using the variable var, respectively. During the execution of a program, a statement always corresponds to a node n in

CDG. In the rest of the thesis, we use the terms vertex and node interchangeably.

Def. 1. $RecDefVar(v)$ and $RecDefControl(n)$: A node defining a variable v maintains a data structure named $RecDefVar(v)$ that stores the set of nodes on which the variable v is dependent. Similarly, a predicate node n, called control node, maintains a data structure named $RecDefControl(n)$ that store the set of nodes on which node n is dependent. At the time of execution, a node which maintains $RecDefVar(v)$ or $RecDefControl(n)$ based on the node type (defining a variable, a predicate) is updated as:
$\{n \cup RecDefVar(var_1) \cup RecDefVar(var_2) \cup \ldots \cup RecDefVar(var_k)$
$\cup$ $RecDefControl(S) \cup ActiveCallSlice\}$ where, $\{var_1,\ var_2,\ \ldots, var_k\}$ are the variables used at node n and S is the most recently executed control node under which node n is executing. *ActiveCallSlice* is described below. It stores the information of calling function. If n is a loop control node, and the present execution of node n corresponds to exit from the loop, then $RecDefControl(n) = \emptyset$.
Def. 2. *ActiveCallSlice*: At the time of execution of a program, the data structure *ActiveCallSlice* is used to maintain the information of the most recent function call. At a particular instance of execution time, it represents node n corresponding to the most recent execution of calling a function.
Def. 3. *CallSliceStack*: It is a stack of function calls. At the execution time, it stores a relevant sequence of nested function calls.
Def. 4. *ActiveReturnSlice*: It maintains the data structure of a return statement. At the execution of a return node n,
$ActiveReturnSlice = \{\ n \cup ActiveCallSlice \cup RecDefVar(var_1) \cup RecDefVar(var_2)$
$\cup \cdots \cup RecDefVar(var_k) \cup RecDefControl(S)\}$,
where $var_1,\ var_2, \cdots, var_k$ are the variables used at node n and S is the most recently executed control node under which node n is executing.
Def. 5. $Formal(n, f), Actual(n, a)$: When a function is called at node n, some parameters may be passed by value or by reference. If at the calling node n, the actual parameter is a and its corresponding formal parameter is f then, $Actual(n, a) = f \Leftrightarrow Formal(n, f) = a$.
The examples for each of the above definition are given later in this section through an example program (working of ITDS algorithm).

Algorithm ITDS

The input data are provided to run the program and the name of the desired object is provided to calculate its influence. ITDS provides two outputs: (i) the dynamic slice of an object at any execution point and (ii) the *influence_value* of the given object after the execution is completed. The dynamic slice of an object is required to compute *influence_value* and also used at the time of debugging. *influence_value* of an object is used as an input at the time of criticality computation. The *influence_value* of an object is computed by checking how many statements of the source code are dependent on the given object out of the total number of statements executed by the supplied input. First, an intermediate representation of the source code called Control Dependence Graph (CDG) [112] is constructed. We store the frequency of use of a node by a given object at run time, as there is a difference between a node used by ten different objects and a node used by an object ten times. At the execution of a scenario, our algorithm maintains the set of objects that are dependent on a given object and computes the *influence_value* of the given object within the scenario.

During the execution of a program, we maintain a set of dependent nodes for each variable that are used during program execution. Our algorithm, ITDS, checks whether the currently executed node is using the desired object, for which the influence_value is computed. The currently executed node will be added to the influence_set of the desired object, if it uses any node from the dependence set of the desired object. We use the data structure named *Active_object_set* to get the list of currently executed objects at any instance of execution. When a method is invoked, all the data members of the corresponding object are passed as call by reference. Now, we present our algorithm, ITDS, in pseudo code form.

```
Algorithm: ITDS(CDG, Object O) {
Input: CDG of an object-oriented program and the desired object for which the influence_value will be calculated.
Output 1: Dynamic slice of an object at any execution point.
Output 2: influence_value of the desired object.

  1. Do the initialization before the execution of the program starts.
     Set $CallSliceStack = \emptyset$, $ActiveCallSlice = \emptyset$, $influence(O) = \emptyset$,
     $total\_executed\_nodes = 0$, $temp = \emptyset$ and $Active\_object\_set = \emptyset$.
```

2. Run the program with the given set of inputs and repeat the following steps until the program ends.
   Let node n in CDG corresponds to the statement s of the program.
   - 2.1 Carry out the following before each statement s of the program is executed.
     - a) If node n represents a method invocation, do the followings:
       - a.1) If node n is a call to a constructor class then, store the object in $Active\_object\_set$ and for each data member d of O do:
         $RecDefVar(O.d) = \emptyset$.
         If node n is a call to a destructor class then, delete the object from $Active\_object\_set$.
       - a.2) Update *ActiveCallSlice* and *CallSliceStack* as in Def. 2 and Def. 3, respectively.
       - a.3) For each actual parameter a explicitly defined in the calling node n do:
         $RecDefVar(Formal(n, a)) = RecDefVar(a)$
         // If the actual parameter is an object then, each data member of that object is an actual parameter for that function call.
     - b) If n is a $RETURN$ node then, update *ActiveReturnSlice* as in Def. 4.
   - 2.2 Carry out the following after each statement s of the program P is executed.

     // update the data structure of node n

     - a) If node n is only defining a variable var and not a call node then Update $RecDefVar(var)$ as in Def. 1.
     - b) Else if node n is a control node then update $RecDefControl(n)$ as in Def. 1.
     - c) Else if n is a node which represents a method invocation statement then
       - c.1) If the corresponding invoked method returns a value which is defining a variable var in node n then
         $RecDefVar(var) = ActiveReturnSlice$.
       - c.2) First, update *CallSliceStack* and *ActiveCallSlice* as in Def. 3 and Def. 2 respectively and then, set $ActiveReturnSlice = \emptyset$.

```
            c.3) For each local variable $l\_var$ of the called function do
                     $RecDefVar(l\_var) = \emptyset$
        d) Store the updated data structure of node $n$ in set $temp$.
        e) $total\_executed\_nodes++$
    2.3 Carry out the following to include the current node $n$ in the influence list
        of object $O$
                        //Check Influence of the Object
        a) If node $n$ is a call to a member function by the input object $O$ or
           object $O$ is an actual parameter in the function call then, mark node
           $n$.
        b) Else, check the set $temp$ to find whether any node of
           $RecDefVar(O.d_i)$ is used in current node $n$. If the node is used then,
           mark node $n$.
           // $RecDefVar(O.d_i)$ contains the set of nodes used in defining the
                              data members of object O
        c) If $n$ is marked then add $n$ and RecDefControl(S) to $influence(O)$,
           where $S$ is the most recently executed control node under which node
           $n$ is executing, i.e. $influence(O) = influence(O) \cup n \cup RecDefControl(S)$.
    2.4 $temp = \emptyset$
                              // Object Slicing
    2.5 If a slicing command $< n, obj >$ is given, where $n$ is the currently executed
        node and $obj$ is any object in the $Active\_object\_set$, then
        DynamicSlice $(n, obj)$ = $RecDefVar(obj.d_1) \cup RecDefVar(obj.d_2) \cup \cdots \cup$
        $RecDefVar(obj.d_n)$, where $d_i$ is the i-th data member of $obj$ and RecDefVar($obj.d_i$)
        is the updated data structure of $d_i$ after execution of current node.
    2.6 Exit if the execution of the program is completed or aborted.
3. After the end of execution, calculate % of influence of the object $O$ as follows:
```

$$influence\_value(O) = \frac{\text{Number of elements in the set } \textit{influence (O)}}{\text{total\_executed\_nodes}} \times 100$$

}

Explanation of ITDS Algorithm: Algorithm ITDS calculates the influence of a single object in an object-oriented program, where influence refers to the contribution or usage of the object at every execution step. The influence is calculated as the total number of execution points (statements) involving the given object divided by the total number of execution points in the program. This influence value serves as a metric in test effort prioritization. The algorithm also produces the dynamic slice of an object, identifying the nodes of the program source that would need to be reviewed with a high priority in order to detect bugs in the given object.

ITDS checks the type of each node in a CDG and performs different operations on different nodes. A statement of a program is represented as a node in CDG. A node may be a call node or a definition node or a control node or a return node. Call node calls a function. A definition node defines a variable. A control node defines a loop, on which the execution of other nodes are dependent. A return node returns the output of a called function. The algorithm performs some computation after the execution of a node. When a node defining a variable is executed, the algorithm maintains the set of nodes on which the variable is dependent. Similarly, when a control node is executed, the algorithm maintains the set of nodes on which the control node is dependent. If a node is a call node or return node, it performs some operation before the execution of the node and also performs some operation after the execution of the node. When a new object is created due to the execution of a call node, which calls to a constructor class, the algorithm maintains dependent list for each data member of the object from that execution point. The algorithm maintains a list of objects that are interacting at any execution point. The algorithm maintains a stack to store the nesting of calls, which is updated before and after the execution of a call node. The algorithm updates the data structure of the formal parameters with that of the actual parameter.

Once the data structures are updated after the execution of a node, the algorithm performs a set of operations to check whether the currently executed node will be included in the influence list of the given object. When a slicing command is given to get the dynamic slice of an object, the algorithm computes the dynamic slice of the object by taking a union of the dynamic slices of its data members. After the execution of the program is completed, the algorithm checks how many nodes are executed and out of that how many nodes have used the given object.

Complexity Analysis: Each statement of our considered program is either a control statement or defining a variable or calling a function or an output statement. The data structures used in the program are *Active_object_set*, $RecDefVar(var)$, $RecDefCon(n)$, *ActiveDataSlice*, *ActiveReturnSlice* and *CallSliceStack*. Excepting *CallSliceStack*, each one has the maximum size of N, where N is the total number of statements in the program. The size of *CallSliceStack* is $c * N$, where c is the maximum level of call nesting in a program. So, the worst case space complexity is $O(N)$. The time complexity is linear to the execution time of a program.

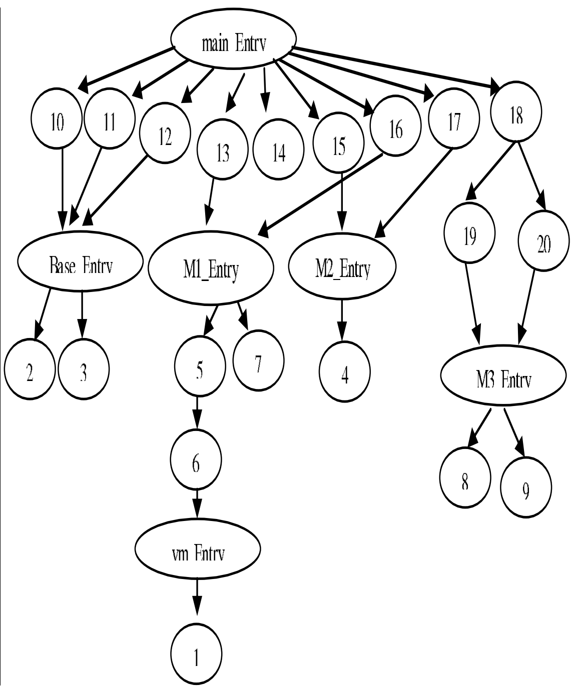
Working of ITDS Algorithm

Consider the example program shown in Figure 6.2a. The Control Dependence Graph (CDG) of the example program is shown in Figure 6.2b. During the initialization step, the algorithm sets CallSliceStack=$\emptyset$ and ActiveCallSlice =$\emptyset$. We have run the program with input *12* for *n* and computed the *influence_value* of object *bx* after the execution is completed. Now, we have the followings for some executed nodes of the program.

```
Class Base
 { int a,b;
   vm( )
   { 1  a = a + b; }
  Public:
  Base()
   {  2   a = 0;
       3. b=0; }
  m2(int i)
   { 4  b=b+i;}

  m1( )
  { 5 if  (b >  0)
     6     vm();
    7  b=b+1; }

 m3(Base O)
  { 8  a=a +O.a;
    9  b=b+O.b; }
```

```
void main()
{
int n;
10 Base bx;
11 Base cx;
12 Base dx;
13 bx.m1();
14 cin >> n;
15 bx.m2(n);
16 bx.m1();
17 dx.m2(n);
18 if(n%5 !=0)
    then
19  dx.m3(cx);
   else
20  bx.m3(cx);
}
```

(a) An example program



(b) Its Control Dependence Graph

Figure 6.2: **Program with its Control Dependence Graph**

- After execution of node 10:$RecDefVar(bx.a) = \{2, 10\}, RecDefVar(bx.b) = \{3, 10\}$

- After execution of node 11: $RecDefVar(cx.a) = \{2, 11\}, RecDefVar(cx.b) = \{3, 11\}$
- After execution of node 12: $RecDefVar(dx.a) = \{2, 12\}, RecDefVar(dx.b) = \{3, 12\}$
- Before execution of node 13: $CallSliceStack = \{13\}, ActiveCallSlice = \{13\}$
- After execution of node 5: $RecDefControl(5) = \{5 \cup RecDefVar(bx.b) \cup ActiveCallSlice\}$=$\{5, 3, 10, 13\}$
- After execution of node 7: $RecDefVar(bx.b) = \{7 \cup RecDefVar(bx.b) \cup ActiveCallSlice\}$=$\{7, 3, 10, 13\}$
- After execution of node 13: $CallSliceStack = \{\emptyset\}, ActiveCallSlice = \{\emptyset\}$
- After execution of node 14: $RecDefVar(n) = \{14\}$
- Before execution of node 15: $CallSliceStack = \{15\}$, $ActiveCallSlice$=$\{15\}$, $RecDefVar(Formal(n)) = \{RecDefVar(i)\} = \{14\}$.
- After execution of node 4: $RecDefVar(bx.b) = \{4 \cup RecDefVar(bx.b) \cup RecDefVar(i) \cup ActiveCallSlice\}$=$\{4, 7, 3, 10, 13, 14, 15\}$
- After execution of node 15: $CallSliceStack = \{\emptyset\}, ActiveCallSlice = \{\emptyset\}$
- Before execution of node 16: $CallSliceStack = \{16\}, ActiveCallSlice = \{16\}$
- After execution of node 5: $RecDefCon(5) = \{5 \cup RecDefVar(bx.b) \cup ActiveCallSlice\} = \{5, 4, 7, 3, 10, 13, 14, 15, 16\}$
- Before execution of node 6: $CallSliceStack = \{\{16\}, \{6 \cup RecDefCon(5) \cup ActiveCallSlice\}\} = \{16, 6, 5, 4, 7, 3, 10, 13, 14, 15, 16\}$,
  $ActiveCallSlice = \{6, 5, 4, 7, 3, 10, 13, 14, 15, 16\}$
- After execution of node 1: $RecDefVar(bx.a) = \{1 \cup RecDefVar(bx.a) \cup RecDefVar(bx.b) \cup ActiveCallSlice\} = \{1, 2, 10, 4, 7, 3, 13, 14, 15, 5, 6, 16\}$
- After execution of node 6: $CallSliceStack = \{16\}$, $ActiveCallSlice = \{16\}$
- After execution of node 7: $RecDefVar(bx.b) = \{7 \cup RecDefVar(bx.b) \cup ActiveCallSlice\} = \{7, 4, 3, 10, 13, 14, 15, 16\}$

- After execution of node 16: $CallSliceStack = \{\emptyset\}, ActiveCallSlice = \{\emptyset\}$

The dynamic slice of an object is the union of the dynamic slices of its data members. The data structure RecDefVar(v) stores the dynamic slice of a variable v. For example, the dynamic slice of bx after the execution of node 16 is given by
$DynamicSlice(16, bx) = \{RecDefVar(bx.a) \cup RecDefVar(bx.b)\}$
$=\{$ 1, 2, 10, 4, 7, 3, 13, 14, 15, 5, 6, 16 $\} \cup \{$ 7, 4, 3, 10, 13, 14, 15, 16 $\}= \{$ 1, 2, 10, 4, 7, 3, 13, 14, 15, 5, 6, 16 $\}$. Proper inspection or review is required only for these statements instead of the whole program in order to find the bugs in object bx.

A node will be added to the influence set of an object, if the node is using a node that belongs to the dynamic slice of that object. We have checked this in Statement 2.3 of **ITDS** algorithm. After the execution of the program is completed, the influence set for object bx is computed as influence_set(bx)= { 10,13,15,16,1,4,5,6,7 }. So, influence_value(bx)=(9/19)*100=47%.

Severity Analysis

For each failure mode of a component within a scenario, we identify the worst consequences on the system, service or customer and determine the seriousness of the worst effect. It is called the severity of the failure mode. The severity weights of 0.25, 0.50, 0.75, and 0.95 are assigned to Minor, Marginal, Major, and Catastrophic severity classes respectively as suggested in [82]. We consider only the highest severity among the severities of all failure modes of a component within a scenario. For example, the severity of the failure of function *dispenseCash(amount)* of component *CashDispenser* within *withdraw* scenario is decided from Table 6.1. As we consider

Table 6.1: Failure mode of *dispenseCash()* of component *CashDispenser*

Potential Failure Mode	Effect of failure	Severity	Cause of Failure
Does not dispense cash	Customer is dissatisfied	Major	Insufficient cash but no message to the customer
Dispense too much cash	Bank looses money	Catastrophic	Loading procedure is wrong or Bills stuck together
Takes too long time to dispense cash	Customer becomes irritated	Minor	Heavy computer network traffic

only the software failures and not the hardware failures, only the first failure mode, *Does not dispense cash* is considered. The failure might be due to a fault in showing an insufficient cash alert message by CashDispenser component.

Criticality computation for a component

We compute the criticality of a component by combining two factors of the component within a scenario: (i) influence_value and (ii) severity.

Actually in the run time environment, we deal with objects instead of components. We get the influence_value and severity for an object within a scenario. If multiple objects of a component exist within a scenario then, influence of the component refers to the highest influence_value among the influence_values of all objects of the same component. Similarly, the severity of the component refers to the highest severity value among all severity values assigned to various services of different objects of the component within the scenario.

The criticality of a component c_i within a scenario S_j is computed as follows.

$$criticality({c_i}^j) = influence({c_i}^j) \times severity({c_i}^j) \tag{6.1}$$

where, $influence({c_i}^j)$ and $severity({c_i}^j)$ represent the influence_value and severity of component c_i within scenario S_j. The normalized criticality of component c_i within scenario S_j, $crit({c_i}^j)$, is obtained by normalizing the criticality of component c_i with respect to the sum of criticality for all active components within the scenario S_j i.e.

$$crit({c_i}^j) = \frac{criticality({c_i}^j)}{\sum_{k=1}^{n} criticality({c_k}^j)} \tag{6.2}$$

Once, the criticality of a component within a scenario is decided, we add the criticality of the component within various scenarios and obtain the criticality of the component within the entire system. For this, the extra input we require is the execution probability of various scenarios within the system. The execution probability of a scenario is estimated through *operational profile* of the system. The criticality of a component within a system is computed as follows.

$$criticality(c_i) = \sum_{j=1}^{nos} (crit({c_i}^j) \times p(j)) \tag{6.3}$$

where, $crit({c_i}^j)$ is the normalized criticality of component c_i within j-th scenario, $p(j)$ is the execution probability of j-th scenario and nos is the number of scenarios within a system. $crit({c_i}^j)$=0, if the component c_i is not used within j-th scenario. We obtain the normalized criticality of a component within a system using Equation 6.2.

Now, we apply our proposed approach on ATM case study. For simplicity, we consider only the main scenario of a use case. As, we consider only one scenario of a

Table 6.2: Execution Probability of various use cases of ATM

Use Case	withdraw	inquiry	deposit	transfer
Probability	0.7	0.2	0.05	0.05

use case, the execution probability of a use case is assigned to its main scenario. We assume the execution probabilities of various use cases of ATM as given in Table 6.2. Table 6.3 shows the normalized criticality of various components within the scenario of *withdraw* use case. It shows that *Withdrawal* component is the most critical. Similarly, the components *Session* and *NetworkToBank* have also high influence_values within the scenario.

Table 6.3: Normalized criticality of various components within *withdraw* scenario

Comp	Withdrawal	Session	CD	CC	CR	NB	Receipt	Message
Crit	0.24	0.21	0.12	0.08	0.13	0.17	0.02	0.03

CD: CashDispenser; CC: CustomerConsole; CR: CardReader; NB: NetworkToBank

We have computed the criticality of various components within the whole system using Equation 6.3 and observed that the components *Session*, *NetworkToBank*, *CardReader*, *Withdrawal* and *CustomerConsole* are critical than others, within the ATM system.

Priority assignment and testing

In this cycle, we prioritize the components within a system according to their criticality. At the unit level, the percentage of code coverage for various components are decided based on their priority values. For example, 100% statement coverage and 90% decision coverage may be conducted for the highest critical component whereas, it may be less for a component having low criticality. Similarly, at the time of integration testing, 90% parameter and 80% interface coverage may be conducted for a high critical object whereas it may be low for others.

At the time of system testing, the test cases are selected keeping in mind that the high priority components will be executed a number of times compared to others. Hence, the cost of a test case[1] is considered as the sum of the criticality of various components that are covered by the test case. The cost of a test case T_i, denoted as $Cost(T_i)$, is expressed as follows.

$$Cost(T_i) = \sum_{C_k \in exe\_set(T_i)} priority(C_k) \tag{6.4}$$

[1]A test case at the system level is designed to execute one scenario.

where, *exe_set*(T_i) is the set of components that are covered by T_i and priority(C_k) is the criticality of k-th component, C_k, in the set *exe_set*(T_i). Suppose the testing team developed a test suite TS_0 and tested an initial build BD_0 for the system under test. Some defects were detected and fixed. This created a new build BD_1. When a new build is developed, two possibilities may arise; either the testing team cannot fix all the detected defects or generate new defects at the time of fixing the detected defects. So, in each test cycle, a sequence of test suites TS_0, TS_1,..TS_k are generated for testing a sequence of builds BD_0, BD_1,...,BD_k. It is expected that the quality of k-th build is higher than (k-1)th build. The test manager decides how many builds to generate based on the test budget and test time. Once a test cycle is complete, the latest build (BD_k) is submitted to the second cycle for testing.

6.1.2 Second test cycle

There is a chance of occurrence of new defects at the time of correcting an existing defect. In this cycle, we consider the failure rate of use case scenarios in the previous cycle. For each failed scenario, we detect the failure point by identifying the faulty component, which is responsible for the failure. The detected bug might be fixed in the previous test cycle. Some detected bugs might not be fixed in the same cycle or new bugs might be introduced at the time of fixing the detected bugs. So, in this cycle, we prioritize the components according to their frequency of failures in the previous test cycle. For this, we also consider the number of defects found in each component in the previous cycle. First, we extract the objects that have failed more than once in the previous test cycle. We put them in a set called *Failed-Set.* Each element of the set consists of two attributes: (i) Name of the object (ii) Failure frequency. Then, we extract the dynamic slice of each object of the Failed-Set based on our proposed ITDS algorithm. It helps us to extract the dependent objects of the said object for testing.

We first extract the objects one by one according to their priority and compute the dynamic slice of the object based on our proposed ITDS algorithm, discussed in Section 6.1.1. It gives us the dependent objects of the said object. During the testing phase, we give importance not only to the failed objects but also to the set of objects that are dependent on the failed objects. It is because, these objects might be infected through the failed objects.

As the priority criteria is changed in this cycle, the priority values of some components may be changed. We check the component's coverage report of the previous cycle. If the required level of coverage of a component in this cycle is already covered in the previous test cycle, then there is nothing to do. Otherwise new test cases are executed in this cycle for the component to cover it upto the desired level. In this cycle, we again conduct unit, integration and system testing. We explain it through ATM case study. We check the failure history of each component of ATM in this cycle and observe that the components *Transfer* and *Session* get higher priority and also equal priority. We check their code coverage history and find that the percentage of code coverage conducted for component *Transfer* is not covered as per its priority in this cycle. New test cases are required for the code coverage of component *Transfer* in this cycle. There is no need of test case generation for component *Session* because the required percentage of code coverage for the component in this cycle is already conducted upto the desired level in previous cycle due to its high criticality. Table 6.3 shows the estimated criticality of various components. We design new test cases only for component *Transfer* and cover it upto the desired level in this cycle. We again conduct integration and system testing as in the previous cycle.

6.1.3 Third test cycle

In this cycle, we conduct a value-based testing [20] with an aim to get a high return on investment and to improve the customer satisfaction on testing. To conduct a value-based testing, it is required to know the business value associated with a high level requirement/feature. A feature is a characteristic or attribute of a product for which work must be done to develop it and deliver it. A feature within a software provides some business value. A feature of a product is delivered to the customer with a hope to get some benefit for a reasonable cost. For a feature, the value is roughly defined as the amount the stake holder is willing to pay for the implementation of the feature.

Business value is estimated based on the relationship among satisfying needs, expectations and the resources required to achieve them [20]. The stake holder decides what is his/her requirement and what he gets for what he pays. From various sources, the domain expert first collects the list of requirements, which are important for the customer and the end-user. The tester and the domain expert sit together and prioritize the requirements for testing based on the business value that come from market and from customer. From a business point of view, test effort distribution should be conducted based on the return on investment. The failure of a high level requirement may cause a great loss to the stake holder and to the organization. Hence, a feature with high business value is assigned high priority in this cycle. During the testing phase, the contribution of business helps to increase the return on investment.

According to Wiegers [44], Business Importance shows the business value of a feature. It is the weighted sum of two factors: the benefit of including a feature within a system and the penalty of not including the feature within the system. Benefit is associated with the requirements of the product's business. Penalty is associated with the consequence that the customer or business would suffer if the feature is not included. Both the benefit and penalty are judged by the customer representatives of the software. For example, failing to comply with a government regulation could acquire a high penalty even if the customer benefit is low. The set of requirements with a low benefit and a low penalty add cost but little value. As benefit and penalty are two factors associated with Business Importance, we define the weights of the features as a vector [1].

Weights of Business Importance $= \begin{bmatrix} W_b \\ W_p \end{bmatrix}$,

where W_b and W_p specify the weights associated with benefit and penalty, respectively. The Total Business Importance is defined as follow:

$$Total\ Business\ Importance = W_b \times Benefit + W_p \times Penalty \tag{6.5}$$

We get the Total Business Importance on the basis of individual ranking of the benefit and penalty of a working feature. We get the normalized Business Importance for a feature by normalizing the Total Business Importance. Figure 6.3 [1] shows the process for estimating the business importance for various features within a product. In the figure, Business Importance index shows the variations in business values of a product over a period of time.

The business values for various use cases of ATM are shown in Table 2.1. We consider only the use cases that are used by the customer. *ATMStartup* and *ATMShutdown* use cases are not considered as they are the basic use cases to run the system.

