Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Models

Baikuntha Narayan Biswal

(Roll No: 607CS003)

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Rourkela-769 008, Orissa, India

July, 2010.

Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Models

Thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Technology(Research)

in

Computer Science and Engineering

by

Baikuntha Narayan Biswal

(Roll No: 607CS003)

under the guidance of

Dr. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Rourkela-769 008, Orissa, India

July, 2010.

Albert Einstein:

If we knew what it was we were doing,

***It would not be called* research.**

Would it?

To my family and friends

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

Certificate

This is to certify that the work in the thesis entitled ***"Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Models"*** submitted by ***Baikuntha Narayan Biswal*** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology (Research)* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Date:
Place: NIT, Rourkela

Durga Prasad Mohapatra
Associate Professor
Department of CSE
National Institute of Technology
Rourkela-769008

Acknowledgment

It is my sincere thanks to all enlisted people as well as not enlisted people, whose help and continuous inspiration leads to get this thesis done in a pleasant way; however it will be tough to thank them enough. I will nevertheless try. . .

I would like to owe my gratitude and deep sense of respect towards my adviser and guide Prof. Durga Prasad Mohapatra, whose timing supervision and guidance has given me the moral boost in doing this work fine and furnished. I would like to express my thankful to him for giving me a chance to work in the field of software testing, which is really interesting and wonderful. I have enjoyed a lot in learning, in expressing my thought in free and embraced manner, for which I feel really fortunate to have a guide like him.

I would also like to express my loving respect and sincere thanks to Prof. Santanu Kumar Rath, for his valuable suggestions and encouragement during this work period at Software Engineering laboratory. I am greatly indebted to his invaluable advice and support in almost every aspect of my academic life. I am very much indebted to Prof. B. Majhi, for his continuous encouragement and support. My sincere thanks goes to Prof. A. K. Turuk, HOD, CSE for his technical suggestions to improve the quality of thesis work. My sincere thanks also goes to Prof. S. K. Jena, Prof. R. Baliarsingh, Prof. B. D. Sahoo and Prof. P. M. Khilar for motivating me to work harder. My overwhelming thanks goes to Prof. P. K. Sa and Prof. K. Sathyababu, who are one time teachers and senior cum friend, for their critical acclaimed comment and encouragement. I would also like to vow my thankful to Prof. S. Chinara, Prof. S. Mohanty and Prof. M. N. Sahu for their support and encouragement.

I wish to thank all the secretarial staff of the CSE Department and NIT Rourkela for their sympathetic cooperation.

I thank to all of my friends for being there whenever I needed them. Thank you very much Soubhagya, Swati Vipsita, Hunny, Swasti, Suraj and Sasmita and others of CSE Dept. and C. V. Raman Hall of residence. I have enjoyed every moment I spent with

you.

When I look back at my accomplishments in life, I can see a clear trace of my family's concerns and devotion everywhere. My dearest mother, whose patience to listen me has been a driven force to all of my achievement; my loving father, for always believing in me and inspiring me to dream big even at the toughest moments of my life; and my brother's and sister; who were always my silent support during all the hardships of this endeavor and beyond.

Last but not last I wish to appreciate and thank Chinu, for her understanding and encouragement when it was most required.

Funding for this research was provided by CS-TOS Project Grant 2007-2010, U. G. C, Govt. of India.

Baikuntha Narayan Biswal

Abstract

Testing guarantees the quality of software to be developed in terms of presence of bugs or errors. Testing can be separated into two categories such as White Box and Black Box testing. White box testing is done through detail analysis of program structure where as black box methodology deals with specification and design document i.e. without program details. Thus black box testing methodology holds major advantages, as tester can generate the test cases before code is developed, using specification and design document.

Off the late, Object-Oriented program have changed the scenario of software development industry in terms of software development and its supporting technology. The object-oriented features like inheritance and encapsulation has made it easy and suitable confined to design. The inheritance feature encourages to re-use the developed components where as the encapsulation conceals the details from others. And other features of object-oriented program like polymorphism, data abstraction and modularity have increased its richness. However these features have increased the job of software tester. Special attraction are needed to look into these features while testing is carried out.

UML, which supports object-oriented technology is widely used to describe the analysis and design specifications of software development. UML models are an important source of information for test case design. UML activity diagrams describe the realization of the operation in design phase and also support description of parallel activities and synchronization aspects involved in different activities perfectly. However UML Collaboration and Sequence diagram describes the way in which different objects interacts with each other, sequence of message passing between different objects. And Class diagram identifies the different classes, its attributes and operations respectively.

We propose a method to generate test cases using UML activity diagram. We first construct the activity diagram for the given problem and then randomly generate initial test cases, for a program under testing. Then, by running the program with the generated test cases, we obtain the corresponding program execution traces. Next, we compare

these traces with the constructed activity diagram according to the specific coverage criteria. We use path coverage as test adequacy criteria.

Next, we propose a novel approach to generate test cases from test scenarios using UML activity, sequence and class diagram. First we generate test scenarios from the activity diagram and then for each scenario the corresponding sequence and class diagrams are generated. After that we analyze the sequence diagram to find the interaction categories and then use the class diagrams to find the settings categories. After analyzing each category using category partitioning method, its significant values and constraints are generated and respective test cases are derived.

Finally, we propose a technique to optimize the generated test cases automatically. We define an error minimization technique in our approach, which works as the basic principle for optimized test case generation. Transition coverage is used as test adequacy criteria in this approach.

Keywords: *UML; test case; program under testing; program execution traces; test scenario; category partition method; test case optimization.*

Dissemination of Work

- **Baikuntha Narayan Biswal**, Soubhagya Sankar Barpanda and Durga Prasad Mohapatra, *"A Novel Approach for Optimized Test Case Generation Using Activity and Collaboration Diagram"*, International Journal of Computer Application (IJCA), vol. 1, no. 14, pp. 67 – 71, 2010.

- **Baikuntha Narayan Biswal**, Pragyan Nanda and Durga Prasad Mohapatra, *"A Novel Approach for Test Case Generation Using Activity Diagram"*, International Journal of Computer Science and Application, vol. 1, no. 1, pp. 61 – 65, 2008.

- **Baikuntha Narayan Biswal**, Pragyan Nanda and Durga Prasad Mohapatra, *"A Novel Approach for Test Case Generation Using Activity Diagram"*, International Conference on Advance Computing, Chikhli, 2008.

- **Baikuntha Narayan Biswal**, Pragyan Nanda and Durga Prasad Mohapatra, *"A Novel Approach for Scenario-Based Test Case Generation"*, International Conference on Information Technology (ICIT), Bhubaneswar, 2008.

List of Acronyms

Acronym	Description
OO	Object-Oriented
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
4*GL*	4th Generation Language
OOST	Object-Oriented Software Testing
OOD	Object-Oriented Design
OOAD	Object-Oriented Analysis and Design
UML	Unified Modeling Language
SDLC	Software Development Life Cycle
MBT	Model Based Testing
OMG	Object Management Group
OMT	Object Management Technology
SUT	System Under Testing
PUT	Program Under Testing
MDG	Message Dependence Graph
FSM	Finite State Machine
UI	User Interface
GA	Genetic Algorithm
TDE	Test Development Environment
CA	Class Attribute
XML	eXtensible Mark-up Language
XSL	eXtensible style sheet Language
DFS	Depth First Search
MDA	Model Driven Architecture

List of Figures

1.1 Testing Information Flow 2

2.1 Structure of GA 18

3.1 Input Domain Data for Testing using GA 28

4.1 An illustration of an Activity diagram 33
4.2 System model of our approach 35
4.3 Activity Diagram for Super Market Prize Winner System 40

5.1 Our Proposed Approach 45
5.2 Activity diagram for money withdrawal from ATM 49
5.3 Class diagram for ATM money withdrawal 50
5.4 Sequence Diagram for money withdrawal from ATM 50

6.1 Proposed framework for optimized test case generation 56
6.2 Activity diagram for cash Withdrawal in ATM 58
6.3 Collaboration Diagram for cash Withdrawal in ATM 59
6.4 Activity Diagram for Balance Enquiry 63
6.5 Activity Diagram for PIN Verification 64
6.6 Collaboration diagram for Balance Enquiry with print receipt 64
6.7 Collaboration diagram for Balance Enquiry without print receipt 65
6.8 Collaboration diagram for PIN Verification 65
6.9 Transition Coverage Vs Error Detected 66
6.10 Errors Detected Vs Errors Recovered 66

List of Tables

4.1 Generated Test Cases for Super Market Prize Winner System 40

4.2 Branch Coverage and Path Coverage using our approach 41

5.1 State of the system for test scenario generation 49

5.2 Generated Test Cases with test scenario 52

5.3 Generated Test Cases for ATM Withdrawal 52

6.1 Result produced by our approach 62

Contents

Certificate **iii**

Acknowledgment **iv**

Abstract **vi**

Dissemination of Work **viii**

List of Acronyms **ix**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**

1.1 Why testing is essential? 3

1.2 Object-Oriented Technology and Software Testing 4

1.3 Unified modeling language and model based testing 5

1.4 UML Diagrams 6

1.5 Automatic Test Case Generation 7

1.6 Test case optimization 8

1.7 Motivation 8

1.8 Problem Statement and Objectives 9

1.9 Thesis Outline 10

2 Basic Definitions and Concepts **12**

2.1 Preliminary Definitions and Concepts 12

2.2 Brief notes on Model Based Testing 13

2.3 Overview of UML Diagrams 13

2.4 Test Case Optimization 15

2.4.1 Basics of Genetic Algorithm (GA) 16

2.5 Conclusions 17

3 Literature Survey **19**

3.1 An overview of Object-Oriented Software Testing 20

3.2 UML as a test model 22

3.2.1 Use Case Diagram Based Approach 23

3.2.2 Class Diagram Based approach 24

3.2.3 Communication Diagram Based Approach 24

3.2.4 Sequence Diagram Based Approach 25

3.2.5 Activity Diagram Based Approach 25

3.2.6 Combined Approach 26

3.3 Test Case Optimization 27

3.4 Summary 29

4 Test Case generation using Activity Diagram **31**

4.1 Basic Concepts and Definitions 32

4.1.1 Activity Diagram 32

4.1.2 Test Adequacy Criteria for Activity Diagram 33

4.2 Proposed Frame work 34

4.3 Proposed Test Case Generation Methodology 36

4.3.1 Paths in the Activity diagram 36

4.3.2 Proposed Approach 37

4.3.3 Test case generation strategy 38

4.3.4 Our Proposed Algorithm 39

4.4 Working of our algorithm 39

4.5 Conclusion 41

5 Test Case Generation using Activity, Sequence and Class Diagram **42**

5.1 Basic Concepts and Definitions 43

5.2 Proposed Test Case Generation Methodology 44

5.2.1 TC-ASEC: The Proposed Approach 44

5.2.2 Test Scenario generation: TSAD 45

5.2.3 Test Case Generation 48

5.3 Working of our algorithm 51

5.4 Conclusion 51
6 Optimized Test Case Generation **53**
6.1 Basic Concepts and Definitions 54
6.1.1 Collaboration Diagram 54
6.1.2 Test Adequacy Criteria 54
6.1.3 Prioritized Scenario 55
6.1.4 Genetic Algorithm 55
6.2 Proposed Frame work 56
6.3 Proposed Approach 57
6.3.1 Our Objective 57
6.3.2 Methodology 57
6.4 Working of our algorithm 62
6.5 Conclusion 63
7 Conclusion **67**
7.1 Contributions of our work 67
7.2 Scope and Future Work 69
Bibliography **70**