In this cycle, our aim is to assign priorities to the use case scenarios according to their business values. Unlike the previous test cycles, we do not prioritize the components in this cycle. We assign priorities to use case scenarios and conduct only system testing. For each use case, we consider only one scenario; the successful scenario. The cost of a test case at the system level is decided based on the priority

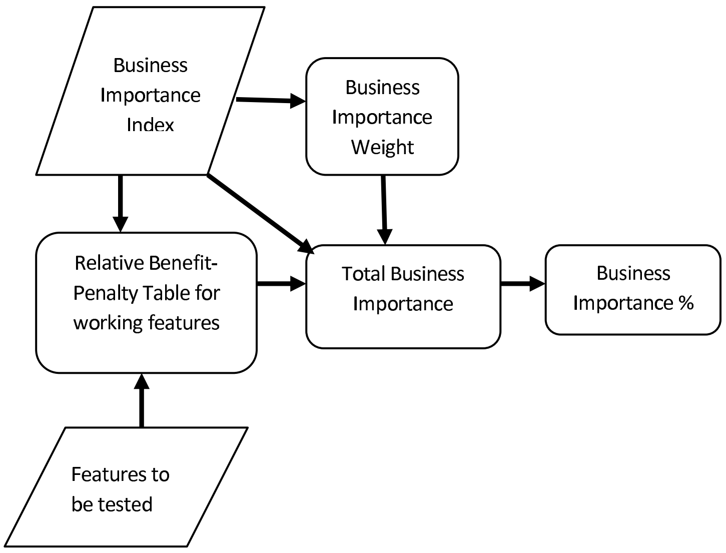

Business Importance Index
Business Importance Weight
Relative Benefit-Penalty Table for working features
Total Business Importance
Business Importance %
Features to be tested

Figure 6.3: **Business Importance estimation**

of a scenario.

6.2 Experimental Studies

We have implemented our proposed multi cycle-based test effort prioritization approach on three case studies and checked the effectiveness of our approach by comparing it with a related approach. We empirically evaluate our approach through ATM, LMS and SMA case studies, explained in Chapter 4.

In order to verify the effectiveness of our approach, we have carried out a series of experiments on the case studies. It has been shown that mutation testing is an effective practice for measuring the efficiency of a testing method [106]. A mutant is said to be killed when it is executed by a test case and the test case fails. We have selected seven number of class mutation operators in our experiment from the mutant model [93]. These operators are mainly designed to modify object-oriented features such as inheritance, polymorphism, dynamic binding and encapsulation. The mutants are selected after a very careful consideration of various types of unit level and integration level faults that may occur during source code implementation. The considered mutation operators are (i) Compatible Reference Type (CRT) (ii) Instance Creation Expression (ICE) (iii) method Parameter Order Change (POC) (iv) Overriding Method Removal (OMR) (v) Access Modifier Changes (AMC) (vi) Exception Handler Removal (EHR) (vii) Exception Handling Change (EHC). These operators are already explained in Chapter 5. Table 6.4 shows the various types of mutants considered for each case study.

Table 6.4: Various types of mutants applied to our case studies

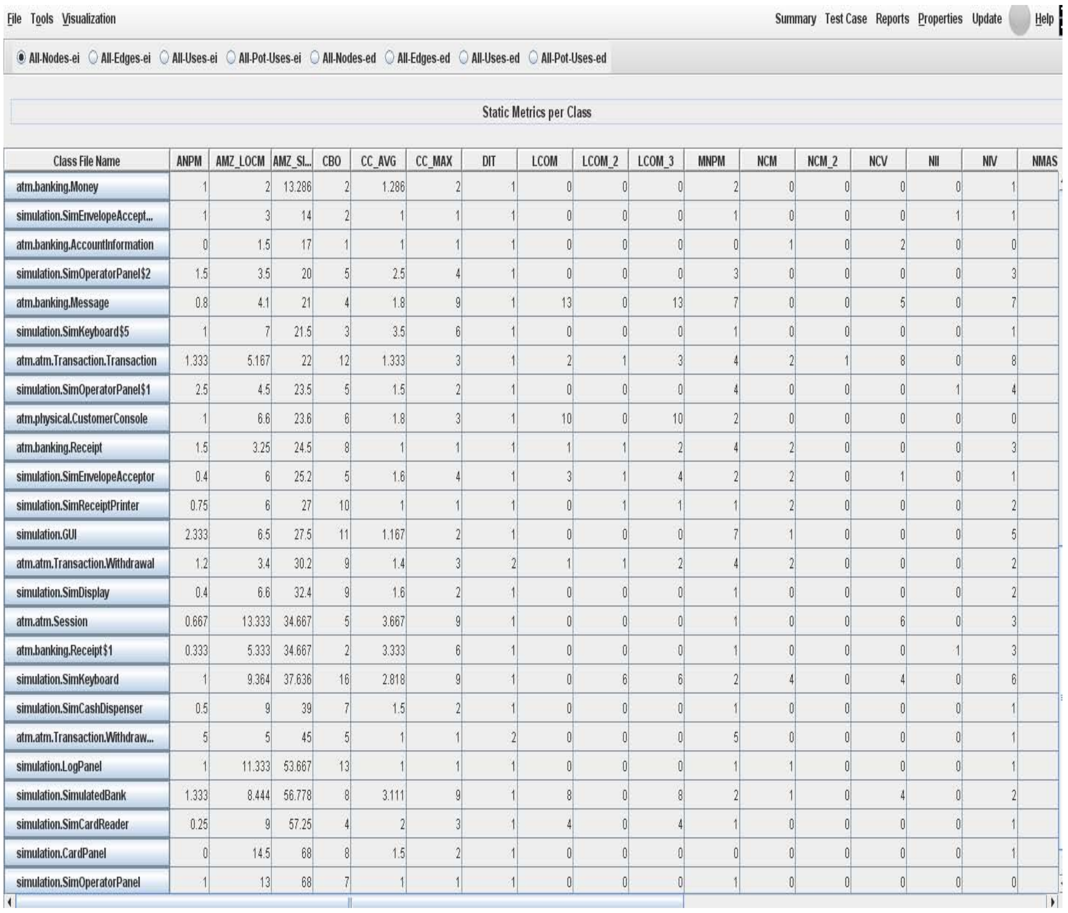
Mutant Operators	*LIS*	*ATM*	*SMA*
CRT	5	8	4
ICE	8	6	2
POC	7	12	9
OMR	6	7	3
AMC	5	2	1
EHR	9	13	0
EHC	13	8	2
Total	53	56	21

We compare our approach with Musa's approach [9] as our aim is to improve the reliability of a system under test which is similar to Musa's approach. We made two copies of the source code of each case study. Musa's approach is applied to a copy of the source code of a case study, called **Copy1** and our proposed multi cycle-based testing approach is applied to the other copy of the source code, called **Copy2**. Musa's approach is basically a black box approach, which assigns test effort to various high level functions based on operational profile. We have extended it from function level to component level and assigned priority values to various components based on their execution probabilities. This helps us to test rigorously the highly executed components during unit testing.

Same testing time was allocated to both the copies. For both the copies, we have used two well known testing criteria based on control-flow: all-nodes (all-primary-nodes[2] and all-secondary-nodes[3]) and all-edges (all-primary-edges[4] and all-secondary-edges[5]) at the unit and integration level. At the time of unit testing, our aim is 100% statement coverage and 70%-99% decision coverage depending on the priority of components. Similarly, at the time of integration testing, our aim is 70%-95% parameter coverage and 60%-80% interface coverage. We use the testing tool JaBUTi [109] to show the percentage of code coverage by a test case. The tool also works as a guide line at the time of generating test cases. It helps to get the various types of complexity through static analysis of the source code, which are required for estimating the number of test cases required for a component. Figure 6.4 shows the complexity metrics obtained by JaBUTi for various components of ATM case study. Figure 6.5 shows (a) the source code of test cases executed to test component Transaction at unit level, (b) the execution result of test cases for component Transaction and (iii)

[2]Statements that are not related with exception-handling mechanism.
[3]Statements that are related with exception handling.
[4]The evaluation of each conditional expressions as true and false.
[5]For each possibility of raise of an exception, the execution of exception-handler.

Figure 6.4: **Complexity metrics of ATM obtained through JaBUTi**



(●) All-Nodes-ei ( ) All-Edges-ei ( ) All-Uses-ei ( ) All-Pot-Uses-ei ( ) All-Nodes-ed ( ) All-Edges-ed ( ) All-Uses-ed ( ) All-Pot-Uses-ed

Static Metrics per Class

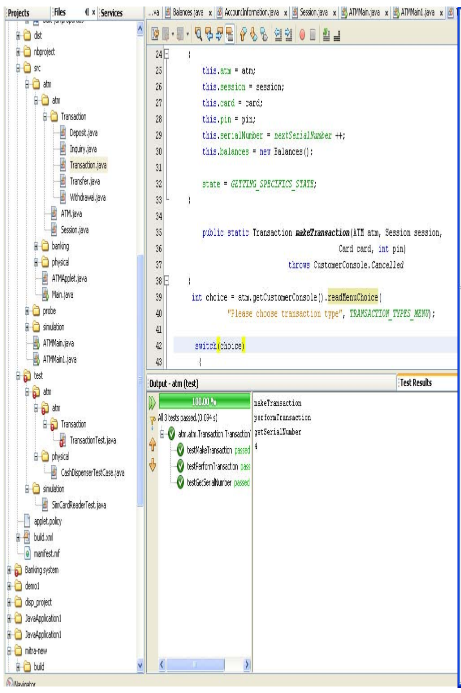
Class File Name	ANPM	AMZ_LOCM	AMZ_SI...	CBO	CC_AVG	CC_MAX	DIT	LCOM	LCOM_2	LCOM_3	MNPM	NCM	NCM_2	NCV	NII	NIV	NMAS
atm.banking.Money	1	2	13.286	2	1.286	2	1	0	0	0	2	0	0	0	0	1	
simulation.SimEnvelopeAccept...	1	3	14	2	1	1	1	0	0	0	1	0	0	0	1	1	
atm.banking.AccountInformation	0	1.5	17	1	1	1	1	0	0	0	0	1	0	2	0	0	
simulation.SimOperatorPanel$2	1.5	3.5	20	5	2.5	4	1	0	0	0	3	0	0	0	0	3	
atm.banking.Message	0.8	4.1	21	4	1.8	9	1	13	0	13	7	0	0	5	0	7	
simulation.SimKeyboard$5	1	7	21.5	3	3.5	6	1	0	0	0	1	0	0	0	0	1	
atm.atm.Transaction.Transaction	1.333	5.167	22	12	1.333	3	1	2	1	3	4	2	1	8	0	8	
simulation.SimOperatorPanel$1	2.5	4.5	23.5	5	1.5	2	1	0	0	0	4	0	0	0	1	4	
atm.physical.CustomerConsole	1	6.6	23.6	6	1.8	3	1	10	0	10	2	0	0	0	0	0	
atm.banking.Receipt	1.5	3.25	24.5	8	1	1	1	1	1	2	4	2	0	0	0	3	
simulation.SimEnvelopeAcceptor	0.4	6	25.2	5	1.6	4	1	3	1	4	2	2	0	1	0	1	
simulation.SimReceiptPrinter	0.75	6	27	10	1	1	1	0	1	1	1	2	0	0	0	2	
simulation.GUI	2.333	6.5	27.5	11	1.167	2	1	0	0	0	7	1	0	0	0	5	
atm.atm.Transaction.Withdrawal	1.2	3.4	30.2	9	1.4	3	2	1	1	2	4	2	0	0	0	2	
simulation.SimDisplay	0.4	6.6	32.4	9	1.6	2	1	0	0	0	1	0	0	0	0	2	
atm.atm.Session	0.667	13.333	34.667	5	3.667	9	1	0	0	0	1	0	0	6	0	3	
atm.banking.Receipt$1	0.333	5.333	34.667	2	3.333	6	1	0	0	0	1	0	0	0	1	3	
simulation.SimKeyboard	1	9.364	37.636	16	2.818	9	1	0	6	6	2	4	0	4	0	6	
simulation.SimCashDispenser	0.5	9	39	7	1.5	2	1	0	0	0	1	0	0	0	0	1	
atm.atm.Transaction.Withdraw...	5	5	45	5	1	1	2	0	0	0	5	0	0	0	0	1	
simulation.LogPanel	1	11.333	53.667	13	1	1	1	0	0	0	1	1	0	0	0	1	
simulation.SimulatedBank	1.333	8.444	56.778	8	3.111	9	1	8	0	8	2	1	0	4	0	2	
simulation.SimCardReader	0.25	9	57.25	4	2	3	1	4	0	4	1	0	0	0	0	1	
simulation.CardPanel	0	14.5	68	8	1.5	2	1	0	0	0	0	0	0	0	0	1	
simulation.SimOperatorPanel	1	13	68	7	1	1	1	0	0	0	1	0	0	0	0	0	

the coverage report by an individual test case through JaBUTi test tool. Table 6.5 shows the fault detection capabilities of the two different prioritization methodapproaches, our approach and Musa's approach.

Table 6.5: Mutants killed by the two different testing approaches

Mutant Operators	Our approach			Musa's approach [9]		
	LIS	*ATM*	*SMA*	*LIS*	*ATM*	*SMA*
CRT	4	7	3	4	5	2
ICE	6	5	2	5	3	1
POC	6	11	7	5	9	6
OMR	5	5	2	5	4	2
AMC	4	1	1	4	2	1
EHR	7	11	0	5	10	0
EHC	11	6	1	9	5	1
Total	43	46	16	37	38	13

From Table 6.5, it is observed that more mutants were killed in our approach than Musa's approach. As our approach considers the *influence_value* of an object and also gives importance to the faulty objects from the test history, more number of faults were detected in our approach than Musa's approach. However, it is not



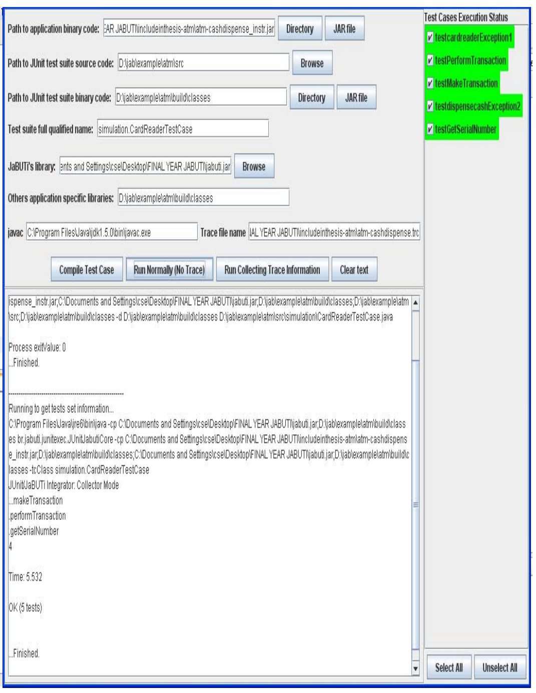
```java
    {
        this.atm = atm;
        this.session = session;
        this.card = card;
        this.pin = pin;
        this.serialNumber = nextSerialNumber ++;
        this.balances = new Balances();

        state = GETTING_SPECIFICS_STATE;
    }

    public static Transaction makeTransaction(ATM atm, Session session,
                                              Card card, int pin)
                              throws CustomerConsole.Cancelled
    {
     int choice = atm.getCustomerConsole().readMenuChoice(
             "Please choose transaction type", TRANSACTION_TYPES_MENU);

      switch(choice)
       {
```

Output - atm (test) | Test Results

```
100.00 %
All 3 tests passed.(0.094 s)
atm.atm.Transaction.Transaction
  testMakeTransaction passed
  testPerformTransaction pass
  testGetSerialNumber passed
```

```
makeTransaction
performTransaction
getSerialNumber
4
```

(a) Source code of a test case to test Transaction class

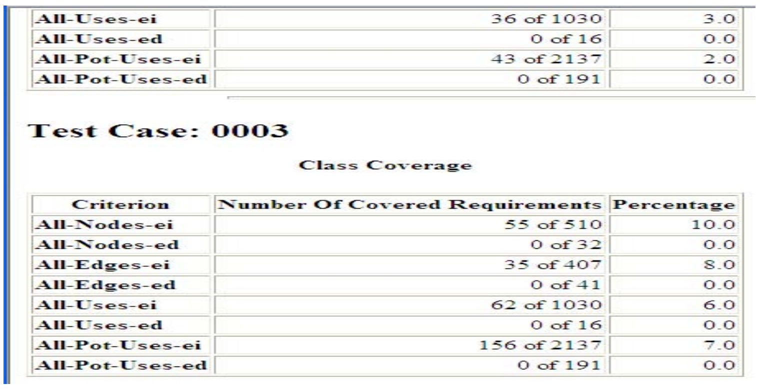


Path to application binary code: :AR JABUTI\includeinthesis-atm\atm-cashdispense_instr.jar
Path to JUnit test suite source code: D:\jab\example\atm\src
Path to JUnit test suite binary code: D:\jab\example\atm\build\classes
Test suite full qualified name: simulation.CardReaderTestCase
JaBUTi's library: ents and Settings\cse\Desktop\FINAL YEAR JABUTI\jabuti.jar
Others application specific libraries: D:\jab\example\atm\build\classes
javac C:\Program Files\Java\jdk1.5.0\bin\javac.exe Trace file name IAL YEAR JABUTI\includeinthesis-atm\atm-cashdispense.trc

Test Cases Execution Status
- ✓ testcardreaderException1
- ✓ testPerformTransaction
- ✓ testMakeTransaction
- ✓ testdispensecashException2
- ✓ testGetSerialNumber

```
ispense_instr.jar;C:\Documents and Settings\cse\Desktop\FINAL YEAR JABUTI\jabuti.jar;D:\jab\example\atm\build\classes;D:\jab\example\atm
\src;D:\jab\example\atm\build\classes -d D:\jab\example\atm\build\classes D:\jab\example\atm\src\simulation\CardReaderTestCase.java

Process exitValue: 0
...Finished.

--------------------------------------------------
Running to get tests set information...
C:\Program Files\Java\jre6\bin\java -cp C:\Documents and Settings\cse\Desktop\FINAL YEAR JABUTI\jabuti.jar;D:\jab\example\atm\build\class
es br.jabuti.junitexec.JUnitJabutiCore -cp C:\Documents and Settings\cse\Desktop\FINAL YEAR JABUTI\includeinthesis-atm\atm-cashdispens
e_instr.jar;D:\jab\example\atm\build\classes;C:\Documents and Settings\cse\Desktop\FINAL YEAR JABUTI\jabuti.jar;D:\jab\example\atm\build\c
lasses -tcClass simulation.CardReaderTestCase
JUnit/JaBUTi Integrator: Collector Mode
...makeTransaction
.performTransaction
.getSerialNumber
4

Time: 5.532

OK (5 tests)


...Finished.
```

(b) Execution Result of test cases



All-Uses-ei	36 of 1030	3.0
All-Uses-ed	0 of 16	0.0
All-Pot-Uses-ei	43 of 2137	2.0
All-Pot-Uses-ed	0 of 191	0.0

Test Case: 0003

Class Coverage

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	55 of 510	10.0
All-Nodes-ed	0 of 32	0.0
All-Edges-ei	35 of 407	8.0
All-Edges-ed	0 of 41	0.0
All-Uses-ei	62 of 1030	6.0
All-Uses-ed	0 of 16	0.0
All-Pot-Uses-ei	156 of 2137	7.0
All-Pot-Uses-ed	0 of 191	0.0

(c) Coverage Report by individual test case

Figure 6.5: **Test Case execution result and coverage report of component *Transaction* in ATM**

true that a testing approach which is effective in detecting faults is also effective in improving the reliability of a system. The reliability of a system is not related to the number of existing faults in a system under test, but related to the probability that a fault leads to a failure which occurs during software execution [9, 13, 14]. It is because, the data input supplied by the user decides which parts of the source code will be executed. An error existing in the non-executed parts will not affect the output.

We conducted a series of experiments for assessing the reliability of the outcome of the two discussed approaches after completion of the testing processes. The software reliability of a system, R, is calculated as given below.

$$R = \sum_{i=1}^{m} p_i \times \Theta_i \tag{6.6}$$

where, p_i and Θ_i represent the execution probability and failure rate of i^{th} sub-domain respectively.
Θ_i is computed as follows:

$$\Theta_i = \frac{1}{n_i}\sum_{j=1}^{n_i} z_{ij} \tag{6.7}$$

where, z_{ij} represents the execution result of a test case which is selected from i^{th} sub-domain for j^{th} time. The value of z_{ij} is 1, if a failure is observed else the value is 0. n_i is the total number of test cases selected from i^{th} sub-domain and $\sum_{i=1}^{m} n_i = n$, where n is the total number of test cases executed in the system and m is the total number of sub-domains of the input domain.

Table 6.6 is a subset of test cases that are designed for ATM case study. Table 6.7 shows the reliability computed for the tested source codes that are obtained by using our approach and Musa's approach, respectively. In this table, $Copy1$ is the source code that is tested by Musa's approach [9] and $Copy2$ is the source code that is tested by our multi cycle-based approach.

6.2.1 Result analysis and discussion

From Table 6.7, we observed that high reliability is observed in $Copy2$ compared to $Copy1$, in each case study. We discuss some situations in which the testing based on Operational Profile implemented in Musa's approach is not giving good result. For example, consider a situation. Suppose, there is a fault in a method m which is executed for a short duration. The return value of m is saved and used by some frequently executed methods of other components. If method m returns a wrong value then, the failure of the system will be high. In this situation, the fault in method m may not be detected as the method is getting less attention in Musa's approach due to low execution probability. It is better explained through the graphical representation of a simple example instead of going to the details of the case studies. Consider the sequence diagrams shown in Figure 6.6. Suppose the execution probabilities of $SD1$ and $SD2$ are 80% and 20% respectively. The average execution time of class D is the lowest as it is used only in $SD1$, but the influence_value is high as it is providing services to a number of classes. As shown in Figure 6.6a, the returned value of class D

Table 6.6: Test cases designed to test various use cases of ATM case study

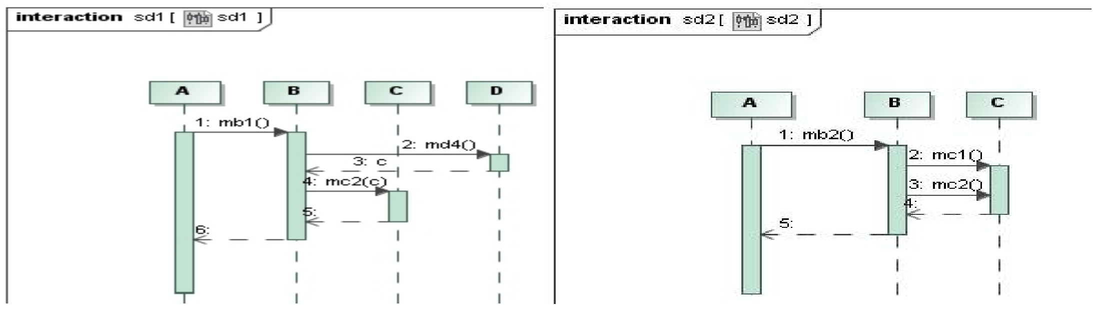
Use case	Function to be tested	Initial condition	Test data	Expected Output
Session	Read ATM card of a customer	System is on and no card is inserted to the system	A card is inserted	System accepts the card and asks for PIN number
	Read an invalid card	same	An invalid card is inserted	System ejects the card and display a message in a new screen
	Accepts the entered PIN	System is asking to enter the pin	PIN is entered	System displays a transaction menu.
	Perform a transaction	A transaction menu is displayed	a transaction is performed	System displays a new screen asking whether to continue another transaction.
withdraw	Choose an account for withdraw	Transaction menu is displayed	Withdraw transaction is chosen	System displays a menu of account types.
	Enter Dollar amount to withdraw	Menu of account type is displayed	Checking account is chosen	System displays a new screen to enter amount for withdraw.
	Perform a withdraw transaction properly	A screen is displayed to enter amount for withdraw	An amount is entered which is not exceeding the total balance of the account and the required amount is currently available in the cash dispenser.	CashDispenser dispenses the entered amount of cash. System prints a receipt showing amount and correct updated balance. Transaction is recorded in the Log.

Table 6.7: Reliability assessment based on two different testing strategies

Case Study	*n*	*TestedCode*	*Reliability*					
			R_1	R_2	R_3	R_4	R_5	R-
ATM	100	Copy1	0.801357	0.800048	0.788101	0.810551	0.811930	**0.8023974**
		Copy2	0.839137	0.851994	0.8428719	0.8379159	0.851109	***0.84460556***
	150	Copy1	0.819388	0.816765	0.819539	0.82117456	0.818901	**0.819153312**
		Copy2	0.840019	0.849270	0.851140	0.842059	0.849718	***0.8464412***
	200	Copy1	0.829901	0.821617	0.820007	0.823166	0.832949	**0.825528**
		Copy2	0.847823	0.839691	0.848823	0.850018	0.848388	***0.8469486***
LMS	100	Copy1	0.890012	0.9019256	0.890081	0.899901	0.902135	**0.89681092**
		Copy2	0.930936	0.940019	0.943211	0.930089	0.929908	***0.9348326***
	150	Copy1	0.891107	0.902517	0.900168	0.900151	0.899079	**0.8986044**
		Copy2	0.941132	0.951108	0.927940	0.949029	0.941187	***0.9420792***
	200	Copy1	0.899567	0.902311	0.899855	0.899129	0.896635	**0.8994994**
		Copy2	0.948783	0.921908	0.942879	0.950125	0.940057	***0.9447504***
SMA	100	Copy1	0.901571	0.915189	0.907651	0.910089	0.901213	**0.9071426**
		Copy2	0.963219	0.958199	0.960001	0.951081	0.948613	***0.9562226***
	150	Copy1	0.902335	0.912877	0.912048	0.908901	0.913687	**0.9099696**
		Copy2	0.957186	0.951928	0.952887	0.959931	0.960896	***0.9565656***
	200	Copy1	0.910093	0.918119	0.908931	0.913913	0.913719	**0.912955**
		Copy2	0.959629	0.959138	0.960019	0.958913	0.951584	***0.9578566***

Copy1:Code tested by Musa's approach; Copy2: Code tested by our multi cycle-based approach; R_i:The reliability obtained at i-th run; R-:=$\frac{\sum_{i=1}^{5}(R_i)}{5}$

is used in class B, C and A, directly or indirectly. If class D will return incorrect value, the highly executed classes A, B and C will be affected. It will increase the failure rate of the overall system. Class D is getting less attention in Musa's approach due to its low execution time. As we are considering the *influence_value* of an object as one factor for test effort prioritization, class D is not neglected in our test approach. It gets appropriate test effort. Another thing is that we have not explicitly shown the

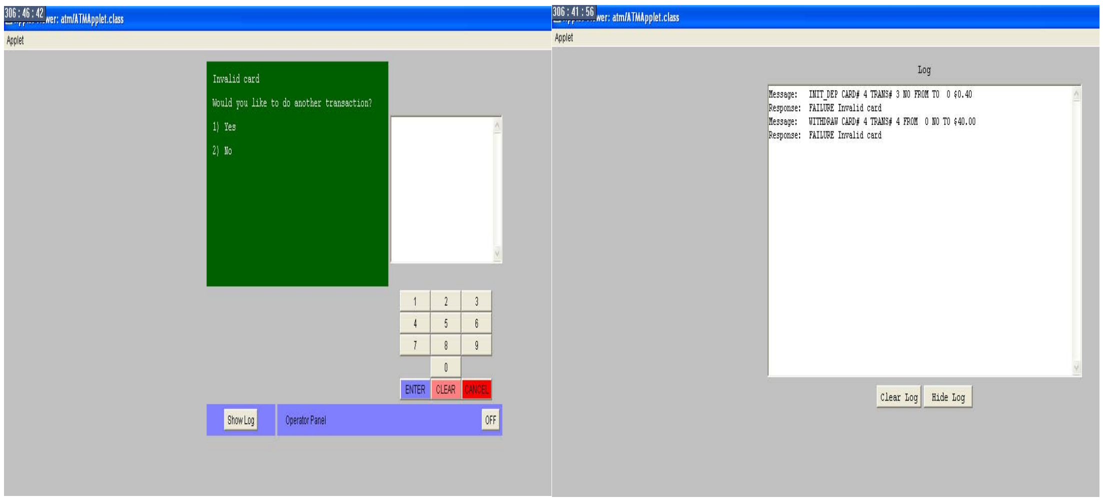


(a) SD1 (b) SD2

Figure 6.6: **Two sample sequence diagrams**

severity of a failure in the experimental result. Reliability estimation only considers the number of failures, the severity of a failure is not considered. The reliability calculated in Table 6.7 gives equal weight to each failure. In a real life application, it is observed that the effect is critical for some failures. As we consider the severity of a failure within a scenario in the first test cycle and business value in the third cycle, a number of important faults were detected and fixed in due time in our approach. All the inserted faults could not be covered in both the copies of the source code, within the allocated testing time. We checked the impact of the observed failures during reliability assessment. Some marginal and minor failures were observed in Copy 2 which was tested by our approach whereas, some major failures were observed in Copy 1 which was tested by Musa's approach. One such major failure found in the tested copy through Musa's approach is just explained below using LMS case study. Instance Creation Expression (ICE) mutant creates an object of same type or different type with different initial states. We observed a failure in Copy 1 at the time of creating a new user (Borrower). In LMS, the borrowers are students (graduates, post-graduates) and staff. The sub-classes of staff are: (i) teaching staffs (professors, assistant professors, lecturers) and (ii) non-teaching staffs. They are classified to various groups according to the access privileges to various system resources. As per the business rule of our case study, LMS, post-graduate students and teaching staffs are only allowed to access journals and transactions. Due to an ICE mutant, the system allowed a non-teaching staff to issue a journal.

As we included severity analysis as one attribute for testing and considered the business value of a scenario, such types of major failures were not observed in the tested source code obtained using our approach. Now, we discuss one minor failure that was observed in Copy 2 of ATM case study. Copy 2 was tested through our approach. We have inserted an invalid card. The system has opened the transaction screen and allowed for a transaction instead of ejecting the card, though any transaction was not performed with the invalid card. This is shown in Figure 6.7. From the log file shown in Figure 6.7b, it is observed that neither the *deposit* nor the


```
Invalid card

Would you like to do another transaction?

1) Yes

2) No
```

```
Message:   INIT_DEP CARD# 4 TRANS# 3 NO FROM TO  0 $0.40
Response:  FAILURE Invalid card
Message:   WITHDRAW CARD# 4 TRANS# 4 FROM  0 NO TO $40.00
Response:  FAILURE Invalid card
```

(a) ATM screen (b) ATM log

Figure 6.7: **A minor failure in ATM**

withdraw transaction is performed in Card# 4, but the system is not ejecting the card after recognizing an invalid card. As shown in Figure 6.7a, only the card was ejected when, the user did not want to continue any transaction further.

6.3 Summary

In this chapter, we have proposed a multi cycle-based test effort prioritization approach for improving the reliability of a system. Our aim is to minimize the critical faults in a system which are responsible for frequent or severe failures in the operational environment. We have computed the criticality of a component and prioritized the components according to their criticality in the first test cycle. In the second test cycle, the components are prioritized based on their failure history and fault detection history. In the third test cycle, the scenarios are prioritized based on their business values. Based on our analytical comparison, we found that the objects with higher criticality indeed determine the probability of failure to a large extent. We have validated our claim through a series of experiments. From our experimental results, it is concluded that by spending extra time and effort in key objects of a system compared to others at the time of testing, the reliability of the system can be improved within the available test budget. The tester gets the best possible chance to reveal the important bugs at the early phase of testing that are occurring frequently or have a negative impact on the user. Since, we have considered the impact of a failure within a scenario, our approach not only helps to increase the reliability of a system, but also minimizes the occurrences of severe types of post-release failures.

Planing at the high level enhances the decision on resource allocation. Estimating the criticality of an architectural element helps both the system analyst and the test manager in planing suitable provision for the crucial elements. If the critical elements are detected at the early phase of software development life cycle then, it will be useful in allotting resources in afterward development phase.

Chapter 7

Ranking Use Cases for Testing

A use case is related to a set of requirements. Cockburn [113] states that, "people seem to consider use cases to be the central element of the requirements or even the central element of the project's development process". For both developer and customer, use case is treated as a semantically meaningful function that provides some value from the user's point of view. As the use cases are available in early iterations, test plan can begin early. An empirical research [19, 77] on early effort estimation have proposed various methods for estimating development effort, but the estimation of test effort is overlooked. A software product can be lunched in due time with sufficient testing, if a test plan is prepared early.

Musa [9] suggested that a tester should select test cases according to the frequency or probability of operational use, for achieving a better reliability. We give an example of test effort distribution among various use cases of an application based on their *execution probabilities* only. First, the test manager decides how many test cases are required to design based on different test variables such as expected time taken to design and run a single test case, time taken to find a bug, time taken to correct a bug and the estimated test budget. Suppose, the following data are given for the estimation of test efforts for LMS case study.

1. To design and run one test takes 2 hours.
2. 10 percent of tests find bugs.
3. Each fault takes 8 hours to correct.
4. Test budget = 1000 hours.

Then, the total number of test cases, T, is decided as: $2 * T + (0.1 * T * 8) = 1000$, i.e. T=358.

Suppose, the execution probability of $IssueItem$ use case is 0.1. The total number of test cases assigned to it is $0.1 * 358 = 36$. In this case, test effort distribution is purely a black box approach and the architectural information of use cases are not used for distributing test cases.

Test effort allocation based on only *operational profile* does not give accurate results always. We present a simple example to explain it. Suppose, for a program under test, the input domain is divided into two sub-domains with equal size. The execution probability of the first one is 49% and the second one is 51%. Further, we assume that the first one contains 50 and the second one contains 200 failure-causing test cases. Suppose, each failure has equal effect on the system. In this case, the test effort distribution based on *operational profile* will not help to achieve high reliability. As the execution probabilities of both the sub-domains are nearly equal, each will get almost equal test effort though, the second input sub-domain is more failure-prone than the first input sub-domain.

A complex program might contain more number of faults compared to a simple program [55]. As the factor complexity is the most important bug generator, the *complexity metric* is used as a parameter for testing [56, 57]. The complexity can vary from one use case to another. In a moderate size application, a simple use case generally takes at most 5 number of steps for its success scenario and its implementation also involves less number of classes. A complex use case takes at least 10 number of steps and its implementation also involves a number of classes. The job of a test manager is to estimate the complexities associated with various use cases and consider the complexity as an important factor at the time of test planing. Though, the estimation of complexity for high level functions at the analysis stage is a tough task, it is better to estimate it as early as possible and refine it in the low level rather than delaying the test estimation and proceeding it in an unplanned fashion. To estimate the complexity of a use case in a quantitative manner at the architectural level, it is required to understand both the structural and functional details of a use case. Complexity estimation at the architectural level helps to estimate the test effort required for a use case, before its implementation and hence, an effective testing can be conducted within the available test resources.

At the time of realization of a use case, many interacting objects change their states and different values of an operational variable force the system to behave differently. Many conditions are checked to execute various scenarios within a use case. Different types of messages such as synchronous and asynchronous pass at the realization of a use case, within a system. Out of these messages, some are critical to the sender and failure of these massages may cause severe damage to the whole system. In a distributed system, different components reside in different nodes and communicate through networks, that increase the probability of connector failures. Due to the architectural dependencies among use cases within a system, some use cases execute in parallel whereas, some execute serially. We analyze the structural and behavioral aspects of a use case and estimate its complexity in a quantitative manner. For this, we collect information from sequence diagrams that are realized for the use cases, state chart diagrams of various components that are activated at the execution of the use case, class diagrams and deployment diagrams of the system under test. Deployment diagrams are required in case of distributed systems. The ability to quantify the complexity of a use case at the analysis stage helps in refining the resource estimation and creates an acceptable quality standard.

Though, the complexity of a use case is related to its fault density, the observed failures within a system are also related to the execution probabilities of various use cases that lead a fault to a failure. The main objective of software testing is to improve the reliability rather than to detect defects. For this, the test cases should be selected based on both the criteria: (i) defect distribution and (ii) how the software is used. Defect distribution is estimated based on the *complexity* of the system and the expected use of a software is decided based on the *operational profile* of the system. To identify the failure-prone use cases, we consider the execution probability of a use case along with its estimated complexity and call it *Occurrence Complexity (OC).*

There is a close relationship between testing and business value of a high level function that comes from market or from customers [43]. Each use case of a system should not be treated with equal importance [97]. Keeping this in view, we propose a test effort prioritization method to estimate the test priority for a use case within a system on the basis of its factors- (i) *complexity*, (ii) *execution probability* and (iii) *business value (Value).* The use cases of a system are ranked according to their priority values. Our proposed prioritization method provides a path to discover the truly critical use cases. This ranking method helps the developer and the test manager to take a decision on test effort distribution in a critical environment, where the customer's expectation is high on the overall quality of the system, timelines are short and resources are limited. It is observed that some use cases with high complexity are less valuable to the organization. The balancing strategy is to assign less effort to low ranked use cases. It can save the resources which can be used for high ranked use cases. As the important use cases are getting a chance to be tested rigorously through our proposed approach, the reliability of the system under test is improved.

The rest of the chapter is organized as follows: We have proposed some factors affecting the complexity of a use case in Section 7.1. We compute the complexity of a use case on the basis of the proposed factors and compute the priority for the use case in Section 7.2. We have conducted a lot of experiments and validated our claim in Section 7.3. The summary of the chapter is discussed in Section 7.4.

7.1 Complexity Factors

When the test plan is made before coding, at the design level, the test manager considers the architecturally relevant aspects. The difficulty lies in analyzing all the architecturally relevant aspects of a use case and ranking it appropriately.

We propose the following eight factors that affect the complexity of a use case. These factors are described with examples in subsequent sections.

1. Sum of complexities of linearly independent paths within a SD.
2. Number of test paths generated within a SD.
3. Number of critical messages transmitted within a SD.
4. Number of operational variables used within a SD.
5. Length of the longest Maximum Message Sequence (MMS) within a SD.
6. Number of external links used within a SD.
7. Number of polymorphic calls within a SD.
8. Architectural dependencies among use cases.

These above proposed complexity factors are explained with examples in the following sections.

7.1.1 Sum of complexities of Linearly Independent Paths within a SD

Cyclomatic-complexity is defined as the number of linearly independent paths[1] in a graph. There is a strong correlation between the cyclomatic-complexity measure and the number of bugs in a program [99]. In this section, we generate the *Control Flow Graph (CFG)* of a SD and count the number of linearly independent paths within the SD and then, estimate the complexity of each path in terms of test effort required. The sum of complexity of all linearly independent paths is the complexity of the SD of the use case. The test manager allocates test effort to the use cases based on their estimated complexity. Higher the complexity of a SD, more test effort is required to test it.

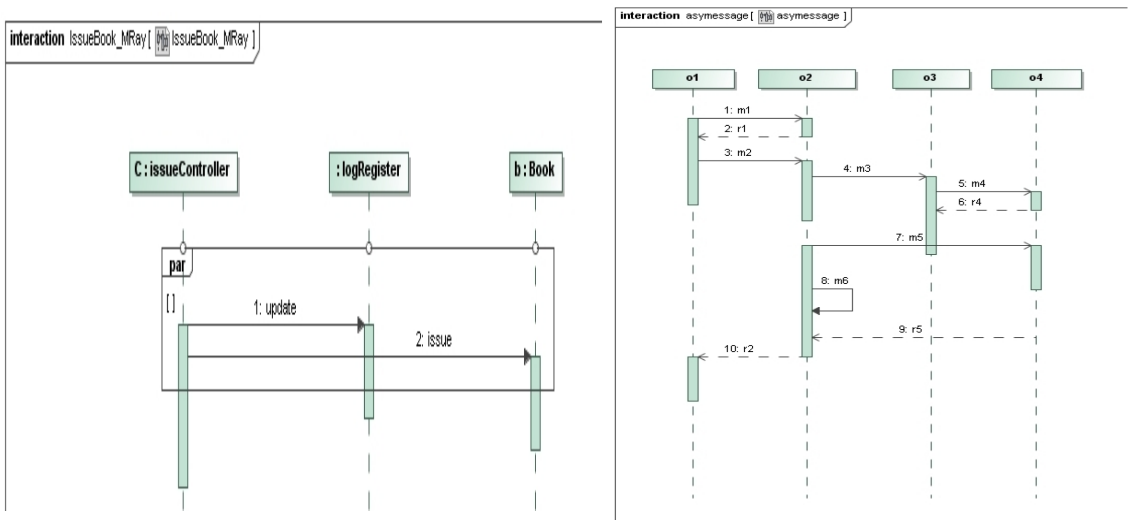
First, the Control Flow Analysis (CFA) of the SD is performed to get the CFG. CFG is used to extract the basic individual paths within a SD. It is the source of estimation for testing. It is named as *Concurrent Control Flow Graph (CCFG)* instead of CFG, due to the presence of *asynchronous* and *parallel* messages within a SD [114]. At the time of execution of a synchronous message, the caller waits for the reply message from the callee. The caller could not initiate any message in between, but in an asynchronous message, the caller does not wait for reply message. It proceeds immediately and cause a concurrent control flow in the SD. Another event for a concurrent control flow is the interaction operator *par*. It is used to support parallel execution of a set of interacting components by causing a number of threads of control. Figure 7.1a shows an example of a SD with *par* interaction operator and Figure 7.1b shows an example of a SD with asynchronous message. Figure 7.1a is a part of the SD that implements *Issue Item* use case of *LMS* case study. The object *IssueController* controls the issue of a book. States of the objects *Book* and *LogRegister*, need to be updated when a book is issued. The controller sends two messages in parallel, one to *Book* and one to *LogRegister* object. Now, these two messages run concurrently. So, the CCFG of a SD is affected by the interaction operator *par*, which causes at least two concurrent threads of control.

In CCFG, each message of SD represents a node. Once the CCFG of a SD is generated as shown in Figure 7.2, we extract all linearly independent Concurrent Control Flow Paths (CCFPs) of the CCFG for testing. CCFP is a control flow path with extra feature: sub-paths are added in CCFG due to concurrent control flow.

[1]A path is linearly independent within a graph if it introduces a node of the graph that is not included in any other linearly independent paths.

Parallel and asynchronous messages cause concurrent control flow at the execution of a scenario within a SD. The concurrency within a CCFG is identified through *fork* and *join* nodes. In a CCFP, an open and close parenthesis represent fork and join nodes respectively. A CCFP within a CCFG is a path which includes all sub-paths going out from a fork node. It includes a path from the start node to the end node containing all residing nodes in the path. There can be a number of CCFPs in a CCFG. In our example, as there is no condition in the SD (see Figure 7.1b), we get only one linearly independent path in the CCFG shown in Figure 7.2 and call it $\rho1$. It is given below.

$$\rho_1 = m1r1m2m3\left(\begin{array}{c} m4r4 \\ m5\left(\begin{array}{c} m6 \\ r5 \end{array}\right) \end{array}\right)r2$$



(a) SD with parallel message (b) SD with asynchronous message

Figure 7.1: **SDs with *asynchronous* and *parallel* messages**

To estimate the complexity associated with a CCFG, we require two inputs: (i) the number of linearly independent paths and (ii) the complexity of each path. The complexity of a path is determined by checking the number of simple nodes and fork nodes (concurrent sub-paths) in a path. If there is a couple of concurrent sub-paths in a path then, extra effort is required to test the path. It is because, testing of a concurrent path is not straight forward. Concurrent bugs are difficult to detect due to the nondeterministic behavior exhibited by parallel applications. Even if these bugs are detected, it is also a difficult process to reproduce them consistently.

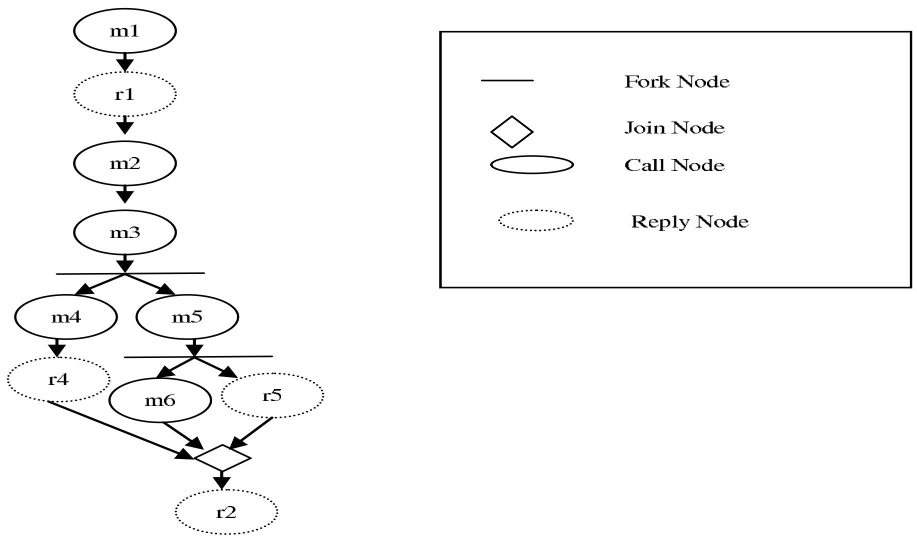


Figure 7.2: **CCFG of Figure 7.1b**

Further, after a bug is fixed, it is also a difficult job to ensure that the bug is corrected truly and not simply masked. The concurrent bugs are categorized as *race conditions*, *incorrect mutual exclusions*, and *memory reordering.* We cannot immediately observe the consequences of a race condition. It might be visible after some time or in a totally different part of the program. There is also a need to synchronize the operations between threads. For this, extra overhead is required. As extra test effort is required for a concurrent node, we assign high weight to a concurrent node compared to a simple node, at the time of calculating complexity for a path. We assign weight of 5 to a concurrent node (node under a fork) and weight of 1 to a simple node. The complexity of path $\rho 1$ of Figure 7.2 is calculated as $1 * 5 + (3 + 2 * 5) * 5 = 70$. The complexity of a CCFG is the sum of complexities of its paths. It is given by

$$Complexity(CCFG) = \sum_{i=1}^{n} (Complexity(CCFP_i))$$

where, $Complexity(CCFP_i)$ is the complexity of $CCFP_i$ of the CCFG and n is the number of paths in CCFG.

Now, we consider some use cases of LMS case study. Figure 7.3 shows the SD of the use case *Remove Title* of LMS. The actor for this use case is the Librarian and the input is the *ISBN* of the book. The pre-conditions for the use case are: (i) the title is not in reserved condition and (ii) no item of the title is issued by any borrower. The post-condition is: (i) the title will be removed from the library database along with its all items, after the successful execution of the use case. Now, we get the number of linearly independent paths within the use case. The CCFG of the SD of the *Remove Title* use case is shown in Figure 7.4. In the figure, Dn stands for

Decision node. There is no asynchronous message or parallel message sent by any object. Hence, there is no concurrent sub-path in a path. All possible paths in the CCFG are:

$\rho 1$=1, 1.1, 1.2, 1.2.1A, 1.2.1A.1, 3, 3.1

$\rho 2$=1, 1.1, 1.2, 1.2.1, 1.2.1.1, 1.2.1.1, 1.2.1.1.1, $(1.2.1.1.2)^*$, 2, 2.1, 2.1.1A, 2.1.1A.1, 3, 3.1

$\rho 3$=1, 1.1, 1.2, 1.2.1, 1.2.1.1, 1.2.1.1, 1.2.1.1.1, $(1.2.1.1.2)^*$, 2, 2.1, 2.1.1, $(2.1.1.1)^*$, 3, 3.1

The nodes $(1.2.1.1.2)^*$ and $(2.1.1.1)^*$ are repeated nodes in path $\rho 2$ and $\rho 3$. The symbol $(*)$ represents the value ≥ 1. The node 1.2.1.1.2 shows that once the given *ISBN* matches with the existing *Title ID* in the database, all the items of that title are displayed to the user. When a user deletes the *Title ID*, all items of that title are destroyed. So, the node 2.1.1.1 is executed for each item of the title. For simplicity, we consider the repeated nodes only once in a path at the time of computing the complexity of the path. As there is no concurrent node in any path of the CCFG, the weight of each node is 1. The complexities of $\rho 1$, $\rho 2$ and $\rho 3$ are 7, 14 and 14, respectively. The total complexity of the use case *Remove Title* is $7 + 14 + 14=$ **35**.

Next, we discuss the concurrent path in a CCFG through an example. For this, we discuss another use case *Issue Item* of LMS. Figure 7.5 shows the SD of the use case *Issue Item.* The actor for this use case is the Librarian and the inputs are *UserID* and *BookID (Title ID).* The pre-conditions for this use case are: (i) the borrower is eligible to issue a book and (ii) at least a single item of the given Book Title is in available state. The CCFG of the use case *Issue Item* is shown in Figure 7.6. The possible paths are:

$$\rho 1 = Start, \begin{pmatrix} m0, m1, m2, m3 \\ m5, m6, m7, m8 \end{pmatrix}, Stop$$

$$\rho 2 = Start, \begin{pmatrix} m0, m1, m2, m3 \\ m5, m6, m7, m8 \end{pmatrix}, m10, \begin{pmatrix} m11 \\ m13, m14, m15 \\ m17, m18, m19, m20 \end{pmatrix}, Stop$$

$$\rho 3 = Start, \begin{pmatrix} m0, m1, m2, m3 \\ m5, m6, m7, m8 \end{pmatrix}, m10, \begin{pmatrix} m11 \\ m13, m14, m15 \\ m17, m18, m19, m20 \end{pmatrix}, \begin{pmatrix} m22, m23, m24 \\ m25, m26, m27 \\ m28, m29, m30 \end{pmatrix}, m31, Stop$$

In $\rho 1$, there are eight concurrent nodes and two simple nodes. The complexity of $\rho 1$ is $8*5+2*1 = 42$. Similarly, the complexities of $\rho 2$ and $\rho 3$ of use case *Issue Item* are 83 and 128, respectively. The total complexity of use case *Issue Item* is 253. Extra test effort is required to test use case *Issue Item* compared to use case *Remove Title*, as the complexity of the CCFG of *Issue Item* is high.

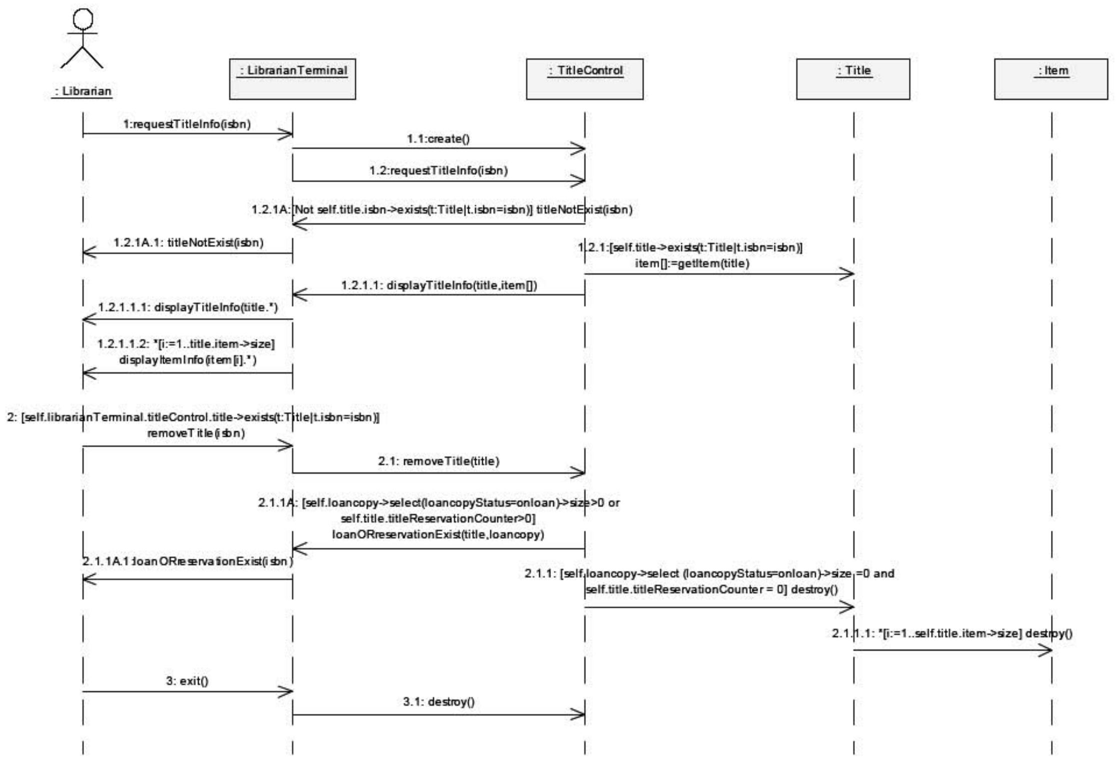

: Librarian
: LibrarianTerminal
: TitleControl
: Title
: Item
1:requestTitleInfo(isbn)
1.1:create()
1.2:requestTitleInfo(isbn)
1.2.1A:[Not self.title.isbn->exists(t:Title|t.isbn=isbn)] titleNotExist(isbn)
1.2.1A.1: titleNotExist(isbn)
1.2.1:[self.title->exists(t:Title|t.isbn=isbn)] item[]:=getItem(title)
1.2.1.1: displayTitleInfo(title,item[])
1.2.1.1.1: displayTitleInfo(title.*)
1.2.1.1.2: *[i:=1..title.item->size] displayItemInfo(item[i].*)
2: [self.librarianTerminal.titleControl.title->exists(t:Title|t.isbn=isbn)] removeTitle(isbn)
2.1: removeTitle(title)
2.1.1A: [self.loancopy->select(loancopyStatus=onloan)->size>0 or self.title.titleReservationCounter>0] loanORreservationExist(title,loancopy)
2.1.1A.1 loanORreservationExist(isbn)
2.1.1: [self.loancopy->select (loancopyStatus=onloan)->size =0 and self.title.titleReservationCounter = 0] destroy()
2.1.1.1: *[i:=1..self.title.item->size] destroy()
3: exit()
3.1: destroy()

Figure 7.3: **SD for use case Remove Title**

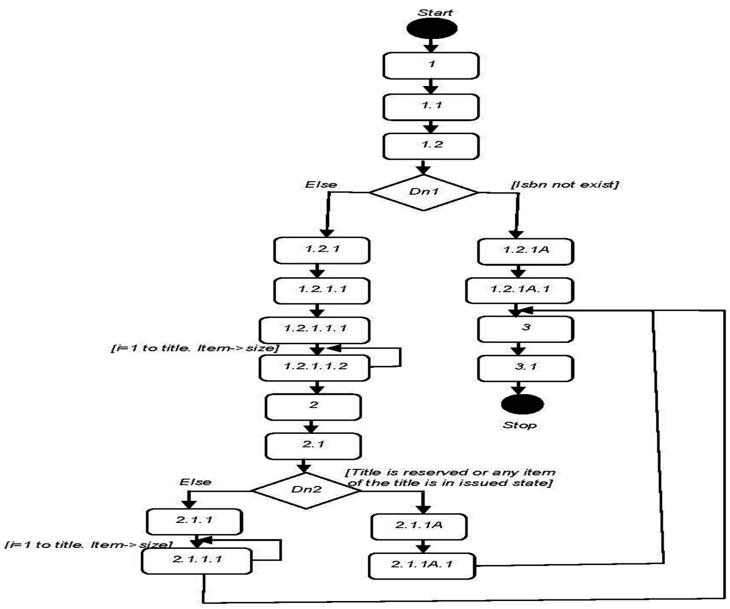


Figure 7.4: **CCFG of Figure 7.3**

7.1.2 Number of Test Paths generated within a SD

We consider all possible test paths within a SD as an influencing factor for complexity computation. Each test path is covered by an individual test case. The amount of test effort required for a use case is decided on the basis of the number of test paths generated within the SD of the use case. We consider the total number of possible