Chapter 1
Introduction

Software testing is the process of exercising a program with well designed input data with the intent of observing failures. In other words, *"Testing is the process of executing a program with the intent of finding errors"*. Testing identifies *faults*, whose removal increases the software quality by increasing the software's potential *reliability*. Testing also measures the software quality in terms of its capability for achieving *correctness, reliability, usability, maintainability, reusability and testability*. The various Objectives of testing are as follows:

- Testing is a process of executing a program with intent of finding an error.
- A good test is one that has a high probability of finding an *as-yet-undiscovered error*.
- A successful test is one that uncovers an *as-yet-undiscovered error*.
- Testing should also aim at suggesting changes or modifications if required, thus adding value to the entire process.
- The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.
- Performance requirements are required as it specified in specification document.
- Software reliability and software quality based on the data collected during testing.

The various advantages of testing are as follows:

- Increasing accountability and Control
- Cost reduction
- Time reduction
- Defect reduction
- Increasing productivity of the Software developers

We will get an abstract view of the objective and flow of testing from Fig 1.1

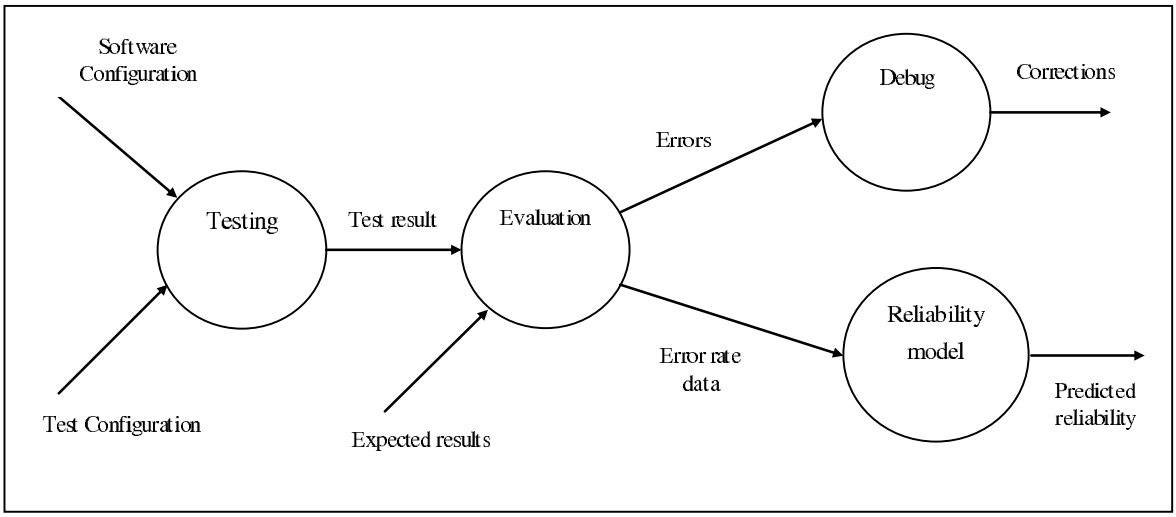

Figure 1.1: Testing Information Flow

Testing information flow is said to be as a testing technique which specifies the strategy to select input test cases and analyze test results [1]. Different testing techniques reveal different quality aspects of a software system, and there are two major categories of testing techniques such as functional testing and structural testing.

Functional Testing: The software program or *system under test (SUT)* is considered as a ***"black box"***. The selection of test cases for functional testing is based on the requirements or design specifications of the software entity under test. Examples of expected results sometimes are called ***test oracles***, which include requirement/design specifications, hand calculated values, and simulated results. *External behavior* of the software entity is the main attraction of functional testing.

Structural Testing: The software entity is considered as a ***"white box"***. The selection

of test cases is based on the implementation of the software entity. The main focus of such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of coverage criteria like path coverage, branch coverage, and data-flow coverage. *Internal structure* of the software entity is the main focus of structural testing.

1.1 Why testing is essential?

Now-a-days, computer applications have diffused into every sphere of life, for manipulation of several sophisticated applications. Many of these applications are of very large, complex and safety critical. Thus, highly *reliable* software is essential. In other words, the good quality software with high reliability is most essential. Apart from existence of many techniques for increased reliability, software testing is an important and common methodology followed. So, testing remains the most important part of quality assurance in the practice of software development. Although so many quality assurance techniques like formal specifications, design reviews, model checking, and inspection, exists till today, further furnishing method of testing is required for effective testing. The large software size is seen as major challenge while developing a quality software. So quality assurance is an important and major issue for large scale software development. According to Miller [2] the goal and need of software testing is "affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances".

Again the evolution of *high level programming* languages such as *object-oriented programming (OOP)* and development of *fourth generation language (4GL)* have added further problems to the scenario. The concepts such as *encapsulation, inheritance, polymorphism and dynamic binding* are the greatest strength of *object-oriented technology (OOT)* but at the same time they increase the complexity of the software and pose special difficulties for testing of the software. A large software with high complexity is the major challenge for software tester.

The computer society defines testing as “A verification method that applies a con-

trolled set of conditions and stimuli for the purpose of finding errors. This is the most desirable method of verifying the functional and performance requirements. Test results are the documented proofs, which shows that requirements are met and can be repeated. The resulting data can be reviewed by all concerned for confirmation of capabilities"[3].

1.2 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)* is an important software quality assurance activity to ensure that the benefits of object-oriented (O-O) programming will be realized. OOST has to deal with new problems introduced by the O-O features such as *encapsulation, inheritance, polymorphism, and dynamic binding*. Below, we discuss different levels of testing associated with object-oriented programs.

Intra-method testing: Tests designed for individual methods. This is equivalent to unit testing of conventional programs.

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.

Intra-class testing: Tests are constructed for a single entire class, usually as sequences of calls to methods within the class.

Inter-class testing: It is meant to test more than one class 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.

1.3 Unified modeling language and model based testing

In the last few years, *object-oriented analysis and design (OOAD)* has come into existence, it has found widespread acceptance in the industry as well as in academics. The main reason for the popularity of OOAD is that it holds the following promises:

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability

Development of *unified modeling language (UML)* has helped a lot to visualize/realize the software development process. At the earliest stage of *software development life cycle (SDLC)*, no one including user and developer can see the software; only at the final stage of the product development it is possible. Any errors/problems found out at the final stage, it incurs a lot of cost and time to rectify, which is very much crucial in IT industry. UML is the modeling language, which supports object-oriented features at the core. UML accomplish the visualization of software at early stage of SDLC, 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. Instrumentation of models into testing process is the prime subject of concern of our thesis. Testing methodologies 0which uses model is called *model based testing (MBT)*.

Model-based software testing generally refers to test case design based on models of the software specifications [4, 5]. Models are the intermediate artifacts between requirement specification and final code. Models preserve the essential information from the requirement, and are the basis for implementation. Therefore, models concisely describe the structural and behavioral aspects, are necessary for implementation of the software. Model based testing can be summarized in one sentence; "it is essentially a

technique for automatic generation of test cases from specified software model". The key advantage of this technique is that the test generation can systematically derive all combination of tests associated with the requirements represented in the model to automate both the test design and test execution process. We are discussing the former i.e. test design as part of the thesis work.

1.4 UML Diagrams

A wide range of modeling languages such as UML [6], SDL [7], Z [8, 9], state machine diagrams, etc have established with their own notations, terminologies and concepts. We can roughly categorize them into formal, semi-formal and informal models. There are several research reports on automatic generation of test cases from formal models [10, 11, 12]. However, at present formal models do not scale to large systems and are very rarely constructed in industry. Software industries typically use semi-formal modeling languages to model software systems. Informal models lack details and are not suitable for development of complex systems. Possibly the most widely used modeling language at present is Unified Modeling Language (UML). UML is a semi-formal visual language that has been developed to support the design and development of complex object-oriented systems [6, 13, 14]. It was adopted as a de facto standard for modeling software systems by the *object management group* (OMG) in November 1997. Later, in 2005, ISO also adopted UML version 1.4.2 as a standard. Since its adoption by OMG, the UML has been widely accepted by the software engineering community for documenting design models. Of late, researchers are focusing their attention on UML models as a source of information for test case generation [15, 16, 17, 18, 19, 20].

Along with the advantages there are also challenges for generating test cases from UML specification. For example, the models from the development process are abstract and typically lack several details present in the code and therefore are inadequate for comprehensive testing. To redress this situation, UML 2.0 adds several new capabilities to UML1.x. It has improved its precision and expressiveness to model large and complex architectures and this alleviates some of the major problems in test case gen-

eration. UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe functional, structural or behavioral aspects of systems. In our thesis, we have discussed merely on the behavioral aspects of UML diagrams for test case generation.

1.5 Automatic Test Case Generation

According to the definition, an important part of test case is to define the expected output or result. So a typical test case should have two components such as:

- An input data to the program.
- Description of correct output from the set of input data.

Generation of test cases to satisfy arbitrary test requirements is a nontrivial problem. Many researchers have focused on automation of this task and their reported results show with varying degrees of success. They have used different methodologies and different design artifacts of the system under test for automatic generation of test cases [16, 21, 22]. An automatic test case generator would take design artifacts as its input, process it and then generate test specifications based on certain pre specified testing criteria called as test coverage criteria. Subsequently, the exact test data for each test specification is determined to form the test cases. Software testing is successively executed by generating test case for set of input data and builds the confidence of the developer.

Test cases can be derived from requirements and specifications, design artifacts, or the source code. Test cases are commonly designed based on program source code. This makes test case generation difficult especially for testing at cluster levels. Test case generation from design documents has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. Another advantage of design-based testing is to test the compliance of the implementation with the design documentation. Manual generation of test case is time consuming and laborious. Hence either automatic or semi-automatic generation of test case from design document is often desired.

1.6 Test case optimization

A good test case should have the quality to cover more features of test objective. In other words effectiveness of testing process relies on the quality of test cases not in the quantity of test cases, which in turn lingers the testing time. We can get an appropriate amount (or optimal) number of test cases of better quality, by eliminating redundant (or unnecessary) test cases. And so the problem of time consuming in testing phase can be reduced. But getting all those test cases in a rush time (time to deliver the software to the client) is a cumbersome task. Therefore, automatic generation of test cases reduces the effort of a tester and developer and so cost and time.

1.7 Motivation

Several approaches to design test cases and application of Genetic Algorithm on software testing have been proposed by researchers. These approaches include generation of test cases from requirements specification i.e. black box testing or from code i.e. white box testing or from model-based specification. Test case generation solely based on requirements specification completely ignores system implementation aspects. Further, it cannot reveal whether the software performed some tasks which are not specified in the requirement specifications. On the other hand, test case design from program code is cumbersome and difficult to automate. Besides, it also cannot reveal missing functionalities. Further, the traditional system testing techniques - black box as well as white box testing, achieve poor state coverage in many systems. The reason being that the system state information is very difficult to identify either from the requirement specifications or from the code [23]. Model-based testing which uses UML design specifications for test case generation overcomes these shortcomings and has emerged as a promising testing method. Further, models being simplified representations of systems are more amenable for use in automated test case generation. Automation of test case design process can result in significant reductions in time and effort, and at the same time it can help in achieving an increased reliability of the software through increased test coverage. An automated model-driven test case framework is therefore desirable.

At the same time we have studied on optimization of generated test cases. Optimized test cases is not only helpful in quick the testing process but also cost saving. It is also essential to differentiate among the various test cases. It defines the clear cut objective in front of the tester. No need to go for different test cases, which may serve different objectives, in turn save the time and money. The optimization of test case process further expands the scope in the field of priority based testing.

1.8 Problem Statement and Objectives

Software Testing is a time consuming and costly process in software development life cycle. Automation of this phase may lead to overcome the above problems and also reduces the human effort in other ways it also helps in detecting the human intended errors and logical errors as well. Automation of testing will not be that much productive in terms of time consuming and cost, if we have to wait till the end of the SDLC stage i.e. if we follow the white box testing methodology of testing. If any errors will be detected in this stage, we have to go for that part of the code and design document as well. We have to follow up strict verification of both code and design document from beginning to short out the error. So only one solution to this problem is to, start the testing process from early stage of SDLC i.e. from requirement specification stage through design phase up to the last phase. So we have studied on Model Based testing approach for both test case generation and test case optimization to achieve some of the goal, described below:

- To propose some generalized techniques to generate test cases for object-oriented softwares using UML diagrams such as:
  - Activity Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - Class Diagram

- To propose a generalized technique for optimized test case generation using UML diagrams.
- To implement the proposed methods and evaluate their effectiveness.

1.9 Thesis Outline

The rest of the thesis is organized into the following chapters:

Chapter 2, describes basic concepts, different terminologies and its definition in related with automation of software testing, model based testing and test case optimization.

Chapter 3, explores the review of existing research works on object-oriented software testing using UML diagrams. We have especially discussed various model based testing approaches and test case optimization, in this chapter.

In **Chapter 4**, we discusses the proposed technique for automatic test case generation using activity diagram. We first discuss a few basic concepts and definitions used in describing our methodology. Next, we have also described a framework to carry out our proposed approach. Subsequently, we describe our proposed test case generation methodology using activity diagrams. Finally we illustrate the test case generation strategy with an example of *super market prize winner schemem* in working of our algorithm section followed by conclusion.

In **Chapter 5**, we describe our proposed approach of scenario based test case generation using UML activity sequence and class diagram. First, we discuss a few basic definitions and terminologies in related to our work. Then, we describe our methodology of test case generation using UML activity, sequence and class diagram. Subsequently, we explain the working of our approach by considering ATM withdrawal as a case study. Finally, we present the conclusion.

Chapter 6, presents our approach for optimized test case generation from UML activity and collaboration diagram using *genetic algorithm (C-GA)*. Initially, we provide few related basic concepts and definitions. Then we present our proposed framework of automatic test case generation, followed by proposed approach of optimized test case generation. Next, we present the working of our algorithm. Finally, we concludes with

conclusion.

Chapter 7, concludes the thesis. We also highlight the important contributions of our work. Finally, we discuss the possible future extensions to our work.

Chapter 2

Basic Definitions and Concepts

It is essential to discuss some basic concepts and definitions to understand the thesis. In this chapter, we discuss the basic definitions and terminologies, on which our research is based. The rest of this chapter is organized as: In Section 2.1, we describes the preliminary definitions and Concepts. Section 2.2, presents a brief introduction on model based testing. Overview of UML diagrams is discussed in Section 2.3 and followed by the discussion on test case optimization in Section 2.4. Section 2.5 presents the conclusion of the chapter.

2.1 Preliminary Definitions and Concepts

Test Case:

A test case is combination of trios such as, set of inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular scenario, a particular scenario sequence or to verify compliance with a specific arguments [24].

Test Adequacy Criterion:

It is well known the fact that testing makes the software error free and increases the reliability. But it is not known when to stop the testing process or what constitutes the adequacy of a test. Goodenough and Gerhart [25] made an early stuck to this. Test adequacy criteria is nothing but an essential and important predicate, which shows the adequacy. In other words it is a stopping condition to stop the testing process, when all the defined criteria have been met [26]. Branch coverage, path coverage, transition coverage, activity coverage are few of such test adequacy criteria, whereas we have

used only path coverage and transition coverage for our work. A test adequacy criterion helps in defining test objectives or goals that are to be achieved while performing a specific software testing. For example, branch coverage requires that every branch in a program under test is to be exercised by at least one test case.

2.2 Brief notes on Model Based Testing

Conventional testing procedure for procedural programs follows the code based approach [27, 28]. However testing of object-oriented program with distinct design pattern and software structure, solely only with source code may not so much effective. It is advisable to follow the mixed approach of source code and requirement specification document for testing the OOPs. In this context model based testing, which is also called as gray box testing is ideal [5]. Gray box testing method is the combined approach of white box and black box. Models are the intermediate artifacts between requirement specification and source code. Model based software testing generates test cases based on models of the specifications [4, 5]. Models preserve the essential information from requirement specification and are base for the final implementation. Though, there are lots of modeling language such as UML, SDL, Z-Specification, we will discuss about use of UML as a test model, which is a semi formal modeling language.

2.3 Overview of UML Diagrams

Unified Modeling Language (UML) is a semiformal visual modeling language, which is a collective approach of trio James Rumbaugh (Object Management Technology), Grady Booch (Booch's Methodology) and Ivar Jacobson (Object-Oriented Software Engineering). It was adopted as a de facto standard for modeling software systems by OMG in 1997 [29]. Of late, popularity of UML models in academic and industry levels is attracting the focus of researchers for test case generation in the context of model based testing.

UML diagrams generally describe the different views of the system, such as User's View, Structural View, Behavioral View, Implementation View and Environmental View

[6, 30]. The UML user's view is expressed by Use Case diagram and it is used to capture the functionalities of the system. The user's view 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. Use case diagrams are mainly used for requirement-based testing and high level test design [31]. Jacobson, et al. [31] suggest four general kinds of tests that can be derived from use cases and these are: 1)tests of expected flow of events, 2)tests of unusual flow of events, 3)test of any requirement explicitly attached to a use case and 4)tests of features described in user documentation.

The UML structural view defines the kinds of objects (or classes), those are important to understand the working of a system and its implementation. Structural view also captures the relationships among the classes (objects) and is also called as the static model, since the structure of a system does not change with time. The structural view includes class diagram, object diagram and composite structure diagrams [6]. The UML class diagrams along with state machine diagrams have traditionally been used for testing object-oriented programs at the unit level [24]. The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system. Behavioral view includes different diagrams such as *Activity Diagram, State Machine Diagram (or State Chart Diagram), Sequence Diagram and Communication Diagram (or Collaboration Diagram)* and are discussed below [29].

- **Activity diagram:** Activity diagrams describe the workflow behavior of the system. These are similar to state diagrams because activities are the state of doing something. The easiest way to visualize an activity diagram is to think of a flowchart of a code. The flowchart is used to depict the business logic flow and the events that cause decisions and actions in the code that take place. In general activity diagram describe the internal behavior of an operation. Activity diagram can show activities that are conditional or parallel. Activity diagram shows how objects behave or how they collaborate. Diagrams are read from top to bottom and have branches and forks to describe condition and parallel activities.

- **State machine diagram:** State machine 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, but state machine diagram may also be used to describe the behavior of other entities such as use cases, actors, subsystems, operations, or methods.

- **Sequence diagram:** An UML sequence diagram is an interaction diagram that captures time dependent (temporal) sequences of interactions down between objects. They show the chronological sequence of the messages, their names and responses and their possible arguments.

- **Communication diagram (or Collaboration diagram in UML 1.X):** A collaboration diagram is an interaction diagram that shows how objects interact with each other to achieve a behavioral goal. A collaboration diagram shows the structural relationships among the objects as well as the behavioral aspects a of the objects that send and receive messages. Compared to a sequence diagram, a collaboration diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads must be determined using sequence numbers.

Both these sequence and communication diagram also comes under interaction diagram. The implementation view captures the important components of the system and their dependencies and includes component diagram. The last and 5th view of UML is environmental view, which models how the different components are implemented on different pieces of hardware. This view is expressed by deployment diagram. Our work is confined to first three views of UML i.e. user's view, structural view and behavioral view, which includes the use case, class, sequence, collaboration and activity diagrams.

2.4 Test Case Optimization

As we know testing is very much expensive and time consuming process, which incurs about 40-60% of the total cost of the software. So it is necessary to reduce the test case

numbers or test suite size without dickering the quality. Size of the test case is reduced that does not mean that the quality factor will be overlooked, rather technically it deals with the term effectiveness. The test cases will be considered those have a good impact in finding the errors along with fulfilling the specified coverage criteria. That is it is an optimized concept, where the best fit test cases are selected for test case execution on SUT and rests are ignored. Reduction of test cases can be done in two ways i.e. either at the time of generating test cases or after generating an initial set of test cases [32]. Reducing the test cases at the time of generating, avoids generation of redundant test case while the other one can be seen as an optimization problem, as reducing the test suite implies optimization of the test suite based on certain defined optimization criteria. Our work is categorically refers to the second case i.e. test case optimization after generation of the initial test case by random method. We have worked on genetic algorithm based optimization approach to reduce the test case and we will go to an insight of it in the rest part of thesis.