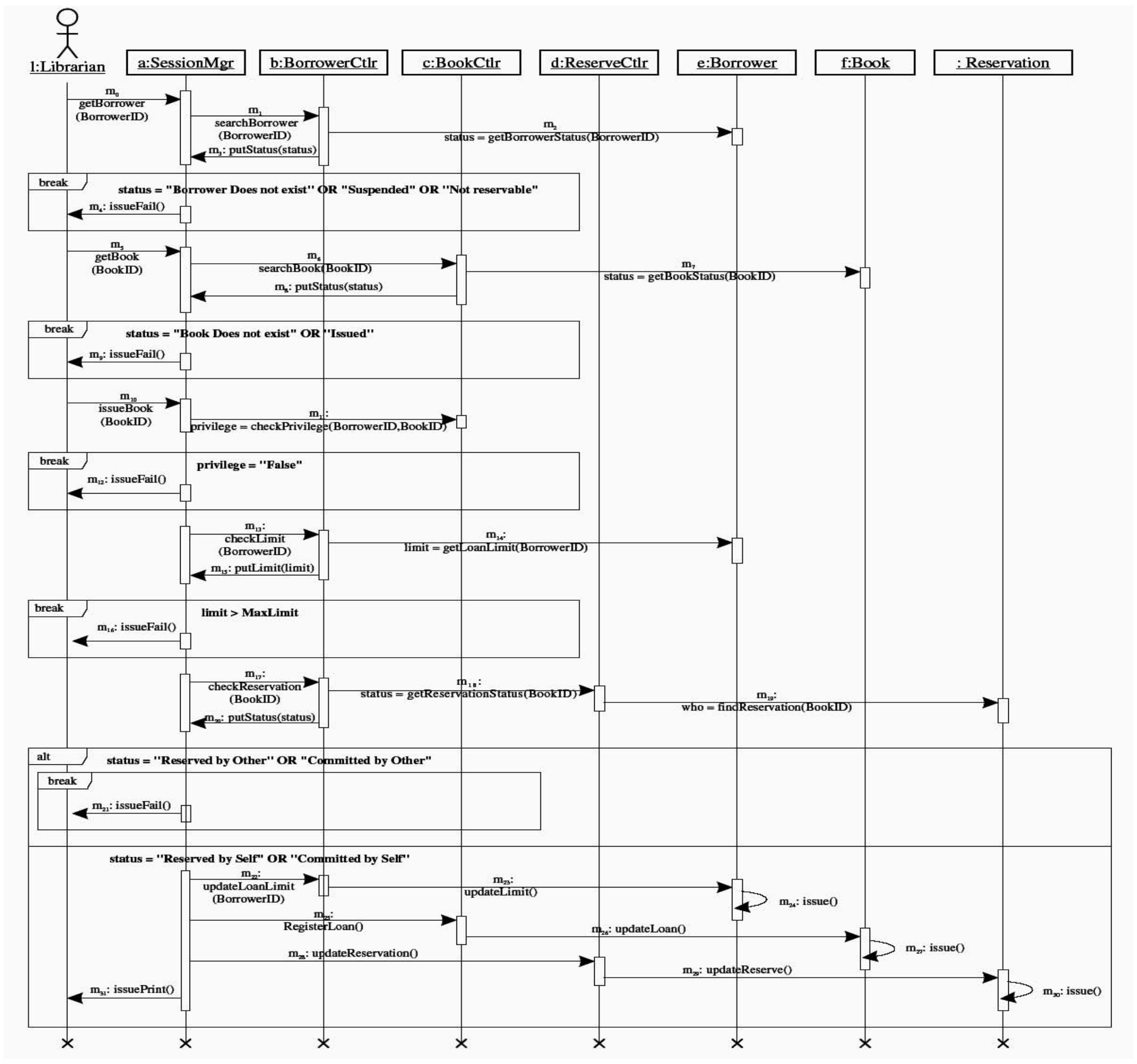

l:Librarian
a:SessionMgr
b:BorrowerCtlr
c:BookCtlr
d:ReserveCtlr
e:Borrower
f:Book
: Reservation
m0 getBorrower (BorrowerID)
m1 searchBorrower (BorrowerID)
m2 status = getBorrowerStatus(BorrowerID)
m3: putStatus(status)
break
status = "Borrower Does not exist" OR "Suspended" OR "Not reservable"
m4: issueFail()
m5 getBook (BookID)
m6 searchBook(BookID)
m7 status = getBookStatus(BookID)
m8: putStatus(status)
status = "Book Does not exist" OR "Issued"
m9: issueFail()
m10 issueBook (BookID)
m11: privilege = checkPrivilege(BorrowerID,BookID)
privilege = "False"
m12: issueFail()
m13: checkLimit (BorrowerID)
m14: limit = getLoanLimit(BorrowerID)
m15: putLimit(limit)
limit > MaxLimit
m16: issueFail()
m17: checkReservation (BookID)
m18: status = getReservationStatus(BookID)
m19: who = findReservation(BookID)
m20: putStatus(status)
alt
status = "Reserved by Other" OR "Committed by Other"
m21: issueFail()
status = "Reserved by Self" OR "Committed by Self"
m22: updateLoanLimit (BorrowerID)
m23: updateLimit()
m24: issue()
m25: RegisterLoan()
m26: updateLoan()
m27: issue()
m28: updateReservation()
m29: updateReserve()
m30: issue()
m31: issuePrint()

Figure 7.5: **SD for use case Issue Item**

transitions covered by each modal class within a SD as an input for getting the possible test paths within the SD. An object of a modal class can receive a message in various states within a scenario and the object may change its state after receiving the message.

Total number of possible transitions covered by a modal class within a scenario is derived on the basis of message sequences generated within the SD and the state chart diagrams of the interacting classes within the scenario. *The aim is to give extra test effort on a use case, in which a number of objects are changing their states by modifying their attributes repeatedly.* Total Number of Test Paths (NTP) generated within a scenario is calculated by taking the product of transitions covered by each interacting modal class [115]. It is given by the following equation.

$$NTP = \prod_{i=1}^{n} NT_i^{x}, \tag{7.1}$$

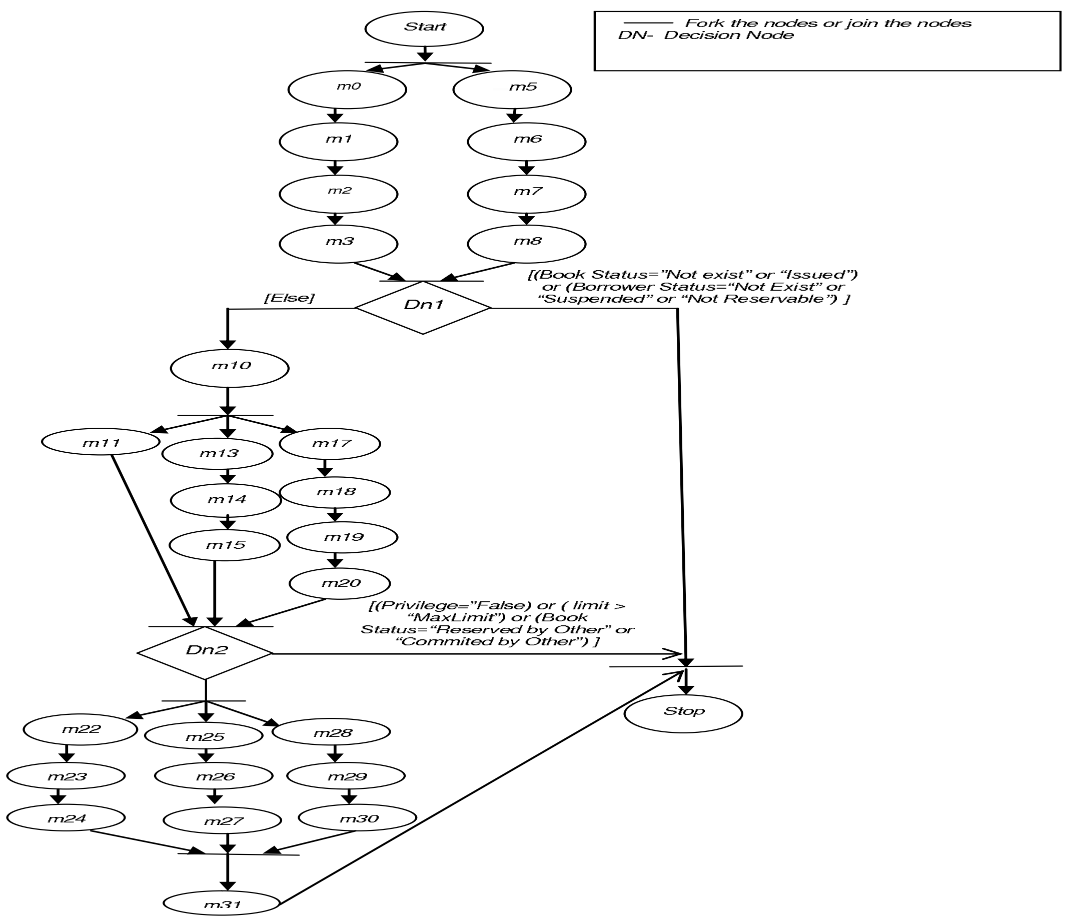
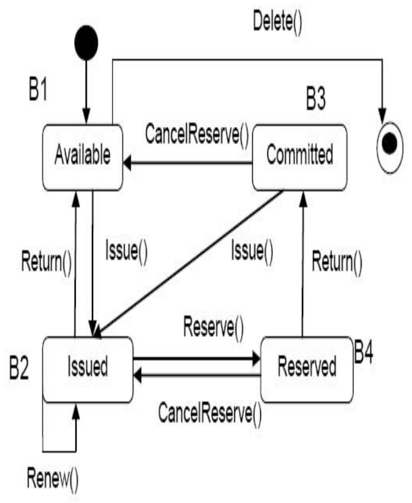

Fork the nodes or join the nodes
DN- Decision Node
Start
m0
m1
m2
m3
m5
m6
m7
m8
Dn1
[Else]
[(Book Status="Not exist" or "Issued") or (Borrower Status="Not Exist" or "Suspended" or "Not Reservable")]
m10
m11
m13
m14
m15
m17
m18
m19
m20
[(Privilege="False) or (limit > "MaxLimit") or (Book Status="Reserved by Other" or "Commited by Other")]
Dn2
Stop
m22
m23
m24
m25
m26
m27
m28
m29
m30
m31

Figure 7.6: **CCFG of the SD of Issue Item use case**

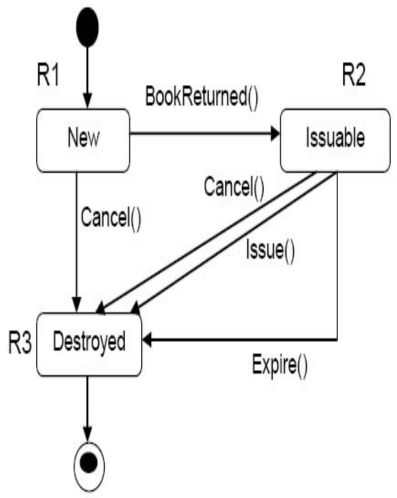
In this equation, n represents the total number of modal classes and NT_i^x represents the total number of transitions covered by i-th modal class within scenario S^x. The total number of possible test paths generated by a use case is obtained by taking the summation of total number of test paths of the SD that implement the use case.

Now, we have to show the total number of test paths generated within the SD of use case *Issue Item*. For this, we should get the total number of state transitions occurred within the SD. To get this, we need the state chart diagrams of the objects that are interacting within the SD of the use case. All the required state chart diagrams are shown in Figure 7.7. For the successful execution of the use case *Issue Item*, the *Book* object may be in the *available* (B1) state or in the *committed* (B3) state at the initial stage. When the *issue()* function is executed on the object *Book* (Refer the SD shown in Figure 7.5 and the state chart diagram of Book shown in Figure 7.7a), its new state will be *Issued*.

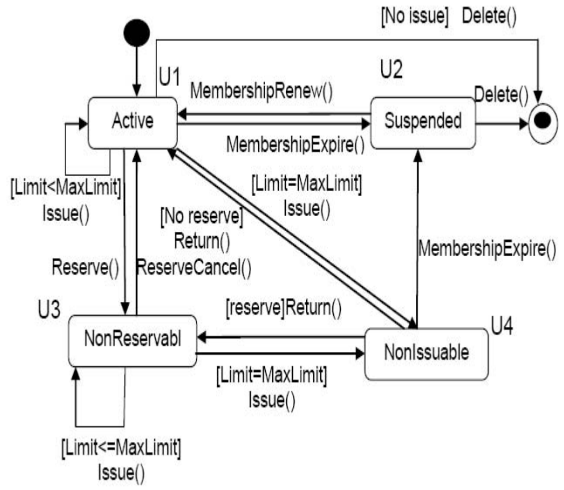
All possible state transitions of various objects in the use case *Issue Item* are shown in Figure 7.8. Let us consider an object of Borrower. At the initial state of



(a) BOOK



(b) RESERVE



(c) BORROWER

Figure 7.7: **State chart diagrams of various objects of LMS**

Table 7.1: Possible transitions of various objects in Issue Item use case

ObjectName	*MessageReceived*	*Transitions#*
Book	issue()	2
Reserve	issue()	1
Borrower	issue()	4

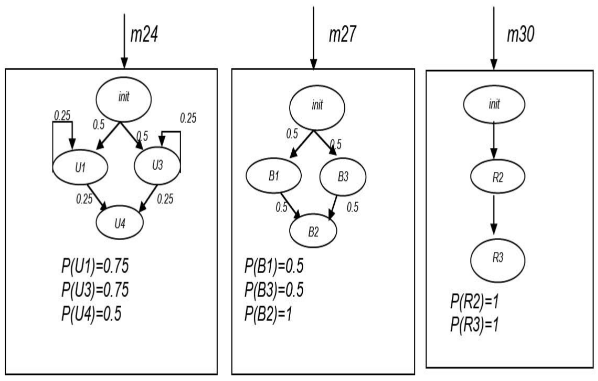


Figure 7.8: **Possible state transitions of objects in *Issue Item* use case**

Issue Item use case, the object of Borrower will be either in U1 or U3 state as shown in Figure 7.8. The possible four transitions in the object are: (i) U1 → U1, (ii) U1 → U4, (iii) U3 → U3 and (iv) U3 → U4. These transitions are extracted by analyzing the SD of the use case and the state chart diagrams of various interacting objects within the use case. Table 8.3 shows the total possible transitions of various objects within the SD of use case *Issue Item*. From the table, we infer that extra test effort is required to Borrower object than others. The total number of possible test paths generated by the successful execution of *Issue Item* use case is $4*2*1 = 8$. Only three messages (m24, m27 and m30) change the states of the interacting objects within the SD of the use case. There is no state transition by other messages in the SD.

7.1.3 Number of Critical Messages transmitted within a SD

There are certain messages within a SD that are critical to the sender [116]. The failure of services for those messages may lead to catastrophic consequences. Therefore, we should check the severity associated with a message within a SD for test effort prioritization. We check how the failure of receiver affects the sender, within a SD. The value returned by the receiver may be used by the sender for taking any important decision. The returned value may be used in some computations in which the inaccuracy may lead to catastrophic consequences. There are some messages within a scenario that are providing exception handling of rare but critical conditions. Though, the execution probability of that messages are low, the failure of any one of them may cause a sever loss to the system. Therefore, we consider the severity associated with a SD through the criticality of messages. We assign severity to a message within a SD on the basis of how the system operation is affected by the failure or incorrect services provided by the receiving object of the message. At the analysis stage, the critical behavior can be identified from domain experts or customers. It is traced to use cases and then to SDs to identify elements of the system that need to be analyzed in depth and need to be tested thoroughly.

For example, Figure 7.9 shows the criticality of messages sent from *FireDetector* and *FireController* with tagged value in an automatic *Fire Controller System*. When the system detects the unwanted fire, it switches off the Oven and inform to the fire department and owner. All three events happen in parallel. The business logic of this system sets the criticality of the message fireAlert() from *FireDetector* to *FireController* to *Very High*, since the failure of this message has a catastrophic impact on the system. FireController sends three messages in parallel. Out of these three, the message switchOff() is *Very High*, then the message MakeCall() to FireDeptt is *High* and MakeCall to Owner is *Low*. The assignment of criticality to a message is subjective. It is decided by checking the impact of the failure of the message on the system. We consider only messages with critical value Very High and High. The complexity of a SD on the basis of critical messages handled is given below.

$$\frac{\#critical\ messages\ transmitted}{\#messages\ transmitted} * 100 \tag{7.2}$$

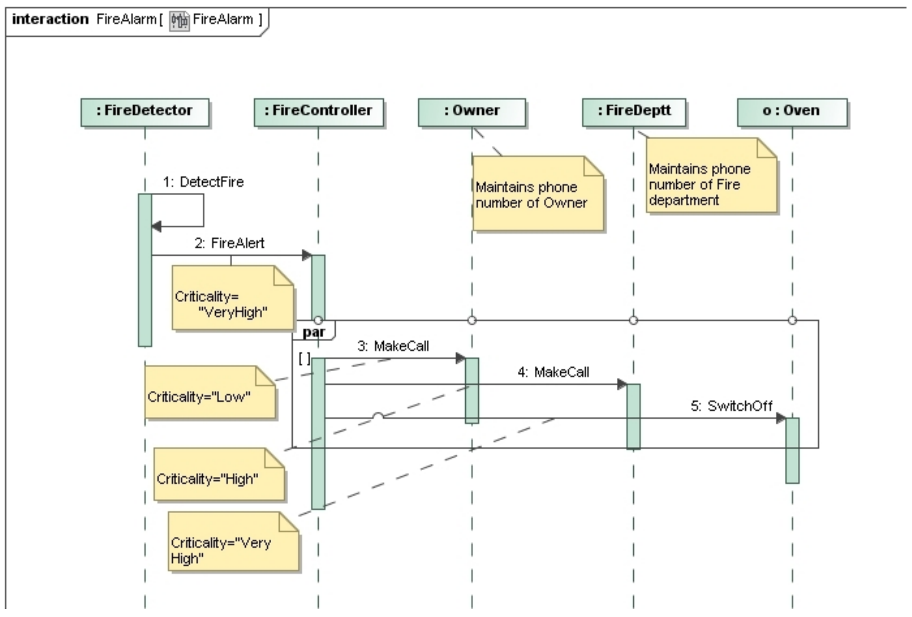


Figure 7.9: **An example of a SD showing message criticality in a Fire Controller system**

7.1.4 Number of Operational Variables used

An operational variable determines the response of a use case by providing systematically the information required for test design. An operational variable can be an explicit input/output, environmental condition and/or abstraction of state of a system under test. For example, the different states of an ATM can be ready, out of cash, out of service etc. These variables play a great role at the time of testing a use case. Test cases are designed based on operational variables. For example, consider the *establish session* use case of ATM system. To establish a new session, a customer has to insert an ATM card into the card reader slot of the machine. The card reader reads the inserted card (If the card reader cannot read the card, it ejects it. An error message is displayed, and the system aborts the session). The system asks the customer to enter the pin. If the pin is valid then, a connection is established with the bank and the customer is allowed to perform transactions by selecting from a menu of possible types of transaction. In the use case (*establish session*), there are four operational variables used. These variables are *encoded pin in the card*, *entered pin by the customer*, *response of customer bank* and *state of customer account.* At the time of testing, a tester first considers each possible operational variable at every step of use case and determines the domain for each variable. Then, the relationship among different variables are set by the help of a *decision table* to model the response of a system. Each row in a decision table is called a **variant**. Table 7.2 shows the decision table for use case *establish session* of ATM. In the table, *DC* stands for *Don't Care* condition. At testing phase, every variant for a use case should be exercised at least once. For a test suite, the *minimal coverage* metric is given by:

$$MinimalCoverage = \frac{\#variants\ tested}{\#variants} * 100. \tag{7.3}$$

So, a use case with a number of operational variables require a number of test cases to test the boundary condition of each operational variable. The size of a use case is proportional to the size of its decision box. For the use case *Issue Item* of LMS case study, the decision table is shown in Table 7.3. The number of variants in the said use case is *7*.

7.1.5 Length of the longest Maximum Message Sequence (MMS)

For the successful execution of a scenario, it is required to know (i) What other classes might be affected when, one class is not behaving properly or returning wrong value?

Table 7.2: Decision table for use case *establish session*

Variants	Operational Variables				Expected Result
	Card Pin	Entered Pin	Customer Bank Response	State of Account	Messages
1	InValid	NA	NA	DC	Invalid Card
2	Valid	Matches with Card Pin	Ack. from Bank	Closed	Contact your bank manager
3	Valid	Matches with Card Pin	Ack. from Bank	Open	Select a Transaction
4	Valid	Matches with Card Pin	No Ack. from Bank	DC	Please try later
5	Valid	Does nit match with Card Pin	DC	DC	Re-enter Pin

Table 7.3: Decision table for use case *Issue Item*

Variants	Operational Variables				Expected Result
	Book ID	Borrower ID	Borrower Status	Book Status	
1	Not Valid	NA	NA	NA	Book does not Exist
2	Valid	Not Exist	NA	NA	Borrower does not Exist
3	Valid	Exist	Suspended	NA	Membership is expired
4	Valid	Exist	Non-Issuable	NA	Cannot issue further book
5	Valid	Exist	Active	Privilege =False	This book cannot be issued
6	Valid	Exist	Active	Reserved by others	Another person has reserved the book
7	Valid	Exist	Active	Available or Reserved by self	The book is issued successfully

(ii) What is the minimal set of classes within a SD that are responsible for a frequent failure of the use case?
Suppose, there are two events *e1* and *e2* generated by two different objects within a scenario. The object generating event *e2* is dependent on the object generating event *e1*, if and only if there exists an execution path, in which triggering event *e1* makes the event *e2* to trigger, either directly or indirectly. At the time of testing an event, we have to test the events which are in the dependence set of that event. We identify the interaction faults through the dependence set of an event. For this, it is required to know the transitive dependencies among objects within a SD. It is easy to detect a fault in a direct method call, but difficult in indirect case. This indirect dependency can be extracted from the flow of messages within a SD. The flow is well understood from message sequences.

First, we define a Message Sequence (MS) within a SD. Then, we define *Maximum Message Sequence (MMS)*. A MS is a concurrent sequence of messages (call message or reply message) within a SD having the first message is a synchronous call and the last one is the reply message corresponding to the first one [116]. A MMS is a message sequence that is not a subsequence of any other message sequence within the SD [116]. All possible MSs for the SD shown in Figure 7.1b are {m1, r1}, {m4, r4}, {m2, m3, m4, r4, m5, m6, r5, r2} and MMSs are {m1, r1}, {m2, m3, m4, r4, m5, m6, r5, r2}. The MS {m4, r4} is not a MMS because it is included within another MS. In the given SD (see Figure 7.1b), the length of the longest MMS is 8 ({m2, m3, m4, r4, m5, m6, r5, r2}). There exist *context-sensitive dependencies* among objects within a MMS which show both direct and indirect interactions.

The longest MMS within the SD of use case *Issue Item* (see Figure 7.5) is as follow.

$$mms = m10, \left(\begin{array}{c} m11 \\ m13, m14, m15 \\ m17, m18, m19, m20 \end{array} \right), \left(\begin{array}{c} m22, m23, m24 \\ m25, m26, m27 \\ m28, m29, m30 \end{array} \right), m31$$

In this MMS, there are two fork nodes. The longest sub-path in the first fork node is *4* (m17, m18, m19, m20) and in the second fork node, each concurrent sub-path has equal length of *3*. Hence, the longest MMS is *9* for the said use case. A MMS with high value indicates that the dependency among objects is high. A fault in one can be easily infected to other dependent objects, which increases the probability of system failure.

7.1.6 Number of External Links used in a SD

In a distributed system, the communication reliability is critical in unsafe environments. It is required to estimate the probabilities of failures for connectors at the analysis stage for an effective testing. Consider the SD shown in Figure 7.1b. Suppose objects *o1*, *o2* are residing in *node1* and object *o3*, *o4* are residing in *node2*. *node1* and *node2* are linked by a network. This is shown through a *deployment diagram* in Figure 7.10. It is assumed that the probability of connector failure is zero for the objects in same node. The probability of connector failure between *o1* and *o2* is zero, whereas there is a probability of connector failure between *o2* and *o3* also *o2* and *o4*. So, the probability of failure is high, when a number of messages are transmitted through connectors in a use case. For the SD shown in Figure 7.1b, out of 10 messages, 3 messages (m3, m5, r5) are transmitted through networks. When the data is transmitted through network, extra test effort is required to check any network problem. So, the complexity of a SD based on number of connectors used within a SD is expressed as Connector Complexity is given by:

$$ConnectorComplexity = \frac{\#messages\ transmitted\ through\ networks}{\#messages\ transmitted} * 100 \quad (7.4)$$

The complexity of SD shown in Figure 7.1b is 3/10 based on the consideration of connectors only. (For simplicity, we have considered equal probability to the failure of each connector.)

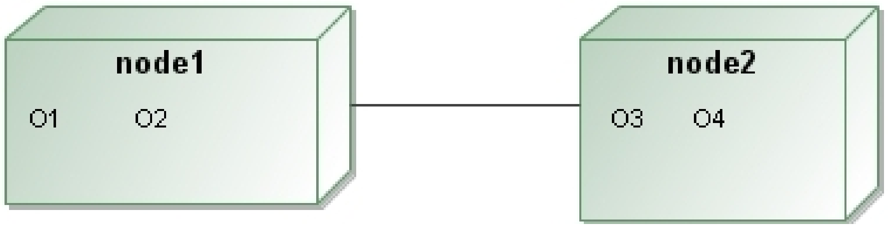


Figure 7.10: **An example of a deployment diagram**

A use case with a number of external links requires extra test effort to test the links.

7.1.7 Number of Polymorphic Calls within a SD

A polymorphic call can be identified within a SD through Class Diagram (CD). A polymorphic behavior occurs at runtime, when the sub-classes override at least one of the method of the base class. *Testing the polymorphic behavior within a scenario requires extra test effort. The complexity of a use case depends on the number of polymorphic messages that are transmitted through the SD of the use case.* The generation of test cases to test a SD which contains polymorphic calls, require manual efforts [117]. In a polymorphic interaction, new test sets are generated for both inherited and overriding methods. The behavior of the program is not predictable due to run time binding, which makes the testing process difficult [117]. Polymorphic interactions are of different types, such as simple polymorphic interaction, parameter-influenced polymorphic interaction and configuration-influenced polymorphic interaction etc.