2.4.1 Basics of Genetic Algorithm (GA)

Typically, genetic algorithm is a searching technique used in computing exact or approximate solutions to optimization and search problems from various domains, including science, commerce and engineering. The primary reason for their success is their broad applicability and ease of use, which is also offers a robust non-linear search option involving large variables [33]. The name indicates its working principle, which is based on the concept of evolution in biological system.

In GA, the candidate solutions are encoded using chromosomes. The algorithm then looks for a better solution among a number of chromosomes (candidate solutions), also called population of solutions, based on the principle of the survival of the fittest (also called evolution). The evolution is based on two primary operators: mutation and crossover. The genetic operator crossover involves segment interchange between two mating chromosome whereas the mutation operator is used to alter the chromosome slightly. Mutation is essentially useful to maintain the diversity from one generation to the next generation of population of chromosome. To be more precise a typical GA

requires three basic components to be defined - (a) a genetic representation of the solution, (b) a fitness function to evaluate the candidate solutions and (c) creation of new population. The pseudo code and structure of the genetic algorithm is presented below for better understanding the concept. Here t is the generation number and P the population.

```
begin
    t=1;
    initialize P(t);
    While not finished
        evaluate P(t);
        select P(t+1) from P(t);
        Recombine P(t+1) using
            crossover;
            mutation;
        survive;
        t=t+1;
end;
```

Pseudo code of GA

2.5 Conclusions

In this chapter we have presented the basic concepts and theory for understanding the thesis work, described in subsequent chapters. First, we have described some basic theory related to automatic test case generation approach. Then we have discussed on model based testing, which is the prime attention of thesis. We have primarily focussed on the pros and cons of the model based testing to the test case generation automatically. To the very next section, we have discussed on UML diagrams, which is the building block of our thesis work. Testing is an optimization problem, is another feature of our thesis work, which sees the many opportunities in the current scenario. Our test case optimization approach using GA, implicitly deals with error minimization.

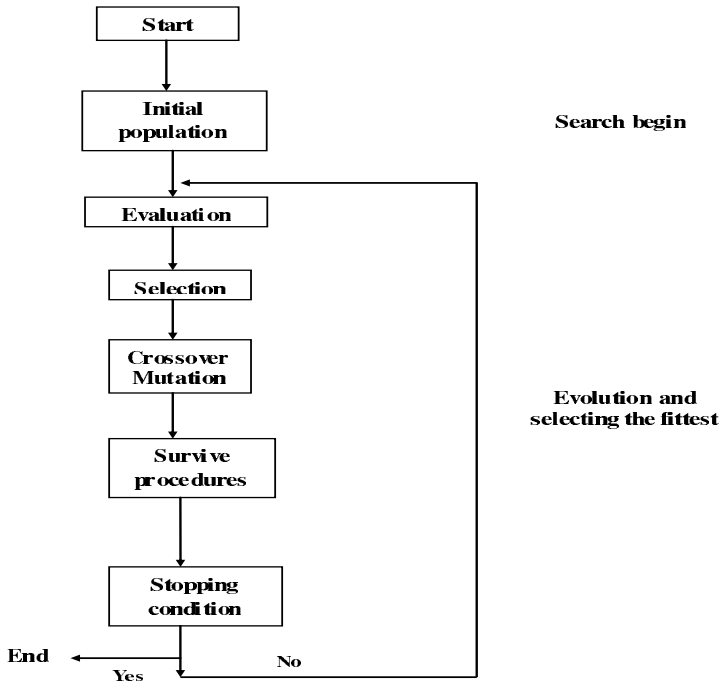

Figure 2.1: Structure of GA

Chapter 3

Literature Survey

Software testing is the process of exercising a program with well designed input data with the intent of observing failures [29, 34, 35]. In other words, Software testing addresses the problem of effectively finding the difference between expected behavior specified by the system models and the observed behavior of the implemented system [24].

At present, software testing on the average makes up as much as 40% to 60% of the total development cost and would increase even further with rapidly increase size and complexity of software [29, 35, 36]. As systems are getting larger and more complex, the time and effort required for testing are expected to increase even further. Therefore, automatic software testing has become an urgent practical necessity to reduce testing cost and time. In this purview, we will start our discussion on theory and related survey on automatic test case generation and optimization of object-oriented software. Our research is focused on the use of UML models for the above said purpose i.e. automatic test case generation and test case optimization. We will find so many researchers who have already worked in UML for test case generation [5, 37, 38]. Basic concepts on Object-Oriented Software Testing (OOST) strategy and use of UML diagrams as test model were covered in this chapter. We also discuss some basic concepts of *genetic algorithm (GA)* and *optimization*. In this chapter, we will go through some research papers those have covered extensively the various aspects of object-oriented software testing using UML diagrams.

This chapter is organized in the following manner.

In Section 3.1, we discuss the overview of object-oriented software testing. Section 3.2

discusses UML diagrams and its application to software testing. Test case optimization and application of GA to testing is discussed in Section 3.3, followed by conclusion in Section 3.4.

3.1 An overview of Object-Oriented Software Testing

Development of object-oriented technology has added the extra advantages to the large scale software development in IT industry. Concepts like *re-usability, encapsulation, dynamic binding* and *polymorphism* are the key drivers of this technology. In the same time these features of OOT has also increased the complexities of testing process. The concept of *message passing* is also adds the further complexity to the problem of testing pool [39]. The message passing invokes the receiver objects upon receiving the message from sender objects and causes some operation based on the received message that leads to change the state of the object.

Kung and Hsia [39] has described the various dependencies in the Object-Oriented programming in addition to the traditional procedural programming dependencies like data dependency, functional dependency and etc. Following are some of the identified dependencies of Object-Oriented system.

- class to class dependencies
- class to method dependencies
- class to message dependencies
- class to variable dependencies
- method to variable dependencies
- method to message dependencies
- method to method dependencies

The concept of *encapsulation and information hiding* features have developed the problem of understanding. Complex relation that exists in OOP causes the problem of de-

pendency. A summarized view on problems of OO testing is discussed by Kung, et al. [40] and are as follows:

- the understanding problem
- the complex interdependency problem
- the object state behavior testing problem
- the tool support problem

The major difference between an OOP and a conventional program is in term of its structure and behavior. Structurally a conventional program consists of three levels of components i.e. functions (or procedures), modules, and subsystems. However, an object-oriented program consists of four levels of components i.e. function members (defined in a class), classes, groups of classes, and subsystems. The conventional data-flow graph and control flow graph can be used to represent the structure of a class function member. A class flow-based graph can be used to model the interactions between functions defined in a class [41]. A class relation diagram can be used to model various relationships between classes, including inheritance, aggregation, and association relations [40].

Another major difference between an object-oriented program and a traditional program is their behaviors. In a dynamic view, a traditional program is made a number of active processes and each of them has its control flow. They communicate with each other through data communication, where as an object-oriented program consists of a collection of active objects. Each of the object communicates with one and another to complete the specified functions. In a multiple-thread program, a concurrent message flow and message execution takes place at the same time. These differences reveal some new problems in integrating different components for an object-oriented program. Jorgensen and Erickson [42] proposed a method for integration testing with five distinct levels of object-oriented testing. The five distinct levels of object-oriented testing includes: a method, message sequence, event sequence, thread testing, and thread interaction testing.

Harrold, et al. [43] have described a class testing methodology that utilizes the hierarchical nature of classes related by their inheritance relationships to reduce test overhead. Since classes are the major components in OOPs, existing techniques are in embraced situation to deal with this, which causes the problem of unit testing and integration testing. Parrish, et al. [41] have discussed on the conventional flow graph-based testing strategies to classes. Based on this flow-graph model, they proposed a general-class graph model by extending the basic modeling concept to represent classes.

Turner and Robson [44] described a state-based testing method for testing the interactions between the features of an object and the object's state. They have used black-box testing methodology for their approach. That means this paper has discussed on object state (it is defined as the combination of the attribute values of the object) testing, which is an important aspect of object oriented software testing. It varies from the conventional control flow testing and data flow testing methods. Control flow testing focuses the program testing according to the control structures (i.e., sequencing, branching, and iteration), where as data flow based testing focuses on testing the correctness of individual data define-and-use. Object state testing focuses on testing the state dependent behaviors of objects. The features of an object are usually implemented as the object's operations or methods. When an object is executed, causes the state transitions from an "input state" to an "output state". They have used finite state machine (FSM) to model object state dependent behaviors [44]. From these models test cases are generated to test the implementation. This paper have used a random order approach to invoke the features.

3.2 UML as a test model

UML is widely used to describe the analysis and design specifications of software development. UML models are an important source of information for test case design. A detailed purview of use of UML diagrams in software testing is presented in the following section.

3.2.1 Use Case Diagram Based Approach

As discussed in above section, use case diagrams are used to visualize the functionalities and behavior of a system. They consist of a set of use cases and actors and their relationships [14]. Several researchers [15, 16, 19, 21, 45] have worked on use case diagram based system test case design. Most of the reported work on use case-based system testing focus on the coverage of all scenarios of each use case [15, 19, 45]. A few other works on use case-based system testing consider the dependencies between the use cases and model these dependency using graphical notations [16, 21].

Frohlich and Link [45] have proposed textual description method to describe use cases to generate system test cases. They first transform a use case along with its pre- and post conditions into a state machine representation. The states are nothing but abstractions, representing the interval between two successive messages sent to the system by a user. Then, valid test sequences are generated from the state model. Their test case generation approach essentially converts the test case design problem into a planning problem. Their approach for test case generation, usage of formal method and automatic approach, requires certain manual annotations to the UML models. The testing criterion they have considered is transition coverage of the state model. Another approach to generating use case based system level test cases has been proposed by Reibisch, et al. [19]. In this work, they first converts the use cases into state diagrams. A usage model is constructed from the state diagrams according to the operational use of the software [46]. The usage model describes both the system behavior and the usage of the software. Finally, the usage model is traversed to generate test cases. Hartmann, et al. [15] have also proposed an automatic test case generation methodology based on the interactions between the system and a user. They semi-automatically convert the textual description of use cases into annotated activity diagrams for model interaction. The annotated activity diagram with the test requirements are designed before test generation by the designer. The annotated activity diagram is then processed automatically to generate a set of textual test procedures called executable test scripts. This approach achieves coverage of transitions of the constructed activity diagrams.

3.2.2 Class Diagram Based approach

A class diagram shows the static structure of a system. It identifies all the entities, along with their attributes, in the system and specifies the relationships between the entities. The final coverage criterion based on class diagram is defined in [47], is the class attribute (CA) criterion. This criterion requires coverage of a set of attribute value combinations for each class in the class diagram. The category partition method is used to produce a set of possible values for each attribute in a class. Elements from each of these sets are combined to create a set of attribute values for each class.

3.2.3 Communication Diagram Based Approach

Various coverage criteria based on collaboration diagrams have been proposed. One such criterion is the all message sequence paths criterion and requires all message sequence paths in a collaboration diagram to be covered by test executions. It is not always feasible to achieve all message sequence paths coverage, as a collaboration diagram may contain an infinite or very large number of message sequence paths. Wu, et al. [48] have proposed a set of criteria that can be used for integration testing of component-based software. They defined, the all message sequence paths criterion along with the all transitions and all content dependence relationship coverage criteria. The second sets of criteria defined by Pilskalns, et al. [47] are related to collaboration diagrams. They defined the condition coverage as a criterion. In a collaboration diagram it is possible to specify that messages may only be executed under certain circumstances. This is achieved by associating a condition with the message. The condition coverage criterion requires that each condition in the collaboration diagram evaluate to both TRUE and FALSE. Therefore there must exists a test case that causes the condition to evaluate to TRUE and another to FALSE. Samuel, et al. [38] have proposed a method to generate test cases at cluster level based on UML communication diagrams. They have introduced a tree base representation of communication diagram, which is then goes on for a post-order traversal for selection of the conditional predicates. By transforming the conditional predicates with function minimization technique, test data is generated. The generated test cases achieve message paths coverage as well as boundary coverage.

3.2.4 Sequence Diagram Based Approach

Bertolino and Basanieri [49] proposed a method to design sequence diagram based test cases following the sequence of messages between components in a sequence diagram. They used category partition method [50] along with the sequence diagrams for each use case to generate the test data. In another work, Fraiklin, et al. [51] provides guideline for generating testable sequence diagrams and SeDiTeC, a test tool that supports automated generation of test stubs based on sequence diagrams. SeDiTeC can automatically generate test stubs for given classes and methods, whose behavior is specified in sequence diagrams and the corresponding test case data sets. Rountev, et al. [52] defined a control flow based coverage criteria for object interaction based on sequence of messages of reverse engineered sequence diagram. They have constructed a data structure called as inter procedural restricted control-flow graph (IRCFG), which is used to represent the set of message sequences in a compact manner in a sequence diagram. They have also discussed a run time analysis mechanism for coverage measurements for each criterion. Samuel and Mall [53] described a methodology to generate cluster level test case based on sequence diagrams. They have constructed a message dependence graphs (MDG) from UML sequence diagrams, which is then applied with edge marking dynamic slicing method to create slices. Based on the slices created with respect to each predicate on the sequence diagram, test data is generated. They have formulated a test adequacy criteria named as slice coverage criteria.

3.2.5 Activity Diagram Based Approach

Linzhang, et al. [5] have proposed an approach to generate test cases using UML activity diagram. They have used gray-box method [54] for test case generation from activity diagram directly, where the design is reused to avoid the cost of test model creation. Each of the test scenarios are used to extract the information for test case generation, i.e. input/output sequence and parameters, the constraint conditions etc. They have advocated the category-partition method for possible generation of all inputs/outputs. They have developed a prototype tool named as UMLTGF to execute the

process. Mingsong, et al. [37] have proposed an automatic approach of test case generation for JAVA based system, using UML activity diagram. They have used UML activity diagram as design specification. At first, abundant test cases are generated randomly for a JAVA program under testing (PUT), which then executed with the generated test cases to get the program execution trace. A comparison between these traces and the given activity diagram is conducted to get a reduced test case set which meets the test adequacy criteria. Hartmann, et al. [15] have proposed an automatic approach for test case generation and execution. In this method they have considered the textual-use case specification method to describe the specification, which is then executed with the formulated test development environment (TDE). TDE is a plug-in to Rational Rose via the rose extensibility interface, for test case generation in extensible markup language (XML) format. These test cases are then converted to a set of executable test scripts by means of an appropriate extensible style sheet language (XSL) template and executed using the commercial *user interface (UI)* test execution tool. Briand and Labiche [16] described the TOTEM (Testing Object-OrienTed systEms with the unified Modeling language), a system test methodology. Functional system test requirements are derived from UML analysis artifacts such as: use cases, their corresponding sequence and collaboration diagrams, class diagrams. And from OCL expressions, across all these artifacts are used to transform the functional requirements to test cases, test oracles and test drivers.

3.2.6 Combined Approach

As we know UML diagrams are described by different views and different diagrams are used to represent each views. So, combined use of different UML diagrams, to generate the test case will lead to clear understanding of the problem. And so ease to design the test case. In this context, we have studied the combined approach of testing using UML diagrams.

Andrews, et.al [26] described several test adequacy criteria for testing executable forms of UML. The criteria proposed for class diagrams include association-end multiplicity criterion, generalization criterion and class attribute criterion. The interaction

diagram criteria like condition coverage criterion, full predicate coverage criterion, each message on link criterion, all message paths criterion and collection coverage criterion are used to determine the sequences of messages that should be tested. They also described a test process but do not discuss any automatic test case generation method. Mayrhauser et. al [55] developed an approach for generating system (black box) test cases using AI Planner (Artificial Intelligence planner). They use class diagram and simple state diagram expressed in the UML to represent the conceptual architecture of the system under test. They developed a representation method at the application domain level that allows both the statement of test objectives at that level, and their mapping into a planner representation. Their method maps the initial and goal conditions into a problem description for the planner. The planner generates a plan based on this input. In the next step, they do a simple conversion of plan to produce executable test cases. The purpose of a test case in a goal directed view is to try to change the state of the overall system to the goal state. The planner decides which operators will best achieve the desired goal states. Cavarra et.al [56] described how to translate UML class diagrams, state chart diagrams and object diagrams into a formal language to characterize the behavior of the system. They have reported that this specification called IF (Intermediate Format) can be used as a basis for test generation. The behavioral descriptions are written in a language of communicating state machines. From this they form a test graph consisting of all traces leading to an accept state, corresponding to a pass verdict with branches that might produce an inconclusive result. Automatic generation of test causes from this graph was not addressed.

3.3 Test Case Optimization

We can visualize the problem involves in testing processes as an optimization problem and solvable with genetic algorithm. Let us consider the different attributes involved in testing process and their establishment with respect to optimization. We can see from the Fig. 3.1, the domain data that serves as input to the GA application. We can distinguish them as boundary data set, input data set for user profile and set of test

cases generated according to the user profile. First source of input domain data is data set that represents the the extreme values called as boundary data set, for the software under test. For example, suppose the system accepts some data value "Money" in ATM withdrawal. Then the input boundary data might specify 100 < X < 10,000. Second, there is data set that represents the users profile. This data defines what input data the user is likely to use and not to use. For example, user can specify the range of the data value "Money" of previous example as 500 < Money < 1000. The third is the set of test cases generated according to the user profile. For example, there may be 3 test cases that specify "Money" as 600, 700 and 900.

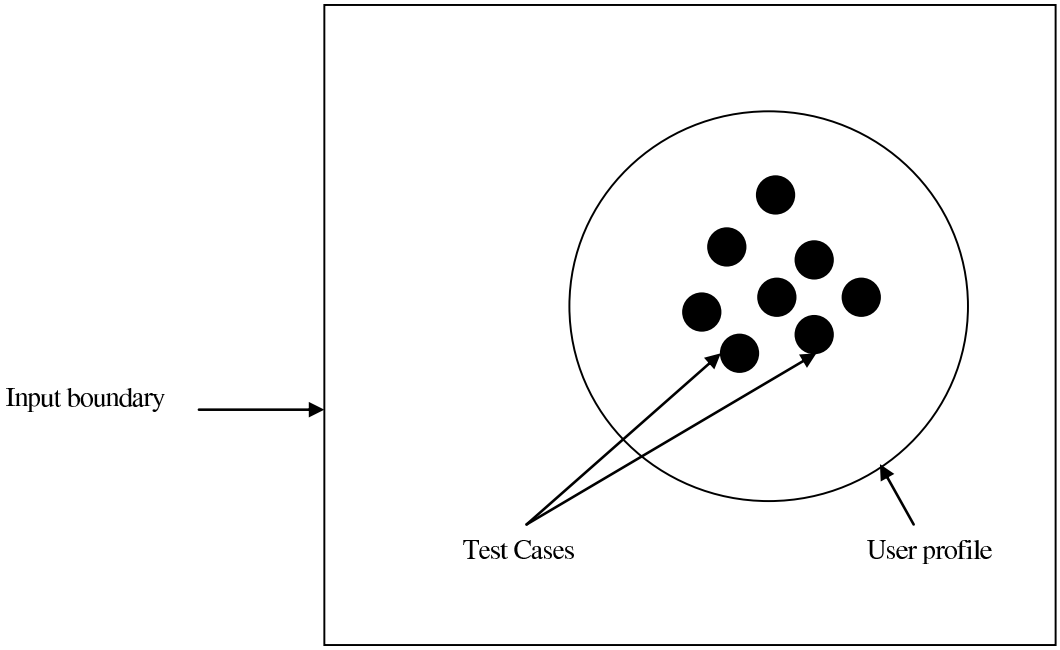

Figure 3.1: Input Domain Data for Testing using GA

We can see from the above discussion that, the test cases are used to initialize the population where as the user profile data set is used to evaluate the fitness of individuals, specifically used to determine the likelihood of occurrence.The input boundary data set is used to validate that new individuals are consistent with what the software under test allows the user to do. If an individual is created that lies outside of the defined input boundary data set, then that individual will be discarded by the genetic algorithm. So the testing process can be optimized with respect to reduced test case set, minimization of error probability or maximized failure intensity (defined as a combination of failure

density and failure severity) [57].

In earlier days, meta heuristic search has been prominently used for automatic generation of test data for structural and functional testing, grey-box and non-functional properties testing. McMinn [58] has studied several papers on development of *meta-heuristic* search techniques and their application to automatic test data generation. In this survey paper, he has elaborately discussed on different techniques of test data generation for structural and functional testing using *metaheuristic* technique. He is foresighted with large scope for search-based non-functional testing, stress testing and security testing in conclusion. Automatic generation of test data, using state diagrams and genetic algorithms is proposed by Lefticaru and Ipate [59]. They have presented the system with *Finite State Machine (FSM)* and applied GA to uncover the possible errors in the implementation, such as erroneous transition labels, erroneous next-states, missing states, extra states, etc. They have also discussed the most general approach *W-methos* [60] for generation of test sequences. An automatic approach for automatic generation of structural state data is advised in [61]. In this approach, they have used a dynamic optimisation-based search technique for the required test-data. The algorithm is driven by a *cost function*, which measures the *goodness* of test data. In their approach, simulated annealing search technique is used for test data generation. Pargas, et al. [62] presents a goal oriented technique for automatic test data generation using genetic algorithm, which is guided by the control dependencies in the program. They have represented their instrumented version of the program with the help of control dependency graph. The proposed approach can handle test-data generation for programs with multiple procedures. Most of the above discussed techniques consider test suite reduction as a single objective optimization problem [63, 64], while recent work considers it as multi objective problem [65].