Simple polymorphic interaction

In this case, the instance of a derived class is directly passed as a parameter. The parameter directly controls the polymorphic behavior. It is easy to test this. The test must consider at least one instance of each class (base and derived classes) as a parameter in the call. An example of a class diagram is shown in Figure 7.11. For

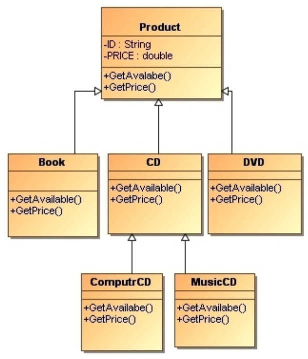


Figure 7.11: **An example of a Class Diagram**

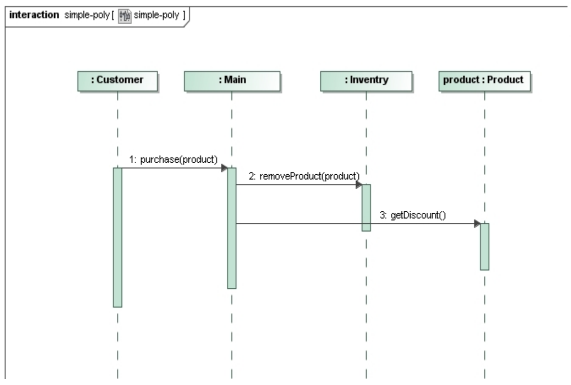
example, consider the SD shown in Figure 7.12a. (Figure 7.12 is taken from [117]). It is simple to determine the test cases for this SD. The possible test cases are an instance of a Book, instance of a ComputerCD, instance of a MusicCD and instance of a DVD.

Parameter-influenced polymorphic interaction

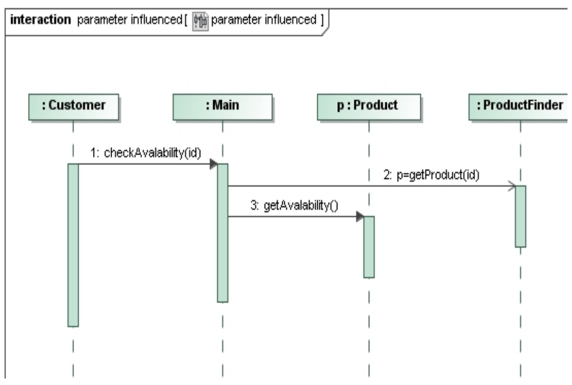
It is explained through an example. Consider the SD shown in Figure 7.12b. The possible test cases are identification number of a Book, a ComputerCD, a MusicCD and a DVD. Comparing to the test cases of Figure 7.12a, these are abstract. For generating the test cases for the SD, extra information is needed to get the identification number for an instance of each sub-class. Manual effort such as the data from domain expert is required to identify the exact appropriate test input values. Due to this, testing this type of polymorphic interaction is less likely to be automated and hence, requires extra test effort.

Configuration-influenced Polymorphic Interaction

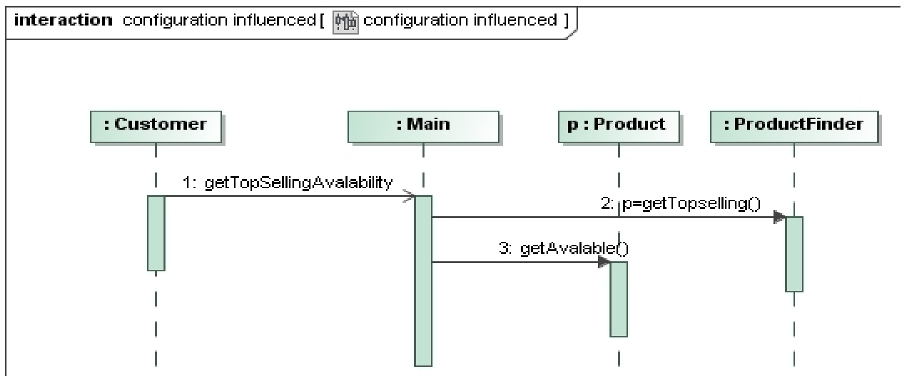
For testing this type of polymorphic call, it is necessary to change the configuration of the system to various states. The initial system state and environment are changed again and again for setting different configurations. Consider the SD shown in Figure 7.12c. The method *getTopselling*() returns an instance of the best selling product, which could be a concrete sub-class of class Product. This polymorphic call is based on the external set up of the system. Parameters of the interaction has no effect on it. The possible test cases for this are obtained by setting the configuration of the system four times. These are setting the top selling product to be a Book, ComputerCD, MusicCD and DVD. Before the execution of each test case, the state of the system needs to be changed to the required configuration state. Testing this polymorphic interaction requires extra testing time and automation of this testing



(a) Simple Polymorphic Call



(b) Parameter-influenced Polymorphic Call


interaction configuration influenced [configuration influenced]
: Customer
: Main
p : Product
: ProductFinder
1: getTopSellingAvalability
2: p=getTopselling()
3: getAvalable()

(c) Configuration-influenced polymorphic call

Figure 7.12: **An example of polymorphic calls**

is difficult. Manual effort is required to set the configuration again and again for testing this type of polymorphic interaction.

7.1.8 Architectural Dependencies among use cases

The use cases of a system can be ordered as per the business logic of the system. In a set of ordered use cases, one use case starts execution after the completion of its preceded use cases. There is a requirement of logical progression for tackling the use case that makes sense to the sequence. If the pre-condition of a use case is same as the post-condition of another use case then, use cases can be ordered to execute sequentially. For example, in LMS case study, a book cannot be deleted if, it is issued. So, for deletion of that book, first *Return Item* use case is called and then, *Remove Title* use case is called. UML-stereotyped association *precedes* is used for this relation. A use case may be followed or preceded by a number of use cases. *Order Flow Graph* shows the dependencies among use cases within a system.

Sometimes, a preceded use case has to execute a number of times to satisfy the pre-condition of a particular use case. For example, consider the LMS case study.

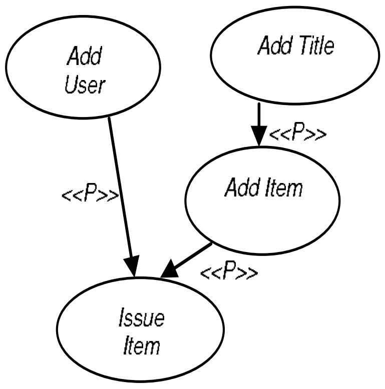


Figure 7.13: **Preceded use cases of use case *Issue Item***

Suppose, the business rule states that a student cannot issue less than five books at a time and the books should belong to different titles. To satisfy this constraint, we have to call *Add Title* and *Add Item* use cases at least five times for testing the use case *Issue Item* (The use cases of LMS are listed in Table 7.4). Similarly, to test a *Return Item* use case, we have to execute *Issue Item* first.

For testing a use case, the tester has to check various pre-conditions of the use case and bring the system to that initial state accordingly. Complication arises when the initial state of a use case requires execution of other use cases in serial/parallel. We draw the architectural dependencies associated with the use case *Issue Item*. Figure 7.13 shows all preceded use cases of use case *Issue Item*. It shows that the tester has to execute the use cases: (*Add User*), (*Add Title*, *Add Item*) at least once, before the execution of use case *Issue Item*. As shown in Figure 7.13, (*Add User*) and (*Add Title*, *Add Item*) can be executed in parallel. A test case which is designed to test the use case *Issue Item* also tests indirectly all the three use cases (*Add User*) and (*Add Title*, *Add Item*) preceded with it, as shown in Figure 7.13. Due to this precedence relationship, *Add Item* requires more test effort than *Add User* and *Add Title* and *Issue Item* use case requires the maximum test effort among the other use cases shown in Figure 7.13.

7.2 Computing Complexity and Test Priority

In Section 7.1, we have discussed a list of eight factors that influence the complexity of a use case. Normally, a weight is associated with each factor, reflecting how much it affects the complexity. In this section, we compute the complexity of a use case and then, assign priority according to its estimated complexity, execution probability and business value.

7.2.1 Computing the complexity of a use case

The factors that affect the complexity of a use case are already discussed in Section 7.1. In this section, we first compute the complexity of a use case based on the above discussed factors. The complexity of a use case U_i is computed as follow.

$$Complexity(U_i) = \sum_{i=1}^{8} W_i * c_i \tag{7.5}$$

In this equation, W_i represents the relative weight and c_i is the estimated value of i-th complexity factor of a use case. The assignment of weights is a subjective matter. It may vary from analyst to analyst. The weight is not static, it may be adjusted and re-calibrated to suit a project's specific needs. The test manager accompanied with key people associated with development is responsible to decide the weight for each complexity factor. Our approach helps to estimate the value for each complexity factor. A value of '0' indicates no influence of the complexity factor on the use case. Once the weight and value for each complexity factor of a use case is decided, the test manager estimates the complexity for the use case by applying Equation 7.5.

Complexity is related to the fault-proneness of a system. To estimate the failure-proneness of a use case, we include its execution probability along with its complexity. We define the *Operational Complexity* (OC) for a use case U_i based on its execution probability p_i and estimated complexity. The OC of use case U_i is:

$$OC(U_i) = Complexity(U_i) * p_i$$

7.2.2 Computing test priority

We consider the Operational Complexity and the Value (business value that comes from customer and market) of a use case along with its estimated complexity for assigning test priority within a system. We compute the *Test Priority* (TP) for a use case within a system by applying the following formula.

$$TP(U_i) = Value(U_i) * OC(U_i) \tag{7.6}$$

In this equation, $TP(U_i)$ is the test priority and $OC(U_i)$ is the Operational Complexity associated with use case U_i. $Value(U_i)$ is the estimated business value of use case U_i. Business value estimation process is discussed in Chapter 2 (Background). The normalized test priority, $NTP(U_i)$, of use case U_i is given by:

$$NTP(U_i) = \frac{TP(U_i)}{\sum_{j=1}^{n}(TP(U_i))} \tag{7.7}$$

Table 7.4: Execution probabilities of use cases with their business values in LMS

Use Case	*EP*	*Value*			
		Relative Weights			
		2	1		
		Benefits	Penalty	Total Value	Value%
Add User	0.05	5	3	13	5.2
Remove User	0.05	2	1	5	1.9
Add Title	0.08	3	2	8	3.2
Remove Title	0.01	1	1	3	1.2
Find Title	0.1	6	8	20	7.9
Add Item	0.01	4	3	11	4.4
Remove Item	0.01	2	3	7	2.8
Make Reservation	0.12	8	9	25	10
Check Reservation	0.1	8	9	25	10
Remove Reservation	0.12	9	9	27	10.8
Search User	0.07	5	8	18	7.2
Issue Item	0.1	9	9	27	10.8
Renew Item	0.03	2	1	5	2
Return Item	0.1	7	9	23	9.1
Find Loan	0.02	5	4	14	5.6
Collect Fine	0.03	6	8	20	7.9
SUM	1	82	87	251	100

EP: Execution Probability.

In equation 7.7, n represents the total number of use cases within the system.

Once the total test cases T for a system under test is decided by the testing team, the number of test cases will be allocated to a use case U_i is $NTP(U_i) * T$.

We have implemented our approach on LMS. The use case diagram of LMS is already shown in Figure 4.1c. Various use cases of LMS with their execution probabilities are shown in Table 7.4. For each high-level function (use case), we collected the information from the users such as the librarian, the library-incharge and students regarding the benefit of implementing the function and the penalty associated with not implementing that function. We have identified sixteen use cases in the LMS case study. We have developed a prototype tool called *Complexity Factor Estimator (CFE)* for automating three influencing factors (*factor 1*, *factor 2* and *factor 5*) out of the discussed eight factors in Section 7.1. Other factors are estimated manually. CFE is implemented using Java. The input to our tool, CFE, are the SD of a use case and the state chart diagrams of all interacting components within the SD. The design artifacts are produced in MagicDraw [118]. First, the UML diagrams are exported in XMI format through an existing XMI parser and then, the XMI format is taken as an input to our tool, CFE. The high level design of our tool, CFE, is shown in Figure 7.14. As shown in the figure, the main modules of our implemented tool are *XMI parser*, *CCFG generator*, *Concurrent path identifier*, *Test path generators* and *Message dependency identifier*. The module *CCFG generator* generates the CCFG of a given SD and the module *Concurrent path identifier* identifies all the possible

Table 7.5: Complexity estimation for use case *Issue Item*

Factors :	*CCFG*	*NP*	*#CM*	*#NV*	*LMMS*	*#ELs*	*#POLY*	*#AD*	TC	OC	
Weight	1	1	1	0.5	0.5	0.25	0.25	0.5	-	-	CCFG:
Value	253	10	0	7	9	0	0	3	272.5	27.25	

The complexity of the CCFG, NP: Number of Transition Paths, CM: Critical Messages, NV: Number of Variants, LMMS: Length of longest MMS, EL: External Links, POLY: Polymorphic Messages, AD: Architectural dependency, TC: Total Complexity and OC: Occurrence Complexity.

paths within the CCFG. Another module, *Test path generator* extracts all the possible transitions covered by a component within a path of the SD. A path within the SD of a use case represents a scenario of the use case. The module *Message dependency identifier* extracts the direct and indirect dependencies among interacting objects within a SD.

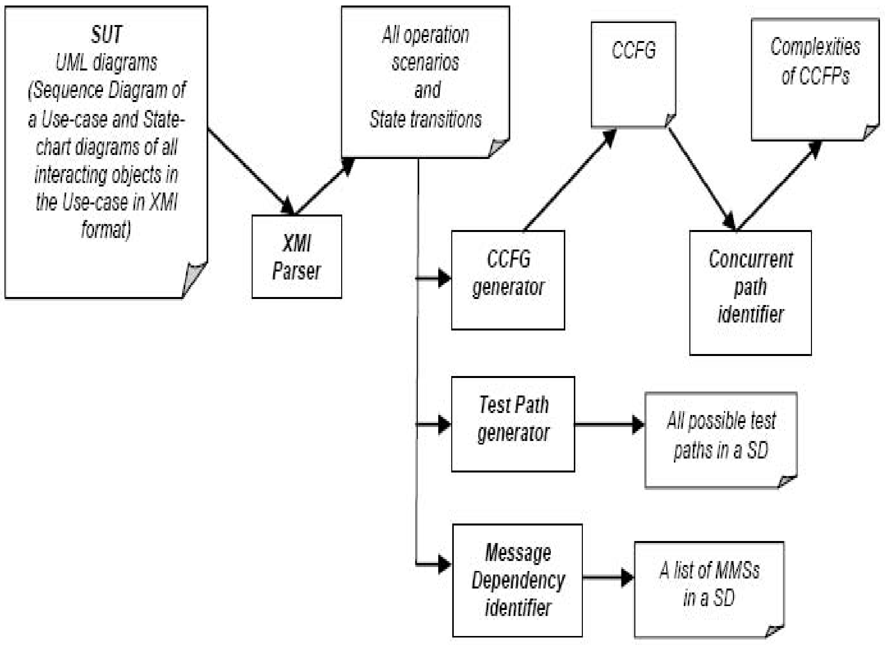

SUT
UML diagrams (Sequence Diagram of a Use-case and State-chart diagrams of all interacting objects in the Use-case in XMI format)
XMI Parser
All operation scenarios and State transitions
CCFG
Complexities of CCFPs
CCFG generator
Concurrent path identifier
Test Path generator
All possible test paths in a SD
Message Dependency identifier
A list of MMSs in a SD

Figure 7.14: **High level design of CFE Tool**

We illustrate our complexity computation method on use case *Issue Item*. We compute the complexity for the use case *Issue Item* by applying Equation 7.5. Table 7.5 shows the estimated complexity for the said use case. In the table, we have assigned different weights to different complexity factors. The weights assigned to complexity factors is a subjective matter and vary from application to application. We have broadly categorized the influences of the factors on the complexity of a use case as high influence, average influence and low influence with weights 1, 0.5 and 0.25, respectively. For any application, the discussed first three factors in Section 7.1 have high influence. The values of various complexity factors shown in second row of Table 7.5 are already estimated in Section 7.1. The estimated complexity for

Table 7.6: Priority calculation

Sl. No	*usecase*	*NTP*
1	Add User	0.002731872
2	Remove User	0.001996368
3	Add Title	0.002689843
4	Remove Title	0.002689843
5	Find Title	0.044270336
6	Add Item	0.00439201
7	Remove Item	0.004952394
8	Make Reservation	0.172318081
9	Check Reservation	0.0420288
10	Remove Reservation	0.097590874
11	Search User	0.007060838
12	Issue Item	0.363128834
13	Renew Item	0.014079648
14	Return Loan	0.227883657
15	Find Loan	0.000980672
16	Collect Fine	0.011205929
	Sum	1

NP: Normalized Priority;

use case *Issue Item* is **272.5**. It is obtained by applying Equation 7.5. We consider the execution probability of the use case and estimate the Occurrence Complexity (OC) in the last column of Table 7.5 as $272.5 * 0.1 = 27.25$. We apply Equation 7.6 and calculate the priority for a use case based on its two important factors: (i) Operational Complexity and (ii) Value. Finally, we compute the normalized priority for various use cases of LMS using Equation 7.7, which are shown in Table 7.6. Figure 7.15 shows the contents of Table 7.6 in graphical form. The figure is an input to the test manager for distributing test effort to various use cases at the architectural level, so that testing and coding can be conducted simultaneously. From Table 7.6, it is observed that the use case *Issue Item* has the highest priority. A use case with high priority requires more test effort than a use case with low priority. So, the use case *Issue Item* requires the maximum test effort. Once the total test effort for the system is decided, test manager distributes the test effort among various use case according to their priority values as shown in Figure 7.15. Our approach helps the test manager in distributing the test resources.

7.3 Experimental Studies

In order to verify the effectiveness of our approach, we have carried out a series of experiments on the source code of the LMS. We have seeded 36 number of faults randomly in the source code of LMS. It has been shown that fault seeding is an effective practice for measuring the efficiency of a test method [106]. There is a number of interactions among components in an object-oriented software. So, there are opportunities for integration or interface faults. The seeded faults are of integration

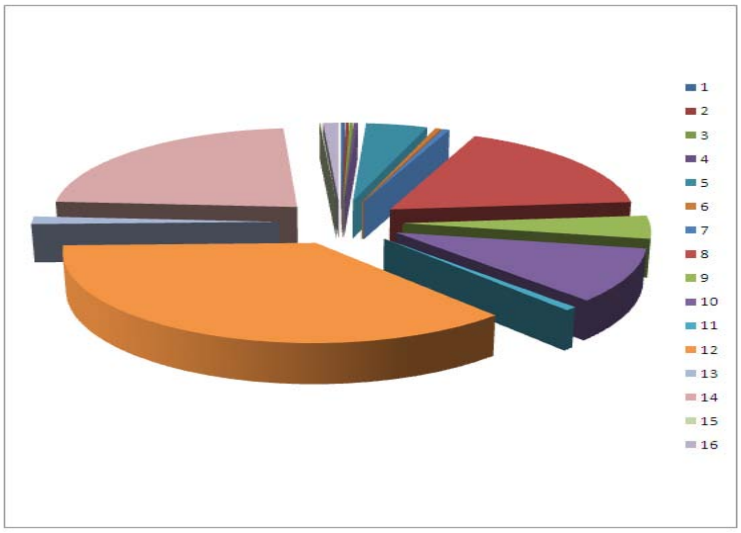


Figure 7.15: **Priority distribution among use cases of LMS case study**

level faults. We assume that a rigorous unit testing has been done by the developers. These seeded faults could not be detected through a rigorous testing at the unit level. The various types of faults that we have considered in our experiment are discussed below.

1. Three types of *interface mutation operators* [107] such as Direct variable replacement operator, Indirect variable replacement operator and Return statement operator are seeded.

2. Six types of *state-based integration faults* [22] were inserted such as Missing transitions, Incorrect transitions, Unspecified event, Incorrect state of the sender object, Incorrect state of the receiver object, Message passing with incorrect/invalid value of arguments. The last one is just explained here. Suppose a message is passed with an incorrect argument or an invalid argument. An object O_i is sending a message $m_i(a1, a2, a3)$. Instead of passing the correct value of a1=x1, it is passed with the value a1=x2, where the value x2 is an incorrect or invalid data. Four number of faults from each discussed type were inserted randomly.

We made three copies of the source code and applied three different types of testing methods. Foe each testing method, the test time was fixed to 36 hours based on the test budget, size of the source code, total number of use cases, total number of classes, total number of scenarios and total number of object-points. The first row of Table 4.2 shows a brief summary of LMS.

We conducted prioritization-based testing based on our proposed use case ranking approach on the first copy of the source code and called it **Ranked** testing. We applied coverage-based testing without any ranking information to the second copy of the source code and called it **Unranked** testing, in which equal importance was given to each use case. We conducted testing based on the operational profile designed for the system to the third copy of the source code and called it **Semi-ranked** testing.

The aim of **Unranked** testing is to cover high percentage of source code and fix as many bugs as possible based on the assumption that fewer bugs are consistent with higher reliability. The aim of **Ranked** testing is to rigorously test the parts of the source code that implement high priority use cases based on the assumption that the reliability will be improved in a higher rate, if high priority use cases will be tested thoroughly. So, in **Ranked** testing, we give effort to a use case based on its priority value as estimated in our proposed approach whereas, in **Semi-ranked** testing, we give test effort to a use case based on its execution probability. It has been shown by many researchers [9, 13, 87] that the user's view on the reliability of a system is improved when, faults which occur in the most frequently used parts of the software are almost removed. Keeping this in view, in **Semi-ranked** testing, we focus test effort on the parts of the source code that are executed frequently. Both in **Ranked** and **Semi-ranked** testing, operational profile is used for testing and the operational profile is accurate as the system is an existing system. Hence, it is assumed that both the testing methods (Ranked and Semi-ranked) could be able to detect the important bugs at the early phase of testing, that are responsible for frequent failures. After the allocated test time was over, we felt that some bugs could not be fixed due to shortage of time, in each copy of the source code. It is because, some bugs were detected during the last stage of testing, that could not be fixed in stipulated time period. Table 7.7 shows the number of mutant bugs detected in three testing methods.

Table 7.7: Testing results of three testing methods

$TestingMethod$	$Mutants\ Killed$	$Mutants\ Fixed$
Ranked	32	28
Unranked	34	30
Semi-ranked	28	27

Figure 7.16 shows the comparison among three different testing methods about the time to detect the defects. We observed that different testing strategies lead

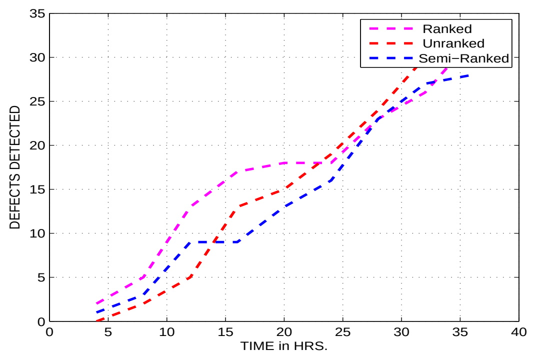


Figure 7.16: **Defect detection rate by various testing methods**

to different testing results. Faults that were detected through Unranked testing method was higher than the faults that were detected through the Semi-ranked testing method. It is because, we iterated a lot in the frequently executed parts of the code and gave less attention to others in this method. Though, Ranked testing method could not detect the maximum number of defects as in Unranked testing method, it detected the maximum number of critical defects as the severity of a message was considered as a factor for complexity computation. It was also found that the fault detection rate in Unranked testing method is nearly linear whereas, in Ranked case, the fault detection rate is high during the early stage of testing. The seeded faults detected in the Ranked testing method was higher than that of the Semi-ranked testing method. The complexity is linearly proportional to defect rates [99] and our approach emphasizes the complexity feature of a use case as one input for ranking. In fact, our aim is to improve the reliability of a system. A software testing method that is efficient in finding faults may not improve the reliability of a system [9, 13, 119]. Our next job is to go for reliability assessment.

The tested source code that were obtained by three testing methods- Ranked, Semi-ranked, Unranked- again tested for reliability assessment. Here, the assumption is that the effect of all types of failures are same, which is practically not true. Some failures have very negative impact on the customer and on the system. A failure could be catastrophic or critical or major or minor [101]. Reliability of a system is assessed by checking how many test cases are executed and out of that, how many test cases failed. In each experiment, we run the three different testing results (tested source code obtained from the three different testing methods) n times according to the operational profile. The value of n varies from experiment to experiment. The defects that caused failures were not fixed and the reliability was estimated.

We have applied two existing reliability assessment methods: (i) random testing (ii) adoptive testing [120]. It is experimentally proved that adoptive testing for reliability assessment is trustable than others [120]. In random testing, test cases are selected randomly from the input domain based on operational profile whereas, in the adoptive testing, the selection of next test case is based on the testing profile[2]. For the reliability assessment, we have made the following assumptions:

1. The code is frozen.
2. A test case either passes or fails.
3. The failure of the software at the current time t is only dependent on the supplied input at that time and is independent of previously executed inputs.
4. The operational profile for the software is $\{I_i; p_i;$ i=1; 2;.;m; $\}$, where p_i represents the execution probability of the input sub-domain I_i, and
$$\sum_{i=1}^{m} p_i = 1.$$
5. Total n number of test cases are allowed to run. The activities that are performed in testing a software for reliability consists of test case selection, test case execution, test result collection and updating the estimated reliability if required. Test case selection is guided by operational profile in random testing and guided by testing history in adoptive testing.

Software reliability R is calculated as follows:

$$R = \sum_{i=1}^{m} p_i \times \Theta_i \tag{7.8}$$

In this equation, p_i and Θ_i represent the execution probability and failure rate of i^{th} sub-domain. Θ_i is computed as follows:

$$\Theta_i = \frac{1}{n_i}\sum_{j=1}^{n_i} z_{ij} \tag{7.9}$$

z_{ij} represents the execution result of a test case which is selected from i^{th} sub-domain for j^{th} time. The value of z_{ij} is 1, if a failure is observed else, the value is 0. n_i is the total number of test cases selected from i^{th} sub-domain and $\sum_{i=1}^{m} n_i = n$. Table 7.8

[2]Testing profile says how to test the software while operational profile says how to use the software [120].

shows the reliability obtained on LMS by applying two testing methods for assessing reliability. In the table, R_{rt} and R_{adpt} represents the reliability assessed by random testing and the reliability assessed by adoptive testing. The tested source code by the three discussed methods: Ranked, Unranked and Semi-ranked are executed for reliability assessment.

Result Analysis

Table 7.8 tabulates the experimental results of software reliability assessed for three discussed methods- Ranked, Unranked and Semi-ranked- based on random testing and adoptive testing. From Table 7.8, we observed that in adoptive testing, the variance is very less compared to random testing. From the reliability values assessed in various experiments, we found that the code tested through *Ranked* method observed the highest reliability and the code tested through *Unranked* method observed the lowest reliability in both the testing methods- adoptive and random testing- for test suites of different sizes.

The observed reliability is the lowest in the code tested through Unranked method. It is because, some residual bugs were found in the frequently executed parts of the tested code. Though, some of these failure causing bugs were detected at the time of testing but, some of these detected bugs were not fixed due to detection at the later phase of testing. This problem though was not observed in the code tested through *Semi-Ranked* method but, the reliability is not high compared to the code tested through *Ranked* method. It is because, the testing was done based on only operational profile. The complexity factor was not considered for testing. Hence, some seeded *state-based integration faults* could not be detected in the semi-ranked testing method.

Though, we have not analyzed the impact of failures on the system at the time of reliability estimation, but it is observed that some serious-failures[3] were observed in the tested source codes through *Unranked* and *Semi-ranked* methods. It is because, we have neither considered the Value associated with a use case nor the criticality of messages within the SD of a use case for testing in *Unranked* and *Semi-ranked* methods. As these two major external factors- Value and criticality of a message- are considered for prioritizing use cases in the *Ranked* testing method, the faults which may cause failures with high negative impact on the user were almost detected

[3]A failure is serious if it causes a heavy financial loss to the organization or it causes a serious damage to the system.

Table 7.8: Reliability assessment based on two different testing strategies

n	$TestedCode$	$TestingStratagies$												
		R_{rt}						R_{adpt}						R_{Final}
		R_1	R_2	R_3	R_4	R_5	R-	R_1	R_2	R_3	R_4	R_5	R-	
100	Code-Ranked	0.818435	0.820048	0.819201	0.830451	0.813092	**0.8202454**	0.827455	0.8279938	0.827343	0.827234	0.827192	**0.827443**	***0.827443***
	Code-Unranked	0.782317	0.7699467	0.771581	0.768901	0.793092	**0.7771674**	0.780118	0.7803604	0.7805426	0.780150	0.780539	**0.780342**	***0.780342***
	Code-Semi-ranked	0.802891	0.7891844	0.827701	0.763190	0.810309	**0.798665**	0.810513	0.810598	0.810484	0.810611	0.810604	**0.810562**	***0.810562***
200	Code-Ranked	0.824505	0.813201	0.810014	0.812574	0.812990	**0.8146568**	0.815344	0.815278	0.815301	0.815076	0.815219	**0.8152436**	***0.8152436***
	Code-Unranked	0.765637	0.789467	0.800158	0.759961	0.771134	**0.777271**	0.764301	0.764241	0.764105	0.764415	0.764339	**0.7642802**	***0.777271***
	Code-Semi-ranked	0.799802	0.800144	0.801577	0.798631	0.810009	**0.8020326**	0.812173	0.812189	0.811984	0.812106	0.811906	**0.8120716**	***0.8120716***
300	Code-Ranked	0.820012	0.811932	0.817309	0.808359	0.816111	**0.814745**	0.813003	0.8131957	0.813095	0.813276	0.813421	**0.813198**	***0.0.814745***
	Code-Unranked	0.797637	0.776596	0.798158	0.801309	0.781134	**0.790967**	0.789573	0.789163	0.789776	0.7893415	0.789234	**0.789417**	***0.790967***
	Code-Semi-ranked	0.800102	0.790144	0.802577	0.814931	0.807009	**0.802953**	0.822509	0.812531	0.812167	0.812116	0.811996	**0.8142638**	***0.8142638***

Code-Ranked:Code tested by Ranked testing; Code-Unranked: Code tested by UnRanked testing; Code-Semi-ranked: Code tested by Semi-Ranked testing method.

R_i: The reliability obtained at i-th run; R-:=$\frac{\sum_{i=1}^{5}(R_i)}{5}$; R_{Final}=max (R_{rt}, R_{adpt}).

and corrected at the early phase of testing through the *Ranked* method.

7.4 Summary

We have proposed a test effort prioritization technique at the architectural level. Our approach is ranking the use cases of an application according to their complexity and business value. For this, we first developed a technique to compute the complexity of a use case quantitatively. Our approach for complexity calculation is purely analytical. For achieving high reliability, the degree of thoroughness with which a use case to be tested is made proportional to its priority value. We have conducted experiments to check the effectiveness of our approach and experimentally proved that assigning test efforts to a use case based on its *execution probability* only, is not sufficient for ensuring the quality of a system. Consideration of both structural and behavioral dependencies within a use case along with its execution probability for computing test priority is a powerful way to improve the quality of the system.

In this chapter, we have not considered the risk associated with a use case. The stakeholder of a software system feels that the measurement of quality of the software system through risk is significant than other factors such as expected number of residual bugs or failure rate etc. Keeping this in mind, we propose a novel risk analysis technique in the next chapter, that works at the software architecture level.

Chapter 8

Analyzing Risk at Architectural Level for Testing

The approach proposed in this chapter is two fold. In the first phase, the risk is estimated for components, use case scenarios and the overall system. In the second phase, risk-based testing is conducted, in which the test priority is assigned to various elements according to their estimated risk.

The existing work on software reliability estimation [9, 14, 15, 57, 63] do not consider the impact of failures observed at the execution of a software system. Due to the availability of design models, stake holders are now getting the opportunity to estimate the reliability quantitatively at the analysis and design stage and hence, the risk associated with the software before its implementation.

Risk analysis is conducted in a software application to assess the damage during use, frequency of use and to decide the probability of failure by looking at defects. There are several types of risks such as reliability-based risk, availability-based risk, acceptance-based risk, performance-based risk, cost-based risk, and schedule-based risk. In the thesis, we are mainly concerned with the reliability-based risk, as our aim is to improve the reliability of a system, within the available test resources. It is the probability that the software product will fail in the operational environment and the adversity of that failure.

In order to save the time and cost in the software development life cycle, there is a requirement of an effective decision-making for allocating resources to various high level requirements. For this, there is a need to assess quantitatively the risk associated with high level requirements as early as possible. Researchers [81, 82, 84] have proposed risk estimation models by gathering data at the requirement stage and analysis stage. As the analysis and design stage is critical compared to other stages, assessing risk at this stage is beneficial to the stake holder. Detecting and correcting errors at this stage is less costly compared to later stages of SDLC. For estimation of risk at an early stage, the important feature is to design a model to predict the dynamic aspects of a system. Un-reliability at different states of a component within a scenario may affect the failure rate of the scenario differently. If the failure probabilities of an interacting component in various states within a scenario are well known, it is easy to analyze their effect on the system behavior. The risk for two different states of a component may vary within a scenario. A fault within a state of a component may be the reason for component failure and the failure of a component/connector may be responsible for a system level hazard [101]. We predict the dynamic aspects of a system and assess risk through the data collected at the detailed design stage. We consider the risk associated with active resources. As connectors are passive[1] in nature, we have not considered any connector faults. It is assumed that connectors are 100% reliable. Unlike the existing work [81, 82] on risk estimation at the architectural level, we introduce the risk associated with different states of a component within a scenario rather than estimating the risk for the component as a whole. Quantifying the risk at different states of a component within a scenario is an input for an effective risk estimation of the scenario.

We propose an intermediate graph called *Inter-Component State Dependence Graph (ISDG)* for getting the complexity for a state of a component. *ISDG* shows intra-component state transitions and inter-component state transitions within a single graph. Once the complexity is calculated, the next step is to analyze the severity associated with various states of a component for risk estimation.

We use the risk associated with various states of an interacting component within a scenario to compute the risk for the whole scenario. The risk for a scenario is estimated based on the estimated risks of the interacting components at various states within the scenario and an existing test model called State COllaboration TEst Model (SCOTEM) [115]. We are primarily motivated by the need to generate a list of scenarios ranked according to their estimated risks. This ranking technique provides a path to find truly critical system functionalities. Assigning test effort to various scenarios based on their estimated risks helps the tester to detect important faults at the early phase of testing. Once the risk for various scenarios within a system are estimated, the risk for the overall system is calculated based on two parameters: (i) estimated risks of various scenarios within a system (ii) list of scenario transition

[1]Passive resources cannot generate their own behavior, but only react to the occurrence of a stimulus, while active resources are those capable of spontaneous unprompted behavior.

probabilities within a system.

The rest of the chapter is organized as follows: Our proposed risk estimation method is described in Section 8.1. The efficacy of our approach is evaluated in Section 8.2 and the summary of the chapter is given in Section 8.3.

8.1 Risk Estimation Method

Our proposed risk analysis method first estimates the complexity associated with individual state of a component. Then, it iterates on a scenario and estimates the severity associated with various states of an interacting component within the scenario. Based on the complexity and severity, it estimates the risk. Our approach estimates the risk for the scenario through the help of an existing state-based integration model called SCOTEM [115] and the estimated risks of various states of the components within the scenario. Our approach helps to carry out a sensitivity analysis for a scenario and generates a list of critical components that are responsible for increasing the risk of the scenario. Finally, we estimate the overall system risk on the basis of risks associated with scenarios and scenario transition probabilities. For calculating the overall system risk, we use Interaction Overview Diagram (IOD) that represents scenario specifications. The procedure of our proposed methodology is shown in Algorithm 2 and a detailed description of the procedure is explained in subsequent sections.

Algorithm 2 Risk Analysis Procedure

```
1: for each component do
2:    for each state do
3:       estimate complexity through ISDG.
4:    end for
5: end for
6: for each scenario do
7:    for each active state of an interacting component do
8:       assign severity.
9:       compute risk.
10:   end for
11:   estimate scenario risk
12:   identify a list of critical components within the scenario through sensitivity
      analysis.
13: end for
14: rank scenarios based on their estimated risks.
15: estimate overall system risk using IOD and scenario risks.
16: identify a list of critical scenarios within the system through sensitivity analysis.
```

8.1.1 Quantifying the complexity for a state of a component

In this section, we propose a method to compute the complexity associated with a state of a component at the architectural level. In a sequence diagram, the interactions among components are represented through event/action pairs. An interaction within a sequence diagram is mapped to an event in a state chart diagram. When, an event is invoked by a component within a scenario, it may trigger an action, which in turn may trigger another event in another component. The event/action interaction describes how invocation of a function in a component affects other components. For example, in the well known case study, LMS, consider a situation when a borrower reserves a book. First, a new object of Reservation component (*New* state) will be created. The newly created Reservation object triggers a message to Borrower object and to Book object, simultaneously. By getting the message, the Borrower object will change its state to *NonResearvable* state from *Active* state (The business rule of our case study says that a borrower can reserve only one book) and the Book object will change its state to *Reserved* state from *Issued* state (Please refer Figure 7.7). Similarly, consider another situation, when a book is returned while it is in *Reserved* state. First, there will be a transition in the Book object from *Reserved* state to *Committed* state. This transition in the Book object triggers the Reservation object for a transition from *New* state to *Issuable* state. Hence, in this situation, a transition in the object Reservation is now dependent on the transition of object Book, but in the previous case, the transition in Book object (transition from *Issued* state to *Reserved* state) was dependent on the transition in Reservation object. The individual behavioral view of various components of LMS are already shown in Figure 7.7. The figure shows the intra-component state transition dependencies but, it is unable to show the inter-component state transition dependencies among components.

Yacoub et al. [70] stated that the dynamic complexity of a component is decided based on the number of transitions of the component within a scenario. They have considered only transitions within a component and computed complexity at the component level. In our approach, we consider both intra-component and inter-component state transition dependencies in a system and compute the complexity at a lower level, for various states of a component rather than for the whole component. For this, we propose a graph at the architectural level called *Inter-Component State-Dependence graph (ISDG)*, through a collection of state chart diagrams of various components within a system. The graph shows both intra-component and inter-component state transitions within a single graph. ISDG shows how a state transition in one component triggers transitions in other components. In the following section, we draw ISDG for our case study LMS and discuss it in detail.

Inter-component State Dependence Graph (ISDG)

We use the concept of Bayesian-model [121] for generating ISDG. It consists of a set of concurrent state machines, SM=$sm_1, \ldots,\ sm_n$ where, n is the number of components in the system. Each state machine (sm_i) consists of a set of states, S= s1,...,sm and a set of transitions T=t1,. . .,tp where, m represents the number of states in the state machine sm_i and p is the number of transitions in the component corresponding to sm_i. Each transition has its origin and destination. There is either a single event or an event/action pair associated with each transition. Each event and action corresponds to an interface of a component. When an event is generated in one component, it may cause a change of state in another component. There is a transition dependency between state machines sm_i and sm_j, if an action of sm_i is matched with an event of sm_j. The ISDG of LMS is shown in Figure 8.1. There

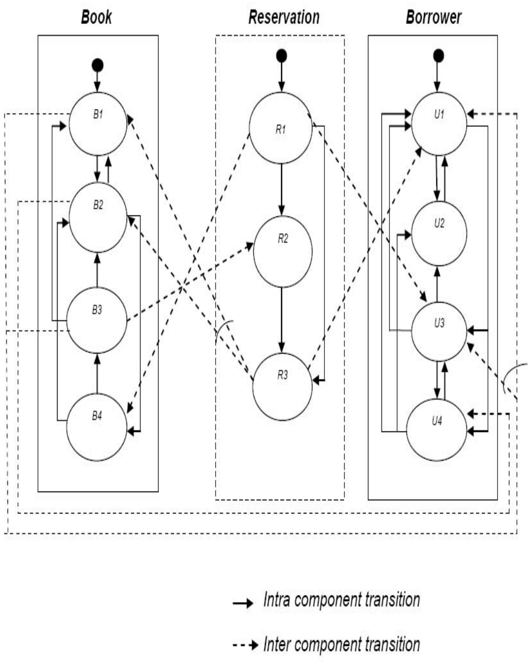


Figure 8.1: **ISDG of LMS**

are two types of edges in ISDG, as shown in Figure 8.1. The solid one shows the *intra-component state transition* and the dashed one shows the *inter-component state transition.* Intra-component state transition can be easily understood from the state chart diagrams of LMS that are shown in Figure 7.7. Now, we discuss one inter-component transition dependency of the ISDG, shown in Figure 8.1. Let us take the state R3 of Reservation object. From R3, there are two outgoing transitions. One is to Book object and the other one is to Borrower object. These transitions say that, when there is a transition in the Reservation object to the state R3 from either the state R1 or R2, it initiates two inter-component state transitions. It sends an event to the component Borrower to change its state to U1 and sends an event to the component Book to change its state to either B1 or B2 depending on the condition. The arc in the two transitions to Book component states that any one transition will occur at a time. Suppose a book is in Reserved (B4) state. A transition in the Reservation object to Destroyed (R3) state will send an event to the Book object to change its state to Issued (B2) state. Figure 7.7 shows various states of a component. The transition of an object of Reservation component to Destroyed (R3) state will cause one transition in an object of Book component to Available (B1) state if the book is in Committed (B3) state. It will be changed to Issued (B2) state if the book is in Reserved (B4) state. The transition in Reservation object is also responsible for a transition in Borrower object. Borrower.U1, Book.B1 and Book.B2 are dependent on the state Reservation.R3. This type of dependency is called *cause and effect* dependency because, the behavior of Reservation component in state R3 implies the behavior of Borrower and Book component. This type of dependency analysis helps to understand the behavior of the system clearly. Hence, ISDG can be used by the developers and testers during development cycle.

Let us explain through ISDG, the behavior of a system to an external event in a given state. For example, let the initial state of LMS be {{B1.4, B2.1}{R1.1}{U1.4, U2.3}} at time $T1$. An object $Oi.j$ represents that i-th object is in j-th state. At time $T1$, it is assumed that the book B1 is already issued to the borrower U1 and it is already reserved by another borrower U2. At this point, there is a chance of occurrence of several scenarios. Let us consider one possible scenario. Suppose the borrower U1 returns the book. The scenario *Return Item* is executed. During the execution, first there will be a transition in object B1 from B1.4 (Reserved) to B1.3 (Committed). Due to the concurrency nature of the three state machines, Book, Reservation and Borrower, this transition of Book causes two inter-component state transitions simultaneously: one transition in the Reservation object R1 and the other one in the Borrower object U1. The state of the Reservation object will be changed from R1.1 (New) to R1.2 (Issuable) and the state of the Borrower object will be changed from U1.4 (NonIssuable) to U1.1 (Active). Hence, after the execution of the scenario *Return Item*, the new state of LMS will be {{B1.3, B2.1}{R1.2}{U1.1, U2.3}}. Our proposed ISDG helps to show the cause and effect dependencies among various components of a system in a single graph. ISDG shows all possible transitions from one state to others. For a particular message, the appropriate one can be detected at the execution of a scenario.

Complexity computation

From the architectural analysis of a software system, we found that a transition in one component may cause a transition in other components. Hence, the reliability of the transition in the second component is dependent on the reliability of the transition in the first component. If a transition to a state Si in a component is dependent on a number of states then, the probability of failure is high in the state Si. We find that the chance of getting failure is high in a state of a component, if it is transition dependent on a number of components. Keeping this in view, we estimate the complexity for a state of a component based on the size of its Dependent-by-Group. Dependent-by-Group of a state Si of a component contains all possible states of the same component (intra-component state transitions) and different components (inter-component state transitions), on which Si is transition dependent. When a failure occurs in a component at state Si, we must check each element of its Dependent-by-Group.

The commonly occurring failures in case of inter-component state transitions are Missing-transitions[2], Incorrect-transitions[3], Unspecified-event[4], Unspecified-state[5], Incorrect-state of sender object[6] etc [22]. As these types of inter-component state transition faults cannot be identified in a rigorous unit testing, there is a chance of increase in failure rate during this transition compared to intra-component state transition. Faults occurring in intra-component state transitions such as Incorrect-

[2]Message m_i is invoked on object O_j, which in turn should trigger changes to object O_k, instead it does not trigger any change to object O_k.

[3]Message m_i is invoked on object O_j, which in turn should trigger changes to object O_k from S_x to S_y, instead it changes the state from S_x to S_z, $S_y \neq S_z$.

[4]Message m_i is invoked on object O_j, which in turn triggers changes to object O_k from state S_x to S_y, which is not specified.

[5]Message m_i is invoked on object O_j, which in turn should trigger changes to object O_k from state S_x to S_y, instead it changes the state to S_z, where S_z is not specified in the design.

[6]A message m_i is to be sent from an object O_j in state S_x to another object O_k, instead the message m_i is sent from object O_j while in state S_y, where $S_x \neq S_y$.

action[7] and Missing-action [8] can be easily identified in a rigorous unit testing. Due to this, we categorize an intra-component state transition as simple with weight 1 and an inter-component state transition as complex with weight 3.

First, we extract the Dependent-by-Group for each state of a component through ISDG (in ISDG, a state is represented as a node). In ISDG, the Dependent-by-Group for a node n contains all the adjacent nodes of n that are connected to node n by its incoming edges. For example, the elements of the Dependent-by-Group of Borrower.U1 are {Borrower.U2, Borrower.U3, Borrower.U4, Reservation.R3, Book.B1}. So, the complexity associated with Borrower.U1 is 1+1+1+3+3=9. Table 8.1 shows the Dependent-by-Group and the complexities associated with various states of Book component in LMS case study. The complexity for state j of component i within a system is represented as $complexity(i.j)$. The normalized complexity is obtained by normalizing the complexity of the state of the component with respect to the sum of complexities of the states for all components within a system. *comp*(i.j) is the normalized complexity for state j of component i. It is computed as follow:

Table 8.1: Complexity for each state of Book component

State	Dependent-by-Group	*Complexity*
B1	{B2,B3,R3}	1+1+3=5
B2	{B1,B3,B4,R3}	1+1+1+3=6
B3	{B4}	1
B4	{B2,R1}	1+3=4

$$comp(i.j) = \frac{complexity(i.j)}{\sum_{i=1}^{n} \sum_{j=1}^{nos} complexity(i.j)} \tag{8.1}$$

In Equation 8.1, n represents the total number of modal components and nos represents the total number of states in i-th component.

8.1.2 Severity analysis

In this section, we use a method based on three hazard techniques [96] to determine the severity associated with various states of a component within a scenario. We are using three hazard techniques: Functional Failure Analysis (FFA), Software Failure Mode and Effect Analysis (SFMEA) and Software Fault Tree Analysis (SFTA) for

[7]A message m_i is sent to object O_k, which in turn should change the state of O_k to S_x, instead the state changes to S_y, where $S_x \neq S_y$.

[8]A message m_i is sent to object O_k, which in turn should change the state of O_k to S_x, instead it does not produce any action.

estimating the severity associated with a state of a component within a scenario. Hazard analysis is done at functional level (top level) through FFA [122]. It shows the possible ways of system failures. First, we identify all possible system level hazards[9]. SFMEA identifies the component level failures and their effect on the system. While FFA needs abstract functional description, detailed architectural design is required for SFMEA. There is a requirement of cause and effect dependency for predicting the likelihood of system failure from the likelihood of component failure [123]. SFTA is conducted to find how the failure of a lower level element is responsible for the failure of an upper level element and finally, the failure of a scenario.

FFA is the first step of severity analysis. The input to a FFA is the list of external events that occur between external actor and the system. For this, we use *System Sequence Diagram* which consists only the messages of a sequence diagram that occur between an external actor and the system [83]. In this case, the system is treated as a black-box. So, the internal events of a sequence diagram that occur among interacting components are not considered. The next step of severity analysis is SFMEA to identify component level failures. It is already discussed in our previous chapters (Chapter 5 and 6). In this chapter, we consider SFMEA at the architectural level instead of code level. SFMEA is done through a detailed analysis of message types in a sequence diagram. Within a sequence diagram, we identify the data transferred between components and the events that are interacting within a use case.

Our severity analysis method at the architectural level consists of the following steps:

1. Performing FFA for a use case.
2. Performing SFMEA: identifying failure modes for various interacting components of the use case through the analysis of sequence diagram and state chart diagrams.
3. Constructing software fault tree using FFA and SFMEA.
4. Converting the results obtained from SFTA and SFMEA into XML files and making a comparison analysis of the resulting XML files for two reasons: (i) to check the consistency between the software fault tree and failure modes of

[9]A system level hazard is associated with all possible hazards at the execution of a scenario. A list of these possible hazards are the outcome of FFA.

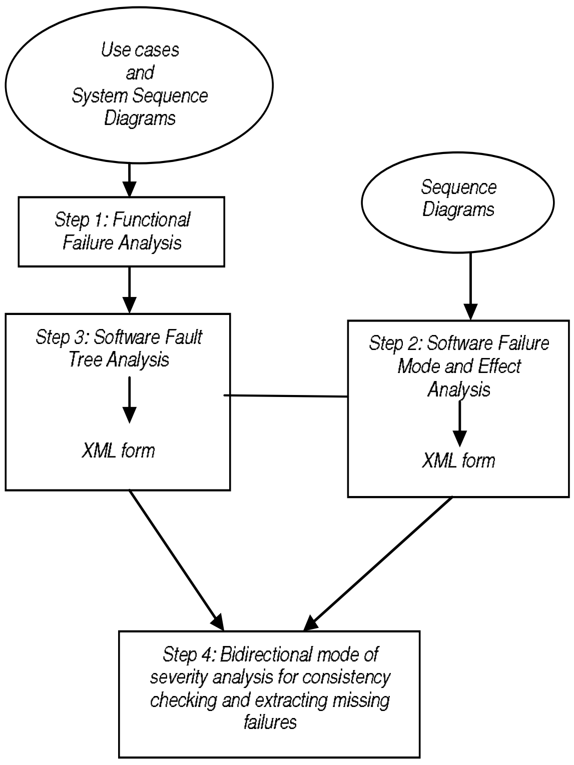

Use cases
and
System Sequence
Diagrams
Step 1: Functional Failure Analysis
Sequence Diagrams
Step 3: Software Fault Tree Analysis
XML form
Step 2: Software Failure Mode and Effect Analysis
XML form
Step 4: Bidirectional mode of severity analysis for consistency checking and extracting missing failures

Figure 8.2: **An overview of the severity analysis method**

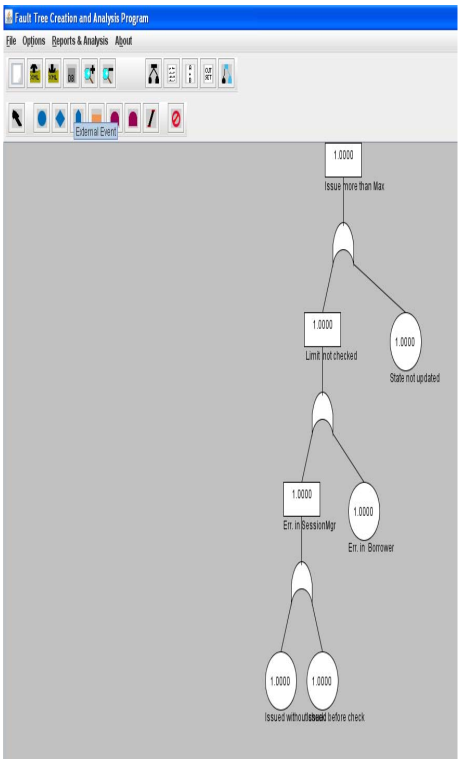
interacting components within a scenario and (ii) to identify any missing failure that is not analyzed.

An overview of our severity analysis process is shown in Figure 8.2. To get all possible failure modes associated with a component within a scenario, we apply bi-directional analysis. A forward search within a scenario is applied through SFMEA and a backward search is applied through SFTA. As SFTA and SFMEA are both complimentary in nature, the combination of both the approaches help us to identify any missing failures in a use case.

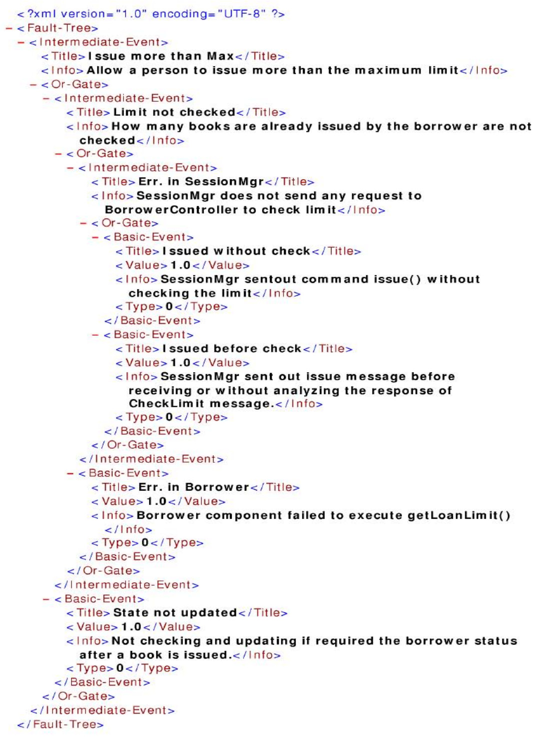
We have created an entire fault tree for each use case of the LMS through the FaultCAT tool [124]. The advantage of this tool is that it allows user to draw and edit the fault tree and calculate the probability of failure of intermediate nodes automatically. It also converts the fault tree into an XML form which helps us to check the consistency with SFTA through Java programs. Let us discuss the use case *Issue Item*. The system level hazards associated with *Issue Item* use case are as follows:

1. Do not allow any borrower to issue book.
2. Allow the non illegible borrower to issue book.
3. Allow the non-issuable book to issue.
4. Allow a person to issue more than maximum number of books.

The root node is "Failure to issue book" and the nodes in first level are the four hazards described above. The second level contains the nodes which contribute to the hazards, etc. We have not presented the entire fault tree due to space reasons. A piece of it with its XML form is represented in Figure 8.3. This is a subtree showing the hazard number 4; the system is allowing a borrower to issue more than maximum number of books. The leaf node of a software fault tree corresponds to a method which is involved in the interaction of the scenario. Hence, the leaf nodes are analyzed using the failure modes from SFMEA. We check whether the failures of events that are associated with a component within a scenario contribute to the errors of the component that lies on the bottom of the fault tree of that scenario. Figure 8.3 shows that either an error in component *SessionMgr* or in component *Borrower* will contribute a system level hazard in the use case *Borrow Item*.