3.4 Summary

In this chapter, we briefly discussed on OOT and its complexity involves to testing, automatic test case generation strategies and test case optimization. We studied several

approaches on automatic test case generation using UML diagrams especially Activity, Collaboration and Sequence diagram based, available in research papers. This chapter is keen to give an eye sight to the existing work on the model based system testing using UML diagrams. Several attempts have made on the automatic test case generation approach but those are partially or semi automated approach and also inadequacy to the complex system to some extent. Test case optimization is also essential and feasible with respect to MBT of object-oriented software.

Chapter 4

Test Case generation using Activity Diagram

It is well known that software testing is a time-consuming, error-prone and costly process [37, 66]. Therefore, techniques that support the automation of software testing will result in significant cost and time savings for the software industry. Automatic generation of the test cases is essential for the automation of software testing. Once the test cases are generated automatically, a software product can even be tested fully automatically through a test execution module, to realize an integrated automated test environment. Automatic approach for test case generation will be not so much productive, if we wait for the generation of test cases at the end of the development. That means it is simply a waste of time, if we follow the source code based testing. So, automatic test case generation using design document (or system specification or model) is more reasonable. In this context, we have proposed an automated framework for test case generation. We have also proposed an algorithm for test case generation using UML activity diagram, in the scenario of model based testing (MBT).

In this chapter, we use UML activity diagrams as design specifications and consider the automatic approach to test case generation by extending [4]. UML diagrams are classified on the basis of the structural or behavior aspects of systems, i.e. whether they are intended to describe the structural or behavior aspects of systems. UML activity diagrams [6, 67] describe the sequential or concurrent control flow of activities. They can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. Our approach first constructs the activity diagram for the given problem and then randomly generates the initial test cases for a PUT [14]. Randomly gener-

ated initial test case helps to generate best test case using *heuristic rule* by satisfying the path coverage criteria. Our approach is interested to develop a technique that will automatically generate test cases with maximum path coverage.

The rest of the chapter is organized as: Section 4.1 illustrates the basic concepts and definitions. Section 4.2 presents our proposed frame work. Proposed Test case generation methodology is described in Section 4.3, followed by working of our algorithm in Section 4.4. Section 4.5 presents the conclusion of the chapter.

4.1 Basic Concepts and Definitions

In the following subsection, we will define some useful concepts and terms related to this chapter.

4.1.1 Activity Diagram

UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe structural or behavioral aspects of systems. Activity diagrams also describe the sequence of activities among the objects involved in the control flow during implementation. Activity diagrams are similar to procedural flow charts. But the major difference between them is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities. Before presenting the detailed procedure to generate test cases using UML *activity diagram*, we need to define the *activity diagram*.

Definition:

An activity diagram is a eight tuple, which is given by ACD = (A, B, F, J, K, T, C, a_0), where

- A = { a_1, a_2, ..., a_n} is a finite set of activity states.
- B = {b_1, b_2, ..., b_m} is a finite set of branches.
- F = {f_1, f_2, ..., f_q} a finite set of forks.

- J = {j_1, j_2, ..., j_r} a finite set of joins.
- K = {k_1, k_2, ..., k_p} a finite set of final states and end flows.
- T = {t_1, t_2, ..., t_s} a finite set of transitions and $t_s \in$ T.
- C = {c_1, c_2, ..., c_v} is a finite set of guard conditions.
- a_0 is the only initial state and $a_0 \in$ A

The above descriptions are shown in below.

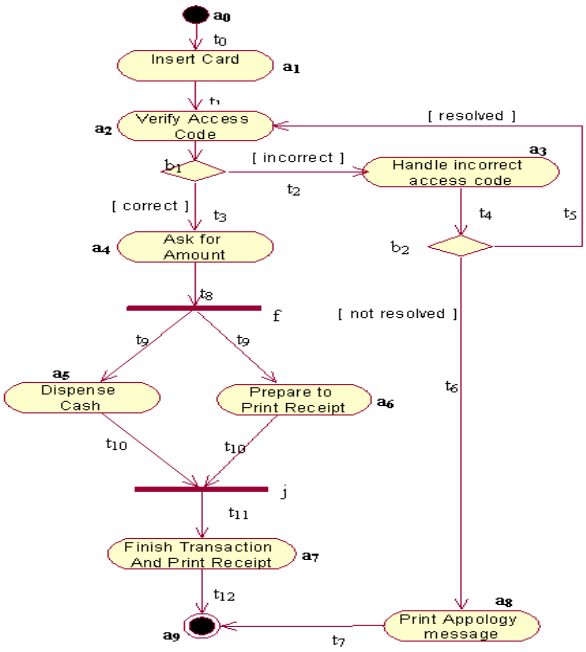

Figure 4.1: An illustration of an Activity diagram

4.1.2 Test Adequacy Criteria for Activity Diagram

Problem specification is the key factor to get the accurate result, which is very much important. Therefore, there is a pressing need for specification of test adequacy criteria, before going to follow the software testing procedure. The adequacy criteria of activity diagrams are based on the matching between the paths of activity diagrams and program execution traces of the implementation codes.

The description about test adequacy as a measurement function is given in [14, 22]. Suppose *p* is a program, and *tcs* be the test case set. The test adequacy criteria, to generate test cases for an activity diagram are given below:

- *Activity coverage:* According to this, all activity states in the activity diagram should be covered. For any $t \in tcs$, we can get the *program execution trace* (*PET*). If there exists any function in *PET* whose corresponding activity is not *marked* in the activity diagram, we mark all the corresponding unmarked activities of *PET* and record the test case *t*. So, the value of *activity coverage* is the ratio of the marked activities to all activities in the activity diagram.

- *Transition coverage:* All transitions in the activity diagram must be covered. For any $t \in tcs$, we can get the *PET*. If there exists any function in *PET* whose corresponding transition is not *marked* in the activity diagram, we mark all the corresponding unmarked transitions of *PET* and record the test case *t*. So, the value of *transition coverage* is the ratio of the marked transitions to all transitions in the activity diagram.

- *Path coverage:* All paths in the activity diagram must be covered. For any $t \in tcs$, we can obtain the *PET*. If there exists any function in *PET* whose corresponding path is not *marked* in the activity diagram, we mark all the corresponding unmarked path of *PET* and record the test case *t*. So, the value of *path coverage* is the ratio of the marked paths to all paths in the activity diagram.

Definitions:

Traverse Path:

Traverse path is a queue type data structure, which is used to store the traversed path of activity diagram. It is symbolized as TravPath[i].

Program Execution Trace Path:

This path is called as program execution trace path. It is also maintained by queue data structure, which is used to store the path covered by executing the PUT with the randomly generated test cases. It is symbolized as PETPath[j].

4.2 Proposed Frame work

The schematic outline of the automatic test case generation strategy is described in the model given in Fig.4.2. Below, we explain the important components of our proposed

model.

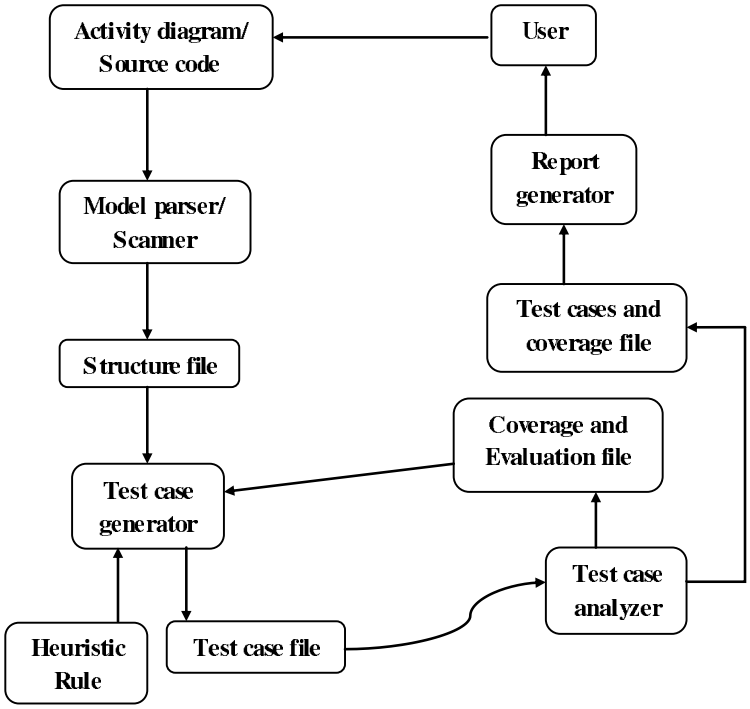

Figure 4.2: System model of our approach

Model parser/ Scanner

The purpose of the model parser is to keep the path traversal details of the activity diagram.

Test case generator

The test case generator produces new test cases that would cover the target branches/conditions in the code from the structure file and determines what conditions/branches should be targeted for new test case generation.

Test case analyzer

Test case analyzer evaluates by running each test case in the program and maintains a track of condition and branch coverage. If the test case satisfies the coverage criteria it generates a report otherwise the analysis result is used by test case generator for further test case generation.

Report generator

The report generator prints the result which includes the generated test cases, condition and branch coverage and percentage of path coverage.

4.3 Proposed Test Case Generation Methodology

In this section, we discuss our work to generate test cases from UML activity diagram. We have described a rule based technique called as *Simple Heuristic Rule or Heuristic Rule* for our application, to generate test case by analyzing the path covered in the activity diagram, satisfying all path coverage criteria. In the subsequent part of the chapter, we may call this technique as *heuristic rule*. The detailed description of our approach is presented in the following subsection.

4.3.1 Paths in the Activity diagram

The selection of path coverage in test case generation is a very complex task. When a path in the activity diagram is matched, we delete this path from the path coverage set. Hence the matching process for activity diagram will terminate when the path coverage set is empty. The algorithm for simple path searching is given in [68]. The complexity in path selection is due to the presence of synchronization, concurrency and loops. Our approach only considers the paths for selecting the program execution traces, which satisfies the semantics of the synchronization such as the *join* and *fork* in the activity diagram. Loops in an activity diagram may result in a path with infinite activities. From Fig. 4.1, we derived the following paths:

start <a_0>,‹<a_1>,

< a_1 >‹<a_2>,

< a_2>‹<a_3>,

< a_2>‹<a_4>,

< a_3>‹<a_8>,

<a_8>‹<a_9>,

<a_4>‹<a_5, a_6>,

< a_5, a_6 >‹<a_7>,

<a_7>‹<a_9> end

We have considered simple path to avoid the complexity due to loops and concurrency, which is beyond the scope of the discussion.