(a) Fault Tree



```xml
  <?xml version="1.0" encoding="UTF-8" ?>
- <Fault-Tree>
  - <Intermediate-Event>
      <Title>Issue more than Max</Title>
      <Info>Allow a person to issue more than the maximum limit</Info>
    - <Or-Gate>
      - <Intermediate-Event>
          <Title>Limit not checked</Title>
          <Info>How many books are already issued by the borrower are not
            checked</Info>
        - <Or-Gate>
          - <Intermediate-Event>
              <Title>Err. in SessionMgr</Title>
              <Info>SessionMgr does not send any request to
                BorrowerController to check limit</Info>
            - <Or-Gate>
              - <Basic-Event>
                  <Title>Issued without check</Title>
                  <Value>1.0</Value>
                  <Info>SessionMgr sentout command issue() without
                    checking the limit</Info>
                  <Type>0</Type>
                </Basic-Event>
              - <Basic-Event>
                  <Title>Issued before check</Title>
                  <Value>1.0</Value>
                  <Info>SessionMgr sent out issue message before
                    receiving or without analyzing the response of
                    CheckLimit message.</Info>
                  <Type>0</Type>
                </Basic-Event>
              </Or-Gate>
            </Intermediate-Event>
          - <Basic-Event>
              <Title>Err. in Borrower</Title>
              <Value>1.0</Value>
              <Info>Borrower component failed to execute getLoanLimit()
                </Info>
              <Type>0</Type>
            </Basic-Event>
          </Or-Gate>
        </Intermediate-Event>
      - <Basic-Event>
          <Title>State not updated</Title>
          <Value>1.0</Value>
          <Info>Not checking and updating if required the borrower status
            after a book is issued.</Info>
          <Type>0</Type>
        </Basic-Event>
      </Or-Gate>
    </Intermediate-Event>
  </Fault-Tree>
```

(b) Fault Tree in XML form

Figure 8.3: **Fault tree with its XML form for a hazard of *Issue Item* use case**

Next, our job is to convert the table obtained through SFMEA into XML format and then, go for a bidirectional analysis to check the consistency between the results obtained through SFTA and SFMEA. Table obtained from SFMEA is converted into a XML file through the help of Java Excel API. The effects of SFMEA are matched with the nodes of the fault tree obtained through SFTA. There are some failures found in the table obtained through SFMEA, that are not exist in the fault tree obtained through SFTA. Let us consider one case. In the table obtained through SFMEA, we have shown a post condition failure (F1) at the time of execution of issue() command in Reservation component. This says that after the execution of issue() command in Reservation component, the state of the Reservation object will be deleted and the state of the Borrower component will be transited from “NonReservable” to “Active” state. In case of a failure F1 (post condition is not satisfied) of issue() method of Reservation component, the object of Reservation component is deleted without making any change in Borrower component. As a result, the borrower will successfully issue a book (there is no problem in current scenario) but could not reserve any book further. As there is no hazard associated with the execution of *Issue Item* use case, the fault is missing in the fault tree obtained through SFTA.

Our methodology considers the message criticality [116] within a sequence diagram, as a parameter for estimating the severity. The weight of each message is not equal. However: (i) there might be certain messages which are critical to the sender objects (ii) some messages (call or reply) might carry larger amounts of data or more parameters (return values) than other messages (iii) the return values of some messages might be used frequently (or for critical decisions/computations) in the caller object than other messages, or (4) some messages might be triggered often than other messages [116]. The job of the designer/analyst is to estimate the weights for various messages of a sequence diagram in an application based on the above four criteria. The severity weight of 0.25, 0.50, 0.75, and 0.95 are assigned to Minor, Marginal, Major, and Catastrophic severity classes respectively, as defined in [81, 82].

8.1.3 Risk computation

In the section, we first estimate the risk for a state of a component within a scenario and then, compute the risk for the whole scenario. We combine the complexity and severity associated with a state to compute the risk for the state. Heuristic risk for a scenario is computed by considering two parameters (i) estimated risk of various states of interacting components within the scenario (ii) SCOTEM [115] of the scenario.

Table 8.2: Estimated risk for various states of *Borrower* component within *Issue Item* use case

Possible state	Risk
NonReservable	0.053
Active	0.089
NonIssuable	0.005

Risk estimation for a state of a component

The heuristic risk, ${hrf_{i.j}}^{x}$, for j-th state of i-th component within scenario S_x is estimated as follow:

$$ {hrf_{i.j}}^{x} = p(i.j)^{x} \times comp(i.j) \times svrty(i.j)^{x} \tag{8.2}$$

In Equation 8.2, $p(i.j)^x$ and $svrty(i.j)^x$ represent the probability and the severity associated with j-th state of i-th component within scenario S_x, respectively. The estimated risk for various states of Borrower component within the main scenario of *Issue Item* use case are shown in Table 8.2. In the table, *Risk* is the normalized risk.

Risk estimation for a scenario

Ali et al. [115] have proposed a state-based integration testing approach for deriving test cases for a scenario. They have proposed an intermediate test model called *SCOTEM* based on state chart diagrams and collaboration diagram. Our risk estimation method uses SCOTEM to estimate the risk within a scenario. An example of SCOTEM for a scenario S_x, $SCOTEM_x$ is shown in Figure 8.4. The Null vertex in Figure 8.4 is a dummy vertex that models an external message (e.g., message from a user). As shown in Figure 8.4, there are four interacting components {C1, C2, C3, C4} in scenario S_x. Each rectangular box represents a component and it contains multiple vertices, where the rectangular vertex corresponds to an instance of the component in a distinct abstract state, corresponding to states defined in state charts and the vertex in ellipse shape represents the starting point, called *init* state. For execution of scenario S_x, all the interacting objects those are modal will be in some specified states. The probability of occurrence of a state in a component box is the sum of probabilities of the paths from *init* node to that state.

For example in our LMS case study, for the successful execution of scenario *Issue Item(U,B)*, the initial state of the requested Book object B will be either in *Available* state or in *Reserved* state and the initial state of the Borrower object U will be either in *Active* state or in *NonReservable* state. We have drawn a SCOTEM for use

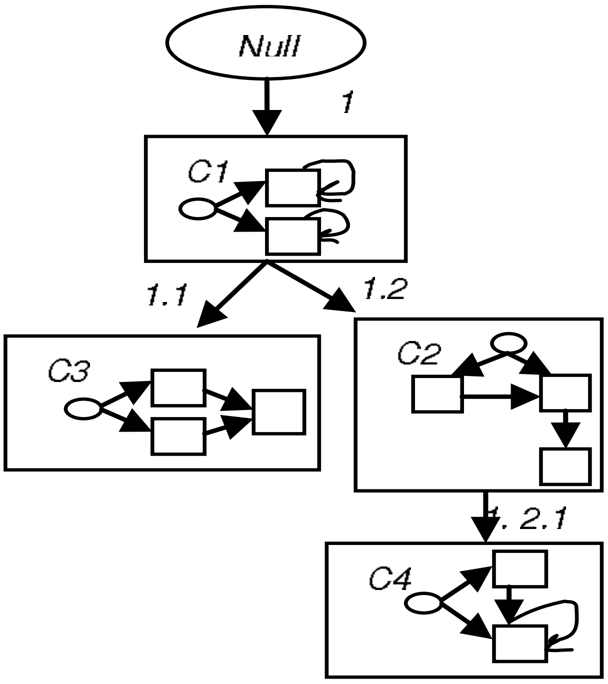


Figure 8.4: **SCOTEM for scenario S_x**

case *Issue Item(U,B)*. The sequence diagram of use case *Issue Item(U, B)* is already shown in Figure 7.5.

In the sequence diagram, only three components Book, Borrower and Reservation are modal. For the execution of messages m1-m21, there will be no change of state of any component. So, in the sequence diagram of use case *Issue Item*, only the execution of three messages m24, m27 and m30 change the state of the system. The behavior of the modal components in the SCOTEM designed for the sequence diagram of *Issue Item* use case is shown in Figure 7.8. Suppose, we consider the message m24. When the message is received, the Borrower object is either in state U1 or U3. We assign uniform probability to each possible state at the initial stage. If it is in U1, then the next state will be either U1 or U4. The probability of a state of a component within a scenario is the sum of probabilities of incoming paths from init state to that state. During the execution of message m24, the probability of U1 is 0.5+0.25=0.75. (init→U1=0.5, U1→U1=0.5, So, init→U1→U1=0.5*0.5=0.25). Similarly, the probability values for various states of other components are calculated.

In a scenario, the risks associated with various states of interacting components are used as input for computing the risk for the scenario. The total number of state paths in $SCOTEM^x$ developed for scenario S_x is determined by taking the product of the number of transitions in each modal component [115]. The possible transitions in various modal components of *Issue Item(U1, B1)* scenario are shown in Figure 7.8. As shown in the said figure, total number of possible transitions by each modal component within the scenario are given in Table 8.3. Hence, the total

Table 8.3: Possible transitions by various objects within *Issue Item* use case

$ObjectName$	$MessageReceived$	$Transitions\#$
Book	issue()	2
Reserve	issue()	1
Borrower	issue()	4

number of state paths in the SCOTEM of *Issue Item(U,B)* scenario is 8. A state path starts with the initial (Null) vertex and contains a complete message sequence of the collaboration. Each state path in $SCOTEM^x$ shows some interactions between the components in appropriate states. We first estimate the risk associated with each state path of $SCOTEM^x$. Risk for k-th state path of $SCOTEM^x$, $(SCOTEM^x)_k$, is estimated as follows:

$$Risk(SCOTEM^x{}_k) = \sum_{i=1}^{n}\sum_{j=1}^{ns}(hrf_{i.j}{}^x) \tag{8.3}$$

In Equation 8.3, n represents the number of interacting components in scenario S_x and ns represents the number of active states of i-th component in k-th path of $SCOTEM^x$. For a large application, number of state paths will be very large. Hence, it will take extra resources to calculate the risk for a scenario. To solve this problem, we have considered *n-Path Coverage criteria* [115]. This coverage criterion selects a specified number n of state paths from the SCOTEM. The value of n ranges from the number of state paths required to achieve *All-Transition Coverage*[10] to the maximum number of possible test paths within an application. The *n-Path Coverage* subsumes *All-Transition Coverage*. It first generates test paths such that each state transition in a modal object is followed at least once. The remaining state paths are then selected randomly until n number of state paths are taken [115]. Then, the risk for the whole scenario is estimated as the sum of the estimated risks of at least n number of paths in the SCOTEM of the scenario, where n is the number of paths considered according to *n-Path Coverage criteria*.

In the sequence diagram of the ongoing example *Issue Item(U, B)* use case, the normalized risk of the main scenario, the successful issue of book, is estimated as 0.3972 using Equation 8.3 and Table 8.2. We have estimated the risk for the successful scenario of each use case. The alternative scenarios of a use case are not considered.

[10]This criteria ensures that each state transition in a modal object is followed at least once.

Table 8.4 shows the estimated risks for various use cases within LMS. From Table

Table 8.4: Risk estimated for various use cases of LMS

Use case	Normalized risk
Add Borrower	0.003
Remove Borrower	0.037
Add Title	0.013
Remove Title	0.009
Find Title	0.005
Add Item	0.009
Remove Item	0.101
Make Reservation	0.069
Check Reservation	0.002
Remove Reservation	0.002
Search User	0.005
Issue Item	0.397
Renew Item	0.071
Return Item	0.179
Find Loan	0.005
Collect Fine	0.093
SUM	1

8.4, it is found that *Issue Item* use case has the highest risk. It is because, a number of state transitions occurred within the scenario.

Estimation of risk for the overall system

We use *scenario-based specifications* as an input for estimating the risk for the overall system. Scenario specification is the composition of a set of scenarios possibly from an user. For details, the reader can refer to [125, 126]. The software industry is widely accepting the scenario specifications as these are well suited for describing the intended behavior of the application in abstract form. Rodrigues et al. [125] have modeled scenario specifications through *Interaction Overview Diagram (IOD).* IOD shows the flow of control among scenarios and the starting state and the end state of the flow which is executed by an average user. In UML 2.0, each activity node of an IOD is a sequence diagram. IOD shows the probability of transfer of control from a scenario to all adjacent scenarios. The transition probability PTS_{ij} between two scenarios represents that the system will execute scenario S_j after executing scenario S_i. Rodrigues et al. [125] have done a sensitivity analysis and made it clear that the system reliability is sensitive to (1) the component reliabilities, and (2) the scenario transition probabilities. Based on this, we use scenario risk and scenario transition probabilities to estimate the risk for the overall system. We have already discussed our proposed method above for estimating scenario risk. The information about scenario transition probabilities are derived from *operational profile* [9] of the system. Each path from starting point to end point in an IOD shows the probability of invo-

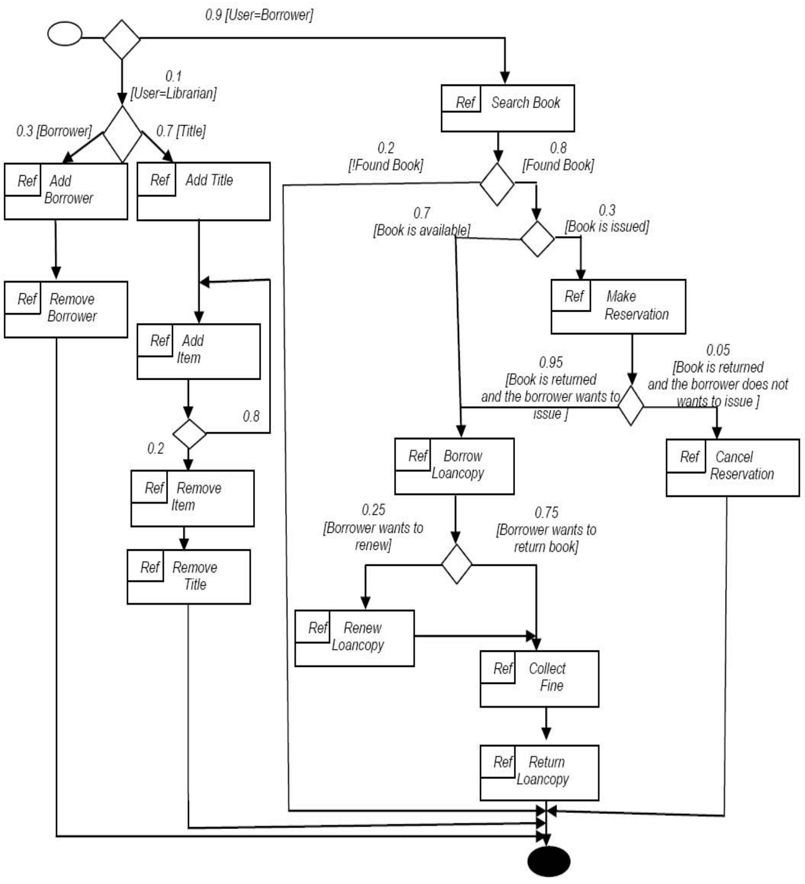


Figure 8.5: **Interaction Overview Diagram of LMS**

cation of a sequence of scenarios by an average user in the operational environment. The risk of the overall system, $Risk(Sys)$, is estimated as follows:

$$Risk(Sys) = \sum_{i=1}^{nop}(Risk(path_i)) \tag{8.4}$$

In Equation 8.4, *nop* represents number of paths in IOD of the system. We have drawn an IOD for LMS in Figure 8.5. In LMS, there are two different types of users: (i) Borrower and (ii) Librarian. According to our assumption, the use of the system by Borrower is nine times more than Librarian. Some use cases are included in other use cases. For example the use case *Collect Fine* is called at the time of execution of the use case *Return Item*, if the copy is returned after the due date. Hence, the included use cases are not explicitly shown in the IOD. We calculate the risk for the overall LMS using Equation 8.4, in which Table 8.4 is used as an input. The estimated system risk for LMS is **0.632**.

Our risk estimation procedure is not amenable to full automation. Construction of ISDG is semi-automatic in our approach. Automatic construction of ISDG is a complex activity in terms of data collection. It is hard to extract all possible state transitions for a non-trivial system. Unfortunately, the severity analysis techniques discussed in the paper are not fully automatic as they involve the user in analyzing the various ways of failures of components / system and determining their effects. The limitation of severity analysis is that SFTA may produce hundreds of combinations of events causing system level failures in a complex system. The analyst / programmer is concerned with what the software is used to perform, whereas SFTA forces the analyst / programmer to estimate the possibility of undesired events within a system and their contribution to system failures. The effort to estimate these may be expensive and time consuming. The skill of the analyst plays an important role for the severity analysis process. Finally, we say that a huge investment is required in order to run our analysis as we require data from a number of UML diagrams and conduct more than one hazard techniques for severity analysis.

8.1.4 Complexity analysis of risk estimation approach

The complexity of our risk assessment procedure shown in Algorithm 2 is dependent on a number of factors such as the number of scenarios, number of modal components, number of states in a component, patterns of component interactions as modeled in the scenario, number of possible hazards associated with the system at various levels. All these factors are not uniform as it varies from application to application. For risk estimation, an intermediate graph called ISDG is introduced in this paper. Now, we analyze the time and space complexity of constructing ISDG.

To determine the space complexity, we consider the space requirement to store the intermediate graph, ISDG. ISDG consists of n number of state machines where, n is the number of modal components in the system. Let state machine sm_i consists of m_i number of states. The value of m_i is usually a small finite number in a moderate size application. Let N be the total number of states in all the state machines of the system. So, $N = \sum_{i=1}^{n}(m_i)$. Each state is represented as a node in ISDG. The transitions in ISDG contain transition to the states of the same component and transition to the states of other components. At the worst case, each node can have n-1 transitions. It is easily realized that the space requirement for ISDG with N number of nodes (states) is $O(N^2)$.

Now, we analyze the time complexity associated with the construction of ISDG. The procedure to construct ISDG takes as input a set of state-chart diagrams and a legal sequence of scenarios. The possible behavior of an interacting component is analyzed at the execution of a scenario. At the time of interaction, we check the initial state of a component. The components should be in some specific states for the occurrence of an interaction. We consider legal sequences of scenarios and analyze the complexity of enumerating all inter-component state transitions of a system. The time complexity of enumerating all possible inter-component state transitions of a system at the design stage is NP-complete.

8.2 Experimental Validation

In this section, we have conducted two experiments to evaluate the efficacy of our approach. The aim of the first experiment is to cross check the estimated risk with the actual failures observed in the system. The aim of the second experiment is to prove that (i) our risk analysis method drives the tester to increase the fault detection rate, (ii) our approach helps in detecting the important faults that are responsible for severe failures. The experiments are conducted on the source code of LMS.

8.2.1 Experiment 1

We performed the following steps in this experiment to cross check the estimated risk with the actual failures observed in the system.

1. Step-1: We applied random testing to test the software. The detected defects were fixed. We recorded the defects found in various scenarios to cross check with the estimated risk.

2. Step-2: We generated a set of test sequences based on the operational profile of the system and executed the tested software for each test sequence. We assumed that the failure rate would be high for a scenario with high risk. As our aim was to check the failure rate of a scenario, in this step, we did not remove any detected defects that were responsible for failures. The failure rate of each scenario was estimated within a test sequence.

3. Step-3: We calculated the average failure rate of a scenario within a system.

The failure rate of a scenario S_j in a test sequence, Seq_i, is Θ_{ji}. Θ_{ji} is computed as follows:

$$\Theta_{ji} = \frac{1}{n_{ji}} \sum_{j=1}^{n_{ji}} z_{ji}, \tag{8.5}$$

z_{ji} represents the execution result of scenario S_j in the test sequence Seq_i. The value of z_{ji} is 1, if a failure is observed else the value is 0. n_{ji} is the total number of times scenario S_j is executed in the i-th test sequence. The following assumptions were taken for the experiment.

1. After the execution of a scenario, the system state may be changed. So, the same scenario may be executed a number of times, but in different system states within a given sequence.

2. A test case that is designed for a scenario either pass or fail. If a test case is blocked, we first, correct the code and then, consider it in our experiment.

3. The output of a selected test case at the current time is not affected by the test results of previously executed test cases.

We executed the system 10 times for 10 test sequences of different length. Table 8.5 shows the estimated risk, detected defects and failure rate of various scenarios in LMS. In this table, We have put the estimated risks of various scenarios for cross checking the detected defects with the estimated risk. In the table, the second column *Risk* is the normalized risk and fourth column *FR* is the normalized failure rate of a scenario, within the system.

Discussion

In Table 8.5, we observed that a number of faults were found in majority of high risk use cases and the failure rate was also high for those use cases. It is because, the number of faults detected within a scenario is related to the complexity of the scenario and complexity is an input for risk estimation. The scenarios within which a number of inter-component state transitions have been occurred were more fault-prone than others. From the table, we observed that the fault detection rate was high in use cases *Issue Item*, *Renew Item* and *Return Item*. We also observed that the failure rate is not linearly proportional to the estimated risk of the system, always. We found that the failure rate is also high for some use cases with low risk. It is because, the third assumption, the output of a selected test case at the current time

Table 8.5: Experimental result for various use cases of LMS

Use case	Risk	DDC	FR
Add Borrower	0.003	0	0
Remove Borrower	0.037	2	0.0001
Add Title	0.013	0	0
Remove Title	0.009	2	0.0004
Find Title	0.005	0	0
Add Item	0.009	1	0
Remove Item	0.101	3	0.0092
Make Reservation	0.069	0	0.0036
Check Reservation	0.002	0	0
Remove Reservation	0.002	0	0.0017
Search User	0.005	0	0
Issue Item	0.397	3	0.0049
Renew Item	0.071	4	0.0019
Return Item	0.179	5	0.0002
Find Loan	0.005	0	0
Collect Fine	0.093	1	0.0038
SUM	1		1

DDC:Defects Detected and Corrected; FR: Normalized Failure Rate of a scenario.

is not affected by the test results of previously executed test cases, may be a threat to the validity of our approach. It is also observed from the experimental result that the failure rate is low for some high risk use cases such as *Return Item* use case. It is because, only failure rate is considered in this experiment, but the risk is actually estimated as a combination of failure rate and severity of failures.

8.2.2 Experiment 2 (Comparison with related work)

We compare our objectives with the existing work on model-based risk analysis techniques according to the six criteria as defined in Table 8.6. In our approach, the smallest individual element for which the risk is assessed is the state of a class whereas, it is a class itself in other two approaches. The other advantage of our approach is that we have done a bi-directional analysis to check the consistency of failure modes in various levels and also extracted any missing failure mode which was not analyzed. We also consider the risk for the whole system on the basis of scenario transition probabilities and risk of scenarios, whereas it is assessed by CDG in [81] and average of risk of use cases in [82]. In Table 8.6, the approach proposed by Goseva-Popstojanova et al. [82] is just an extended version of the approach proposed in [81].

Table 8.6: Comparison of our work with the existing work

CC	Yacoub et al. [81]	Popstojanova et al. [82]	This Work
Source	UML Models	UML Models	UML Models
Approach	Bottom up	Bottom up	Bottom up
Smallest individual item for risk analysis	Component and Connector	Component and Connector	State of a Component
Severity analysis	SFMEA	SFMEA	SFFA, SFMEA and SFTA.
Graph used for assessing scenario risk	(Scenario risk is not calculated)	Discrete Time Markov Chain	SCOTEM
Graph used for analyzing system risk	Component Dependence Graph (CDG)	(Averaging use case risks)	Interaction Overview Diagram.

CC:Comparison Criteria

We have conducted another experiment with the aim to show that our estimated risk and the intermediate results guide the tester in increasing the fault detection rate and detecting important faults. In order to verify the effectiveness of our approach, we have seeded 43 number of faults in the source code of LMS after the completion of unit testing. The seeded faults are of integration level faults. We assume that a rigorous unit testing has been conducted before error seeding. The various types of faults that we were selected in our experiment are discussed below.

1. Three types of *interface mutation operators* [107] are seeded. These are IMO1: Direct variable replacement operator, IMO2: Indirect variable replacement operator and IMO3: Return statement operator .

2. Six types of *state-based integration faults* [22] were inserted such as SF1: Missing transitions, SF2: Incorrect transitions, SF3: Unspecified event, SF4: Incorrect state of the sender object, SF5: Incorrect state of the receiver object, SF6: Message passing with incorrect/invalid value of arguments. These faults are already discussed in above section.

Details of bugs seeded to LMS are shown in Table 8.7. We made three copies of the source code and allocated three different testing approaches for testing and debugging at the higher level. Our aim is to check which testing method is efficient in minimizing the post-release failures and also the types of failures which have a negative impact on both the system and the user. The test time was fixed to 36 hours for each method on the basis of the test budget, size of the source code, total number of use cases, total number of classes, total number of scenarios and total number of object-points.

Table 8.7: Bugs seeded to LMS

IMO1	4
IMO2	4
IMO3	5
SF1	7
SF2	8
SF3	3
SF4	4
SF5	5
SF6	3
Total	43

The first copy was tested by our proposed approach called *State-based Approach* in which the supplied use cases are sorted in a prioritized order according to our calculated risk. The second copy was tested by an approach called *component-based Approach* in which the use cases are sorted according to the risk calculated by the approach of Goseva-Popstojanova et al. [82]. Both the approaches allocated test effort to a use case based on its estimated risk. The third copy was tested by *Randomized Approach*, in which a tester gives equal importance to each use case. For simplicity, we have considered only the main scenario of a use case which shows the successful execution of the use case. Our experiment was aimed at investigating the following queries:

1. Q1: Does our approach guide the tester in improving the test efficiency by detecting more number of important faults than the related approaches?

2. Q2: Does our approach help in improving the test efficiency by increasing the fault detection capability?

3. Q3: Does our approach guide the tester in detecting certain types of faults compared to other two approaches?

Experimental Result and Discussion

The experimental results are shown in Table 8.8. From the table, we observed that the generated test scenarios in our approach uncovered several *state-based integration faults* which could not be detected in other approaches. It is because, we have tested the components which are responsible to change the states of other components during run time whereas, the method proposed in [82] tested the components in which a number of intra-component state transitions occurred during run time. The test priority was assigned to the components based on the number of intra-component state transitions in [82]. Any bug related to that can be easily detected in rigorous unit testing but, it requires extra test effort to identify the bugs related to inter-component state transitions. As state transition concept was not used in Random based Approach, it detected the lowest number of *state-based integration faults.* From

Table 8.8: Mutants killed

MO	SA	CA	RA
IMO1	4	4	4
IMO2	4	3	3
IMO3	5	4	5
SF1	7	5	4
SF2	6	5	4
SF3	3	2	2
SF4	2	2	1
SF5	4	4	4
SF6	3	2	1
Total	38	31	28

MO:Mutation Operator; SA:State-based Approach; CA:Component based Approach; RA: Random based Approach

the experimental result shown in Table 8.8, we answer to the above stated queries Q2 and Q3.

1. Ans2: Yes, our proposed state-based risk analysis approach guides the tester to improve the test efficiency by increasing the fault detection capability.

2. Ans3: Yes, our proposed approach guides the tester in detecting certain types of seeded faults compared to other two approaches. *State-based integration faults* were detected through our approach as the state complexity was taken as one input for risk estimation of a scenario, in which both intra and inter-component state transition dependencies were considered.