4.3.2 Proposed Approach

First, we construct the activity diagram for the given problem. We are using UML activity diagram to generate the test cases. Next, we use a randomly generated test case [69] as the initial test case is to get the program execution traces for a PUT. Then, by applying a "Heuristic Rule" we get the best test case. At last, by comparing the execution traces with the constructed activity diagram satisfying specification criteria, we get the reduced test cases which meet the test adequacy criteria.

Our main goal of the approach is to cover the maximum path based on maximum branch coverage. So, first we consider to maximize coverage of branch. In this context, we target the uncovered branches, on which our heuristic rule is applied to cover. This can be done by modifying the input value based on which PUT is being executed. Now, we presented our "Heuristic rule" as:

$$\frac{1}{n}\{\frac{|LHS(t_1) - RHS(t_1)|}{2 * max(|LHS(t_1)|, |RHS(t_1)|)}\} \tag{4.1}$$

where, t_1 is a test case, LHS (t_1) and RHS (t_1) represent the evaluated value of LHS and RHS, respectively, when t_1 is used as the input data and n is the number of branches covered. The above equation is not sufficient to change the branch status from uncovered state to covered. The *goodness* of test case is dependent on the changing value of that input, at the given guard condition. The smallest value indicates the best test case. Now, we defined the formula to measure the goodness of the input (or test case, which is also a test input) in below:

$$G(t_1, C) = w * L(t_1, C) + (l - w) * P(t_1, C) \tag{4.2}$$

Where, G(t_1,C): Goodness of test case t_1 at condition C.

L(t_1, C): Freedom space of t_1 at C.

P(t_1, C): Sum of freedom space of t_1 along the path toward C.

w: Weighting factor between L(t_1, C) and P(t_1, C), 0 < w < 1.

L(t_1, C) is defined as in 4.1, and P(t_1, C) is defined as:

$$P(t_1, C_i) = \sum C_i(m * L(t_1, C_i)) \tag{4.3}$$

Here, C_i is a condition that is on the path toward C, and m is the total number of these conditions. Although this definition does not represent the actual distance of test case t_1 to a boundary, it is a reasonable approximation. Both these definitions are derived heuristically.

4.3.3 Test case generation strategy

We use a *Heuristic Rule* to achieve the maximal branch coverage. A branch coverage analysis is required to get the *best test case* (BCASE). The branch coverage status of the code is recorded in a coverage table. When a branch is covered by any test case, the corresponding entry in the table is marked with a "✓". The target of the test case generation is to mark all entries in the table. Therefore, the partially covered branches are the main targets for modification, to cover all paths. The uncovered conditions at branches will not be targeted for new test case generation. Earlier test cases can be used as models for new cases, because, no test case model yet exists that can be used for modification. The main problem arises to select a model test case when, more than one test case drives the same path. So it is very essential to identify the goodness of a test case. We define the goodness of a test case by using the above heuristic rules.

Here, we have considered a typical format of an IF-THEN statement where the *expression* (exp) can be expressed in the form of: LHS<op>RHS. The *goodness* of a test case t_1 relative to a given condition can be calculated using Eqn. 4.1. This measures the closeness between LHS and RHS of Eqn. 4.1 [70]. When this measure is small, it is generally true that a slight modification of t_1 may change the truth value of *exp*, thus covering the other branch. A test case that yields the smallest measurement is considered to be the best test case of the condition under consideration.

4.3.4 Our Proposed Algorithm

In this subsection, we present our proposed algorithm in pseudo code form to get the reduced test case. The pseudocode format of our algorithm is as: begin

BCASE=F, OP =Null; //BCASE= best test case, a boolean variable and OP is output.

TravPath[i]=Trav(AD); // AD is activity diagram

PETPath[j]=Null;

Supply AD and RTC to TCG as an input; // RTC is randomly generated test case, TCG is test case generator.

Level 1: Execute PUT with RTC to give *PET*; // PET is program execution traces.

if (PETPath ≠TravPath)

{

Apply heuristic rule on RTC to TCG for generating best test case;

Go to Level 1;

}

else

{

OP=RTC;

BCASE=T;

end;

4.4 Working of our algorithm

We have considered Super Market Prize Winner system as a case study for our approach. We have implemented using IBM Rational Rose Version 7.0.0 and JAVA. We have considered some manual analysis of UML activity diagram, since our approach is not fully automated.

Example 1: Super Market Prize Winner Scheme

We have considered super market prize winner scheme as a case study to analyze our approach. The detailed description of the working of the super market prize winner is as follows:

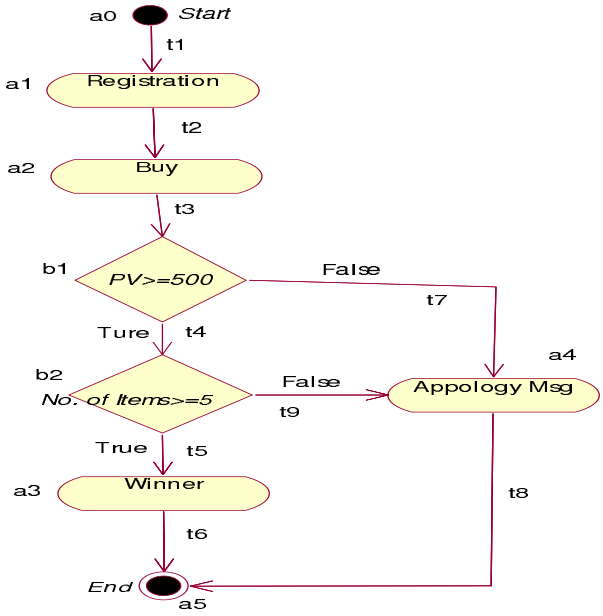

Figure 4.3: Activity Diagram for Super Market Prize Winner System

Table 4.1: Generated Test Cases for Super Market Prize Winner System

TC-ID	Test Scenario	Input	Expected Output	Observed Output
1	< Start, Registration, Buy, b1, Apology Msg, End >	< 400, 7 >	Not Winner	Not Winner
2	< Start, Registration, Buy, b1, b2, Apology Msg, End >	< 700, 4 >	Not Winner	Not Winner
3	< Start, Registration, Buy, b1, b2, Apology Msg, End >	< 600, 4 >	Not Winner	Not Winner
4	< Start, Registration, Buy, b1, Apology Msg, End >	< 490, 7 >	Not Winner	Not Winner
5	< Start, Registration, Buy, b1, b2, Winner, End >	< 500, 7 >	Winner	Winner

The customer has to registered with super market prize winner scheme and get a valid card to purchase the items under the scheme. Every purchased value from the super market store will be recorded with the registered customer's id. A customer will be declared winner of the scheme when a total purchase value from the super market store exceeds or equal to rupees 500 per day and the purchased items should be more or equal to 5. We represented the above scheme by an activity diagram in Fig. 4.3 The performance of our proposed approach is studied with an example of super market prize winner scheme. In Table 4.1 and Table 4.2, we present the result of our proposed approach.

Table 4.2: Branch Coverage and Path Coverage using our approach

TC-ID	Branch Covered	Path Covered	Branch Covered (By our Approach)		Total Path Covered (By our Approach)
			Total Branch Covered	Branch Covered (%)	
1	1	5	2	100	6
2	2	6	2	100	6
3	2	6	2	100	6
4	1	5	2	100	6
5	2	6	2	100	6

4.5 Conclusion

We have proposed an approach to generate test cases for *object-oriented programs* by using UML activity diagrams. We have used a heuristic rule to obtain the reduced test cases, which satisfy path coverage as the test adequacy criteria. In this chapter we have considered only the simple path for automatic test case generation. Our approach achieves the maximum branch coverage and path coverage, which is an added advantage. Our approach is fails to handle the large and complex system. This approach is very much suitable for simple systems where no more fork-joins, like nested-fork joins and etc.are involved, which is our next objective. However our proposed system is not sufficient to handle different kind of errors such as *work flow errors, state based errors and etc.* To overcome this bottleneck, a combined approach is essential and hence we have used the multiple UML diagrams such as *Activity, Class and Sequence diagram* in the next subsequent chapters.

Chapter 5

Test Case Generation using Activity, Sequence and Class Diagram

As the complexity of systems is increasing gradually, more systems perform mission-critical functions and dependability requirements such as safety, reliability, availability, and security are vital to the users of these systems. The competitive marketplace is forcing companies to define or adopt new approaches to reduce the time to-market as well as the development cost of these critical systems. In this context, it is highly desirable to analyze the system carefully to meet all the aspects of the system by using UML diagrams such as *Activity, Class and Sequence diagram*. Close insight to the requirement helps in finding the different stubs (or critical functionalities) in and design the system accordingly, lead to an ease the testing process. So, we have followed a scenario based testing process for test case generation from design specifications.

In this chapter, we have proposed an algorithm to generate test cases from scenarios of a system under testing. We first generate test scenarios from activity diagrams, which achieve path coverage criteria perfectly, followed by generation of test cases by analyzing the respective sequence and class diagrams of each scenario. Our approach achieves maximum path coverage. Also in our approach, the cost of test model creation is reduced as the design is reused.

The rest part of the chapter is organized as follows: Section 5.1 gives a brief idea about the basic concepts and definitions, which we will use in rest of the chapter. Section 5.2 presents our proposed for test case generation methodology. Section 5.3 present the working of our algorithm and its result and Section 5.4 concludes the chapter with conclusion with further scope of our work .

5.1 Basic Concepts and Definitions

Scenario-based testing is a software testing activity that uses scenarios for tests, or simply scenarios, which are based on a hypothetical story to help a person think through a complex problem or system. They can be as simple as a diagram for a testing environment or they could be a description written in prose. These tests are usually different from test cases in that test cases are single steps and scenarios cover a number of steps. Scenarios are also useful to connect to documented software requirements, especially requirements modeled with use cases. More complex tests are built up by designing a test that runs through a series of use cases. Within the Rational Unified Process, a scenario is an instantiation of a use case. Now we describe some terminologies related to this chapter.

Fork:

It represents the splitting of a single flow of control into two or more concurrent flows of control for performing the parallel activity. A *fork* may have one incoming transition and two or more outgoing transitions and each of the activities associated with each of these paths continues in parallel.

Join:

A *join* is just like a thick bar used to represent the synchronization of two or more concurrent flows of control. It may have two or more incoming transitions and one outgoing transition. At the *join*, the concurrent flows synchronize that means each flow waits until all incoming flows have reached at the *join*. From the *join*, one flow of control continues further.