To answer the first question, we had gone for another level of testing. After the detected faults were debugged, we run the three tested copies to test their behavior in the operational environment. We used the same test cases for each tested copy. This time the test cases are designed based on only operational profile. We assume that a test case either fail or pass. This time a failed test case is not corrected, only the action is taken to execute a blocked test case. We counted the total number of post-release failures and the impact of those failures on the system and the user. Table 8.9

Table 8.9: Failure observation at the time of release

TC	$Fail_{state-based}$				$Fail_{comp-based}$				$Fail_{Random}$			
	F_{Ca}	F_{Cr}	F_{Ma}	F_M	F_{Ca}	F_{Cr}	F_{Ma}	F_M	F_{Ca}	F_{Cr}	F_{Ma}	F_M
100	**0**	0	2	1	**0**	1	2	0	**0**	3	1	1
200	**0**	0	2	1	**0**	1	4	1	**0**	4	2	3
300	**0**	0	2	3	**0**	2	1	1	**0**	4	1	1

TC:Test Cases; F_{Ca}: Catastrophic failure; F_{Cr}: Major failure; F_{Ma}: Marginal failure; F_M: Minor failure;

shows the result of our risk-based prioritization approach. The failures shown in the table were obtained after the completion of testing phase; at the operational environment. From Table 8.9, we observed that there is no Major type failure observed in the copy of the source code of LMS that is tested by our proposed state-based approach, whereas 2 and 4 numbers of Major type failures were observed in the source code of the LMS that were tested by component-based and randomized approach, when the number of test cases were 300. As shown in the table, Major type failures were also found in the tested copy of component-based approach, though risk analysis was conducted before testing. It is because the dynamic complexity of a component proposed in [82] did not help the tester to detect the state-based integration faults. The example of one such Major type failure observed in the case study LMS is described below.

Though the scenario *Issue Item* was executed successfully when, a borrower requested to issue a book which was already reserved by him however, we observed that the same borrower could not reserve any book further. It is because, due to a seeded bug, the system could not change the state of the borrower to *Active* state from *NonReservable* state, after the execution of *Issue Item* scenario. The severity of this failure is assumed to be Major, as the same borrower cannot reserve any book further.

Now we answer to query Q1. Ans1: Yes, our approach helps to improve the test efficiency by finding bugs that are responsible for severed failures such as Catastrophic and Major types.

8.2.3 Applicability

Risk analysis is a part of safety engineering. Errors related to the temporal behavior of a safety-critical system is hard to detect during testing. These errors may lead to severe failures such as causing severe harm to the life of people or equipments or environment. Our risk analysis approach is mainly applicable for pre-testing analysis of safety-critical systems such as software systems embedded in medical devices, nuclear power station, telecoms systems and industrial robots etc. These embedded softwares are of different sizes and different complexity. The basic principle of a safety-critical system is to keep the system as simple as possible.

Constructing ISDG is complicated and the severity analysis process is time consuming for a large and complex system, but it helps to detect the important errors at the early phase of testing and deliver the product with right quality within the limited budget and time. Our risk analysis approach ranks the components/scenarios within a system for testing according to their estimated risks. There are some components/scenarios with high risk and low execution time. Though their contribution to the overall system risk is less, more testing is needed for that items as they check the exception handling of critical conditions. Our approach also identifies the contribution of a component/scenario risk for increasing the risk of the whole scenario/system through the sensitivity analysis.

8.3 Summary

In this chapter, we have proposed an analytical method for risk estimation of a software system at the architectural level for testing. The approach proposed in the chapter is two fold. In the first phase, the risk is estmated for components, use case scenarios and the overal system. In the second phase, risk-based testing is conducted, in which the test priority is assigned to various elements according to their estimated risk.The data collected from UML diagrams: sequence diagrams and state chart diagrams are used for risk estimation. We have also considered the *operational profile* of the system to know the transition probability between any two scenarios. Compared to the existing work on software risk estimation, our proposed method is a new one that considers (i) risk associated with various states of a component rather than the whole component within a scenario (ii) additional valuable information required for severity analysis of a component such as message criticality and bidirectional analysis to extract possible types of failure modes within a scenario. We have experimentally proved that, testing process is efficient when the testing team is guided by our approach compared to the approach proposed in [82].

Chapter 9

Conclusions and Future Direction

We have explored some test effort prioritization issues at various levels of software development life cycle. Our proposed approaches identified the program's critical paths in which the impact of failure is high. At the implementation level, we have exposed the critical components that are responsible for increasing the system failure rate. At the architectural level, we have proposed novel methods to compute the complexity and risk associated with various high level functions within a system. As our approaches expose the critical elements at the architectural level, the testers and the developers are guided to produce a high quality software, within the available test resources.

9.1 Contribution

In this section, we summarize the important contributions of our work. There are five important contributions: (i) Computing the influence of a component toward system failures (ii) Computing the criticality of a component using both internal and external factors (iii) Improving the software quality using a multi cycle-based testing approach (iv) Estimating the criticality of a use case at the architectural level (v) Estimating the risk associated with various states of a component within a scenario, the risk of a scenario and the risk of the overall system.

9.1.1 Computing the influence of a component

We have proposed a framework to prioritize the components within a system according to their influence toward the system failures. For this, we introduced a metric called *Influence Metric* using forward slicing technique to compute the influence value of a component within a system. It shows the influence of the component toward system failures. We first constructed Extended System Dependence Graph (ESDG), an intermediate representation of an object-oriented program. Then, we presented an efficient algorithm called *MethodInfluence Algorithm* to get the influence value of a method within a system. Our MethodInfluence Algorithm marks a node of ESDG as influenced, when the associated dependency exists. We have shown that the space complexity of our algorithm is $O(n^2)$, where n is the total number of nodes in the ESDG and the time complexity is O(E), where E is the number of edges in ESDG. Influence Metric of a class is obtained by applying *MethodInfluence* Algorithm to all its methods. At the statement level, MethodInfluence Algorithm shows how many other statements are depending directly or indirectly on the output produced by a method within a program. Test Priority (TP) is assigned to a component within the system based on its *influence value* and *average execution time.* We have conducted our experiments on three case studies, LMS, SMA and ATM. We have experimentally proved that decreasing the reliability of a high priority component drastically increases the failure rate of the application, whereas it is not true in case of a low priority component. We have shown that our approach is efficient than the existing approach [9] on test effort prioritization.

9.1.2 Computing the criticality of a component

Prioritizing the program elements within a system based on only influence value and average execution time may not help to expose all the important bugs during testing. So, we have included some important factors for exposing the critical components within a system. We have computed the criticality of a component by adding two external factors: severity associated with each failure and the business value associated with various high level functions within a system and one internal factor: structural complexity, to our previous work. From the experimental results, we observed that by allocating test effort to various components according to their estimated criticality helps in decreasing the failure rate of the application as well as the chance of getting severe failures in the operational environment.

9.1.3 Conducting multi cycle-based testing

We have proposed a multi cycle-based test effort prioritization approach, in which the priority values of various components/scenarios change between test cycles within a system under test. In this work, we introduced the concept of *Influence Metric* through the dynamic slicing approach and used it as one input for prioritizing components within a test cycle in a sequence of many test cycles. From the experimental results, we obtained that the test cases generated through our multi cycle-based testing approach could uncover some important bugs that could not be detected in Musa'a approach [9]. As our approach considered the influence value of a component within a scenario during run time, the components providing a number of services got high test priority. We also assigned priority to the components based on their failure history. These factors helped to improve the reliability of the system under test, within the available test resources. We considered the business value associated with use case scenarios as a factor for prioritization in the third test cycle, which helps to increase the customer certification on the tested system.

9.1.4 Estimating the criticality of a use case

Identification of critical components during the early stage of software development enhances the decisions on test resource allocation. Test efficiency and the quality of software product can be improved, if the test priority of a component is decided at the architectural level rather than at the implementation level. Keeping this in view, we have proposed a novel approach to prioritize the use cases within a system at the architectural level to guide both the tester and developer during software development life cycle. The use cases are prioritized based on their internal criteria- *complexity* and external criteria- *business value*. Unlike the existing approaches on complexity estimation at the early stage, we evaluated the complexity of a use case analytically through a collection of data at the architectural level with little or no involvement of subjective measures from domain experts.

9.1.5 Estimating risk at the architectural level for testing

Risk assessment at the early stage of software development helps achieving high level of confidence in a system and saves the cost and time during software development life cycle. We have proposed a novel risk analysis technique that works at the software architecture level. The main idea is to rank the components within a scenario and to rank the scenarios within a system according to their estimated risk. Unlike the existing work on risk assessment at the architectural level [81, 82], our work assesses risks at a finer granularity level. The efficacy of our approach is evaluated on the Library Management System case study.

9.2 Future Work

We briefly outline the following possible extensions to our work.

1. Prioritization-based testing covers two aspects: (i) prioritizing the program elements for testing and (ii) prioritizing the test cases. In the present work, we concentrated on the first one, i.e. prioritizing the program elements for testing. Automatic selection of test cases from a pool of test cases according to the estimated priority of components can be taken up as a future work.

2. We have considered complexity and failure history as defect generators. The software industry is considering a number of other factors such as change frequency, impact of new technology and impact of the number of people involved, optimization etc. Our proposed method will be effective, if these factors will be considered with it. In future, this scope may be explored.

3. We have proposed eight factors that affect the complexity of a use case at the architectural level. We have automated only three factors. We are planning to automate the rest factors in our future work.

4. We have proposed risk estimation method at the architectural level. One of the future work would be estimating the risk at the requirement phase using requirement models in UML and semi-formal languages.

5. Our approach can be applied to industry standard projects to analyze its effectiveness.

Bibliography

[1] L. Huang and B. Boehm. How much software quality investment is enough: A value-based approach. *IEEE Software*, 23(5):88–95, Sep. 2006. doi:10.1109/MS.2006.127.

[2] R. Mall. *Fundamentals of Software Engineering.* Prentice Hall of India, 3rd edition, 2009.

[3] R. Schach. *Object-Oriented and Classical Software Engineering.* Tata McGraw-Hill, 2002.

[4] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[5] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transaction on Software Engineering*, 28(2):159–182, 2002.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[7] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11):583–594, 1998.

[8] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for Event-Driven software. *IEEE Transactions on Software Engineering*, 37(1):48 –64, Jan. 2011.

[9] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

[10] D. Jeffrey and N. Gupta. Experiments with test case prioritization using relevant slices. *Journal of Systems and Software*, 81(2):196–221, 2008.

[11] J. J. Li. Prioritize code for testing to improve code coverage of complex software. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 75–84, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J. Li, D. Weiss, and H. Yee. Code-coverage guided prioritized test generation. *Information and Software Technology*, 48(12):1187–1198, 2006.

[13] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software*, 7(6):44–54, 1990.

[14] R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. on Software Eng.*, 6(2):118–125, Mar. 1980.

[15] B. Littlewood. A reliability model for systems with markov structure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 24(2):172–177, 1975.

[16] S. Nageswaran. Test effort estimation using use case points. In *14th International Internet Software Quality Week 2001*, June 2001.

[17] J. Capers. *Applied Software Measurement*. McGraw-Hill, 1996.

[18] RoyClem. Project estimation with use case points. http://www.codeproject.com/KB/architecture/usecasep.aspx, 2005.

[19] G. Karner. Metrics for objectory. Diploma thesis LiTHIDA-Ex-9344:21, University of Linkping, Sweden, Dec. 1993.

[20] Q. Li. Using additive multiple-objective value functions for value-based software testing prioritization. Technical report, University of Southern California Computer Science Department, University Park Campus, Los Angeles, CA 90089, USA, 2009.

[21] Q. Li, M. Li, Y. Yang, Q. Wang, T. Tan, B. Boehm, and C. Hu. Bridge the gap between software test process and business value: A case study. In *ICSP '09 Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes*, 2009.

[22] R. V. Binder. *Testing Object-Oriented Systems - Models, Patterns and Tools*. Addison-Wesley, 2000.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[24] M. H. Halstead. *Elements of Software Science*. Elsevier Science Inc., North-Holland, 1977.

[25] M. Weiser. Program slicing. *IEEE Trans Software Engineering*, 10(4):352–357, July 1984.

[26] D. P. Mohapatra, R. Mall, and R. Kumar. An overview of slicing techniques for Object-Oriented programs. *Informatica*, 30:253–277, 2006.

[27] D. P. Mohapatra. *Dynamic Slicing of Object-Oriented Programs*. PhD thesis, Indian Institute of Technology Kharagpur, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, WB 721 302, India, May 2005.

[28] L. Larsen and M. J. Harrold. Slicing Object-Oriented software. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[29] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM.

[30] Y. Song and D. T. Huynh. Forward dynamic Object-Oriented program slicing. In *ASSET '99: Proceedings of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology*, page 230, Washington, DC, USA, 1999. IEEE Computer Society.

[31] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[32] J. T. Lallchandani and R. Mall. A dynamic slicing technique for uml architectural models. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 37(6), Nov. 2011.

[33] Y. N. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, 2002.

[34] Z. Chen, B. Xu, and J. Guan. Test coverage analysis based on program slicing. *Journal of Electronics*, 20(3):232–236, April 2012.

[35] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Scotch: Test-to-code traceability using slicing and conceptual coupling. In *The International Conference on Software Maintenance*, 2011.

[36] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transaction on Programing Languages and Systems*, 9(3):319–349, 1987.

[37] B. A. Malloy, J. D. McGregor, A. Krishnaswamy, and M. Medikonda. An extensible program representation for Object-Oriented software. *SIGPLAN Not.*, 29(12):38–47, 1994.

[38] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 358–367, Washington, DC, USA, 1998. IEEE Computer Society.

[39] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Massachusetts, 1999.

[40] I. Jacobson and M. Christerson. *Object-Oriented Software Engineering: A use case driven approach.* Addison-Wesley, 1992.

[41] S. R. Chidamber and C. F. Kemerer. A metrics suite for Object-Oriented design. *IEEE Transaction on Software Engineering*, 20(6):476–493, 1994.

[42] B. Boehm and L. Huang. Value based software engineering: A case study. *IEEE Computer*, pages 21–29., Mar. 2003.

[43] R. Ramler, S. Biffl, and P. Grnbacher. *Value-Based Management of Software Testing.* ISBN:3-540-25993-7. Springer, Berlin Heidelberg, 2005.

[44] K. E. Wiegers. First things first: Prioritizing requirements. Software Developmen, Sept. 1999. http://www.processimpact.com/articles/prioritizing.html.

[45] C. G. Bai. Bayesian network based software reliability prediction with an operational profile. *Journal of Systems and Software*, 77(2):103–112, 2005.

[46] M. Gittens. *The extended operational profile model for usage-based software testing.* PhD thesis, Faculty of Graduate Studies, University of Western Ontario, 2004. Ph.D Thesis.

[47] S. Ozekici and R. Soyer. Reliability of software with an operational profile. *The European Journal of Operational Research*, 149(2):459–474, 2003.

[48] Software safety. Nasa Technical Standard, Sept. 1997.

[49] A. Mockus, N. Nagappan, and T. T. DinhTrong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[50] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34:497–515, 2008.

[51] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Automating algorithms for the identification of fault-prone files. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 219–227, New York, NY, USA, 2007. ACM.

[52] K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using Object-Oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, Feb. 2001.

[53] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31:340–355, 2005.

[54] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 357–366, Washington, DC, USA, 2011. IEEE Computer Society.

[55] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transaction on Software Engineering*, 18(5):423–433, 1992.

[56] R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for Object-Oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29:297–310, 2003.

[57] M. R. Lyu. Software reliability engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[58] L. C. Briand, W. Jürgen, J. W. Daly, and D. V. Porter. Exploring the relationship between design measures and software quality in Object-Oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[59] D. Kundu, M. Sarma, D. Samanta, and R. Mall. System testing for Object-Oriented systems with test case prioritization. *Software Testing, Verification and Reliability*, 19:297–333, 2009.

[60] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *International Conference on Software Engineering (ICSE)*, pages 412–421, ACM: New York, 2005.

[61] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[62] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[63] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Transaction on Software Engineering*, 20(10):812–824, 1994.

[64] T. Thelin, P. Runeson, and B. Regnell. Usage-based readingan experiment to guide reviewers with use cases. *Information and Software Technology*, 43(15):925–938, Dec. 2001.

[65] K. Goseva-Popstojanova, A. P. Mathur, and K. S. Trivedi. Comparison of architecture-based software reliability models. In *Proceedings of the International Symposium on Software Reliability Engineering,*, Hong Kong,, 2001.

[66] J. H. Lo, S. Y. Kuo, M. R. Lyu, and C. Y. Huang. Optimal resource allocation and reliability analysis for component-based software applications. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, pages 7–12, 2002.

[67] S. Krishnamurthy and A. P. Mathur. On predicting reliability of modules using code coverage. In *proceedings of the 1996 conference of the Center for Advanced Studies on Collaborative research*, pages 22–33, Toronto, Ontario, Canada, 1996.

[68] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, page 146, Washington, DC, USA, 1997. IEEE Computer Society.

[69] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of UML based software models. In *Proceedings of the 3rd International Workshop on Software and Performance*, pages 302–309, New York, NY, USA, 2002. ACM.

[70] S. M. Yacoub, B. Cukic, and H. H. Ammar. Scenario-based reliability analysis of component-based software. *IEEE Transactions on Reliability*, 53(04):465–480, 2004.

[71] T. L. Booth. Performance optimization of software systems processing information sequences modeled by probabilistic languages. *Software Engineering, IEEE Transactions on*, SE-5(1):31–44, Jan. 1979.

[72] J. Smith. The estimation of effort based on use case. IBM Rational Software, 1999. White Paper.

[73] Y. Chen, B. W. Boehm, R. Madachy, and R. Valerdi. An empirical study of eservices product UML sizing metrics. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.

[74] S. Kim, W. Lively, and D. Simmons. An effort estimation by UML points in the early stage of software development. In *International Conference on Software Engineering Research & Practice*, pages 415–421, June 2006.

[75] G. Robiolo and R. Orosco. Employing use cases to early estimate effort with simpler metrics. *Innovations System Software Engineering*, 4:31–43, 2008.

[76] X. Zhu, B. Zhou, F. Wang, Y. Qu, and L. Chen. Estimate test execution effort at an early stage: An empirical study. *International Conference on Cyberworlds*, pages 195–200, 2008.

[77] M. Parastoo, A. Bente, and C. Reidar. Effort estimation of use cases for incremental large-scale software development. In *27th International Conference on Software Engineering, St Louis*, pages 303–311, 2005.

[78] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of System and Software*, 53(3):287–295, 2000.

[79] M. S. Feather, S. L. Cornford, J. Dunphy, and K. Hicks. A quantitative risk model for early lifecycle decision making. In *6th World Conference on Integrated Design and Process Technology*, Pasadena, California, USA, June 2002.

[80] S. L. Cornford and M. S. Feather. DDP: A tool for life-cycle risk management. In *IEEE Aerospace Conference*, pages 441–451, Big Sky, Montana, Mar. 2001.

[81] S. M. Yacoub and H. H. Ammar. A methodology for architecture-level reliability risk analysis. *IEEE Transaction on Software Engineering*, 28(6):529–547, 2002.

[82] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. E. Nassar, H. Ammar, and A. Mili. Architectural-level risk analysis using UML. *IEEE Transaction on Software Engineering*, 29(10):946–960, 2003.

[83] K. Appukkutty, H. H. Ammar, and K. G. Popstajanova. Software requirement risk assessment using UML. In *AICCSA '05: Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, pages 112–115, Washington, DC, USA, 2005.

[84] V. Cortellessa, K. Goseva-Popstojanova, K. Appukkutty, A. R. Guedem, A. Hassan, R. Elnaggar, W. Abdelmoez, and H. H. Ammar. Model-based performance risk analysis. *IEEE Transaction on Software Engineering*, 31(1):3–20, 2005.

[85] C. S. Smidts, Y. Shi, M. Li, W. Kong, and J. Dai. A large scale validation of a methodology for assessing software reliability. Research Paper NUREG/CR-7042, Reliability and Risk Laboratory Nuclear Engineering Program, The Ohio State University Columbus, Ohio, Office of Nuclear Regulatory Research, July 2011.

[86] I. Sommerville. *Software Engineering.* Pearson, 5th edition, 1995.

[87] J. D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper.* AuthorHouse, 2004.

[88] E. Arisholm, L. C. Briand, and F. Audun. Dynamic coupling measurement for Object-Oriented software. *IEEE Transactions on Software Engineering*, 30:491–506, 2004.

[89] L. C. Briand, J. Wuest, and H. Lounis. Using coupling measurement for impact analysis in Object-Oriented systems. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA, 1999. IEEE Computer Society.

[90] L. C. Briand, J. W. Daly, and J. K. Wst. A unified framework for coupling measurement in Object-Oriented systems. *IEEE Transaction on Software Engineering*, 25(01):91–121, 1999.

[91] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: Cocomo 2.0, 1995.

[92] Object Point. http://sunset.usc.edu/csse/research/cocomoii/cocomo main.html, 2008.

[93] S. K. John, A. C. John, and A. M. John. Class mutation: Mutation testing for Object-Oriented programs. In *Proc. Net.ObjectDays*, pages 9–12, 2000.

[94] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transaction on Software Engineering*, SE-7(5):510–518, Sep. 1981.

[95] N. Nagappan, T. Ball, and B. Murphy. Using historical in-process and product metrics for early estimation of software failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 62–74, Washington, DC, USA, 2006. IEEE Computer Society.

[96] A. Hassan, K. Goseva-Popstojanova, and H. Ammar. UML based severity analysis methodology. In *Proc. of Annual Reliability and Maintainability Symposium (RAMS 2005)*, pages 158–164, Alexandria, VA, Jan. 2005.

[97] B. Boehm. Value-based software engineering: reinventing. *SIGSOFT Software Engineering Notes*, 28(2):3–10, 2003.

[98] M. H. Tang, M. H. Kao, and H. Chen M. An empirical study on Object-Oriented metrics. In *Sixth International Software Metrics Symposium*, pages 242–249, 1999.

[99] S. R. Luke. Failure Mode, Effects and Criticality Analysis (fmeca) for software. In *5th Fleet Maintenance Symposium*, pages 731–735, Virginia Beach, VA (USA), Oct. 1995.

[100] L. H. Rosenberg, R. Stapko, and A. Gallo. Risk-based object oriented testing. In *Proceedings of the 24 th annual Software Engineering Workshop, NASA, Software Engineering Laboratory*, 1999.

[101] Procedures for performing a failure mode, effects, and criticality analysis. Defense Department, US MIL STD 1629A/Notice 2, Nov. 1984.

[102] N. Ozarin. Failure mode and effects analysis during design of computer software. In *Annual Reliability and Maintainability Symposium*, pages 201–206, LA, USA, Jan. 2004.

[103] P. Vyas and K. Mlittal R. Operation level safety analysis for Object-Oriented software design using sfmea. In *WEE International Advance Computing Conference*, Patiala, India, Mar. 2009.

[104] S. Bhattacharya and A. Kanjilal. Code based analysis for Object-Oriented systems. *Journal of Computer Science and Technology*, 21(6):965–972, Nov. 2006.

[105] S. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice.* Willey, 2008.

[106] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *IEEE Internal Conference on Software Engineering*, pages 402–411, 2005.

[107] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Software Engineering*, 6(2):111–142, 2001. DOI: 10.1023/A:1011429104252.

[108] L. C. Briand, Y. Labiche, and Y. Wang. A comprehensive and systematic methodology for Client-Server class integration testing. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2003. IEEE Computer Society.

[109] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado. JaBUTi Java Bytecode Understanding and Testing: users guide. Carlos,S. and SP, Brazil, Mar. 2003.

[110] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using process history to predict software quality. *Computer*, 31:66–72, Apr. 1998.

[111] G. B.Mund and R. Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79(6):791–806, 2006.

[112] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 11–20, New York, NY, USA, 1998. ACM.

[113] A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2001.

[114] V. Garousi, L. C. Briand, and Y. Labiche. Control flow analysis of UML 2.0 sequence diagrams. In *LNCS 3748*, pages 160–174. European Conf. on Model Driven Architecture-Foundations and Applications, Springer Berlin / Heidelberg, 2005.

[115] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Iqbal, and A. Nadeem. A state-based approach to integration testing based on UML models. *Information and Software Technology*, 49(11-12):1087–1106, 2007.

[116] V. Garousi, L. C. Briand, and Y. Labiche. Analysis and visualization of behavioral dependencies among distributed objects based on UML models. Technical Report TR SCE-06-03, Carleton University, Ottawa, Canada, Mar. 2006.

[117] S. Supavita and T. Suwannasart. Testing polymorphic interactions in UML sequence diagrams. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, pages 449–454, Washington, DC, USA, 2005. IEEE Computer Society.

[118] MAGICDRAW UML. Available from:. http://www.magicdraw.com.

[119] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. Software Eng.*, 24(08):586–601, Aug. 1998.

[120] K. Cai, Y. Li, and K. Liu. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology*, 46:989–1000, 2004.

[121] R. Roshandel. *Calculating architectural reliability via modeling and analysis*. PhD thesis, University of Southern California, Los Angeles, CA, USA, 2006.

[122] P. Johannessen, C. Grante, A. Alminger, U. Eklund, and J. Torin. Hazard analysis in object oriented design of dependable systems. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 507–512, Washington, DC, USA, 2001.

[123] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[124] M. Burgess. *Fault Tree Creation and Analysis Tool: User Manual*, 2003. http://www.iu.hio.no/FaultCat.

[125] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel. Reliability prediction in model-driven development. *LNCS*, 3713:339–354, Oct. 2005. Springer-Verlag Berlin Heidelberg 2005.

[126] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transaction on Software Engineering and Methodology*, 13(1):37–85, 2004.

Dissemination

Published & Accepted in Journals

1. **Mitrabinda Ray** and Durga Prasad Mohapatra, Towards a multi-cycle based test effort prioritization method, International Journal of Software Engineering (ACTA Press), Accepted.

2. **Mitrabinda Ray** and Durga Prasad Mohapatra, Risk analysis: A guiding force in the improvement of testing, IET Software, Accepted.

3. **Mitrabinda Ray** and Durga Prasad Mohapatra, Code-based prioritization: A pre-testing effort to minimize post-release failures, Innovations in Systems and Software Engineering: A NASA Journal (Springer), Vol. 8, No. 3, 2012, doi: 10.1007/s11334-012-0186-3.

4. **Mitrabinda Ray**, Kanhaeya Kumwat, Durga Prasad Mohapatra, Source code prioritization using forward slicing for exposing critical elements in a program, Journal of Computer Science and Technology (Springer), Vol. 26, No. 2, 2011, pp: 314-327.

5. **Mitrabinda Ray** and Durga Prasad Mohapatra, Risk analysis at the architectural level is intended to act as a guiding force in the improvement of testing, SETLabs Briefings, Infosys, Vol. 9, No. 4, 2011, pp: 41-60.

6. **Mitrabinda Ray** and Durga Prasad Mohapatra, Designing Influence Metric at the Architectural Level for Improving the Reliability of a System, International Journal of Computer Application, Vol. 29, 2011, pp: 16-23.

7. **Mitrabinda Ray** and Durga Prasad Mohapatra, Prioritizing program elements: A pretesting effort to improve software quality, ISRN Software Engineering (Hindwai), Volume 2012 (2012), Article ID 598150, 20 pages, doi:10.5402/2012/598150.

Published in Conferences

8. **Mitrabinda Ray** and Durga Prasad Mohapatra, A scheme to prioritize classes at the early stage for improving observable reliability, Proceedings of the 3rd India software engineering conference, Mysore, India. (ACM), 2010.

9. **Mitrabinda Ray** and Durga Prasad Mohapatra, Reliability improvement based on prioritization of source code, Distributed Computing and Internet Technology (ICDCIT), Bhubaneswar, India. (Springer-Verlag), 2010.

Communicated in Journals

10. **Mitrabinda Ray** and Durga Prasad Mohapatra, Ranking Use Cases of a System for Testing Based on Complexity and Business Value, Communicated to Journal Systems Engineering (Wiley interscience).