Path Coverage:

It is one of the coverage criteria used for achieving the adequacy of testing. A path is the logical sequence of executable statements of a component from an entry point to an exit point and each path of the activity should be covered at least once.

Category-Partition Method (CPM):

It is simply a specification based testing technique with respect to some specific criteria, which was developed by Ostrand and Balcer [50]. CPM first decomposes the functional

specification into functional units and then examines each functional unit. It finds the categories for each parameter and environmental condition. It helps in identifying the parameters and environmental conditions that affect the execution behavior of the function. CPM is also helpful in finding the categories of information that characterize each parameter and environmental condition.

5.2 Proposed Test Case Generation Methodology

We use Activity diagram to generate the scenarios of the system followed by respective sequence diagrams. After that we analyze the sequence diagrams to find the interaction categories and then use the corresponding class diagrams to find the settings categories. The detailed description of our approach is presented in the following subsection.

5.2.1 TC-ASEC: The Proposed Approach

In this section we propose an approach to generate test cases from design models using activity diagram, sequence diagram and class diagram called as TC-ASEC(*Test Case Generation using Activity, Sequence and Class*). In the proposed scheme we are using gray-box testing method, where the advantages of both black-box and white-box testing are combined together. The generated test cases extend the logical coverage criteria of white-box testing and finds all possible paths from the design model which describes the expected behavior of an operation. In our approach, we have used activity diagrams as test models. First of all, our approach parses the activity diagram and generates the test scenarios which satisfy the path coverage criteria. Activity diagrams represent the implementation of an operation like the flow chart of code implementation. The executing paths are derived directly from the activity diagrams, as the executing path is a possible execution trace of a program. We have considered path coverage in our approach, since it has the highest priority among all the coverage criteria for testing. Our approach also handles the complexity of nested fork-joins using a criterion that checks whether the target activity state of a transition is a fork or an activity state. If the target of the transition is a fork, then the fork has higher priority over the activity state. So it should be considered first and then only the other path is considered. As a

consequence of this priority criterion, the complicated nested fork-join pair is handled properly in our approach. After all the possible test scenarios are generated, we generate the corresponding sequence diagram, and class diagrams for each scenario. Now using category partition method, we analyze the functional requirements to divide the analyzed system into functional units, to be separately tested. For each defined functional unit, the environmental conditions (system characteristics of a certain functional unit) and the parameters (explicit input of the same unit) relevant for testing are identified. Then, test cases are derived by finding significant values of environmental conditions and parameters. The approach is described in Fig. 5.1

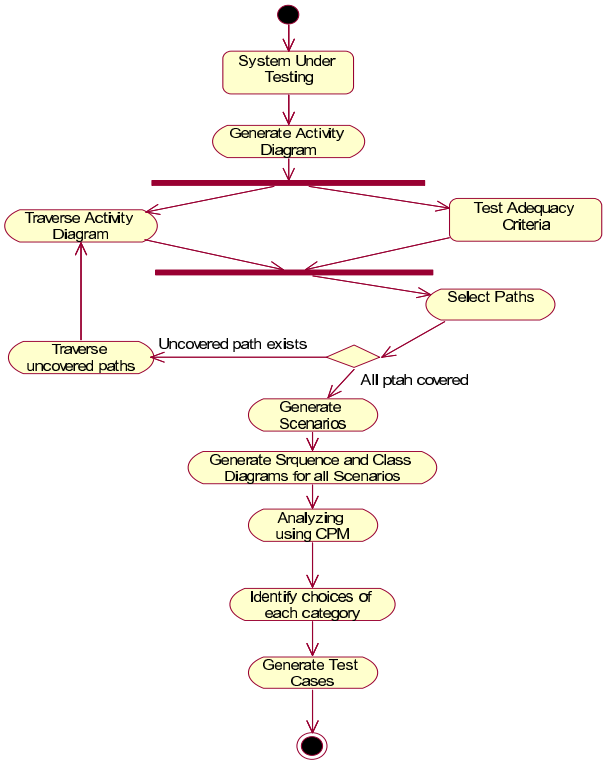

Figure 5.1: Our Proposed Approach

5.2.2 Test Scenario generation: TSAD

In order to generate test scenarios from activity diagram (TSAD), we have considered all the activities, decisions, forks and joins as nodes. Our approach traverses the activity diagram using modified depth first search (DFS) method. In order to traverse the activity diagram from initial node to final node, our approach visits all the current nodes and the

corresponding transitions released from the current node. Next, a record of the trace of a run of the executing path of activity diagram is maintained by recording the visiting trace of the current nodes and transitions. Each loop present in the activity diagram is executed at most once covering the corresponding activity states and transitions. A loop is bypassed in the sequence if it is already considered earlier. We have proposed an approach to generate test scenarios from UML design models. We presented below our approach in pseudo code format.

Input:D = (A, T, F, J, C, a_I , a_F); //where D is the 6 tuple represents activity diagram.
CoveredNode[ ] is an array of the occurred times of *CN*
CoveredTrans[ ] is an array of the visited times of *t*
s is a stack to record the covered CN and occurred *t*
output: *ts*[n];
begin
i = 0; *j* = 1; *ts* = null; *coverednode* =0; *coveredtrans* =0; *CN[i]* = a_I;
while $CN[i] \neq NULL$
push(*CN*[*i*], *s*);
coverednodeCN[*i*] = *coverednodeCN*[*i*] + 1;
if(possible$CN[i] \neq NULL$);
checknodeprioritCN[*i*];
checknextCN[*i*];
else
Read out the stack *s* from bottom to *top* to *ts[j]*;
j=j+1;
if(all the *CN*[*i*] and *t* at least shown one time in *ts*)
exit;
else
While($s \neq NULL$)
t=pop(s);
CN[*i*]=*pop(s)*;
if(possible $CN[i] \neq NULL$)

checknodepriorityCN[*i*];

checknextCN[*i*];

end if;

end while;

end else;

end else;

end while;

end;

Pseudocode for ***checknextCN[i]***:

begin

t= next unvisited transition in possible*CN*[*i*];

possible*CN*[*i*]= possible(*CN*[*i*] t);

$CN[i+1] = (CN[i] - prenode(t)) \bigcup postnode(t)$;

$i = i + 1$;

if(*coverednodeCN*[*i*]) == 2

checknext*checknextCN*[*i*];

end if

push(coveredt[t], s);

coveredt[t] = coveredt[t] + 1;

end;

Pseudocode for ***checknodepriority(CN[i])***:

begin

n = *postnode(t)*

if($n \in F$)

CN[i]=n+1;

else

CN[i]= n;

end if

end;

The *TSAD* visits *current node CN[i]* from the initial *activity state(CS[0] = a_I)* to the

final *activity state(CN[n] = (a_F))*, and transitions fired from it in turn. A stack *s* is employed to sequentially record the visiting trace of *CN*s and *transitions*,which is also the trace of a run of the activity diagram. After the algorithm is initialized, *CN[i]* is pushed into the *top* of *s*, and its occurring time in the stack is set in the flag array. Before going to next node the priority of the node is checked, if the target of a transition is a fork then it is given lower priority then the node having fork is given lower priority. This checking is done each time *TSAD* visits the next node. When the possible *CN[i]* is not empty, one possible *transition t* is chosen and fired from *CN[i]*, and its occurring time in the stack is set in the flag array. After that *t* is deleted from possible(*CN[i]*), the corresponding guard conditions are pushed into *top* of *s*. The new state set *CN[i + 1]* could then be calculated by deleting the pre state of *t* from *CN[i]* and merging the post state of *t* into *CN[i]*. If a loop has been executed once, i.e. the occurring times of *CN[i]* in s equal to two, it is bypassed in the sequence. If *CN[i]* in the *top* of the stack *s* is empty, a full path is completed. We can read out a test scenario from the bottom to the *top* of the stack into a test scenario *ts[j]*. Then the algorithm backtracks to the last visited *CN[i]* with an unvisited fired transition in the enabled (*CN[i]*) and continues the traverse. This progress continues until all the current states set and transitions of the activity diagram were found at least one time in the set of test scenarios. In Fig. 5.2 activity diagram for ATM Withdrawal is presented. We also presented the state of the system in Table. 5.1, which are goes on changes by firing the events and this is essential to generate the test scenarios.

5.2.3 Test Case Generation

After all the test scenarios are generated, we analyze the corresponding sequence diagram for each selected scenario. Each sequence diagram is composed of objects and the messages they exchange. The objects involved in sequence diagram realize and execute the functionalities described in the scenario through elaboration and message exchanges. In this phase, class diagrams are also considered, as class diagram defines operations and attributes required for the interactions of their objects. In our approach, we have applied category- partition method on sequence diagram and class diagram for

Table 5.1: State of the system for test scenario generation

Input	Operation status	Expected output
	ATM.Start state	
Valid card	ATM.Insert card	Ask for PIN code
	ATM.Ask for PIN code	
PIN code	Bank.Verify PIN code (Valid)	Display menu
Amount (condition: i. Amount<= Max. Amount & ii. Amount<=Total. Balance)	ATM.Dispense Cash	Cash dispensed
	Bank.Print receipt	Printed receipt
	ATM.Finish transaction & Print Receipt	
	ATM.End state	

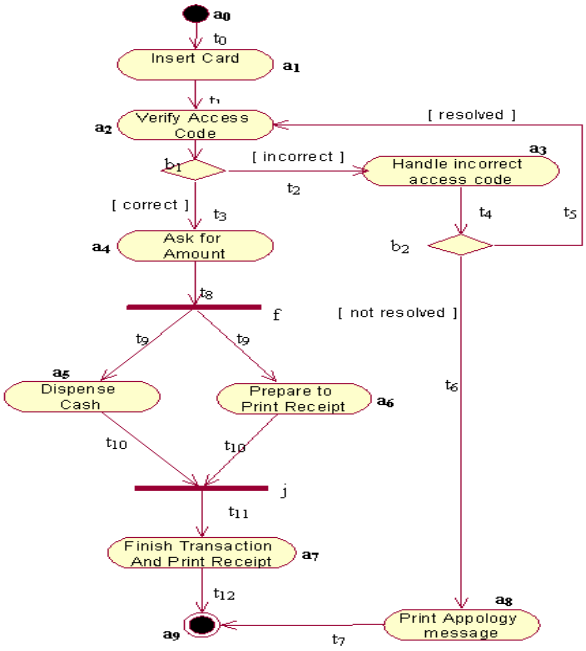

Figure 5.2: Activity diagram for money withdrawal from ATM

generating test cases. The major steps involved are:

1. We analyze the sequence and class diagrams for identifying the various parameters and environments of the function, in selected test scenario.

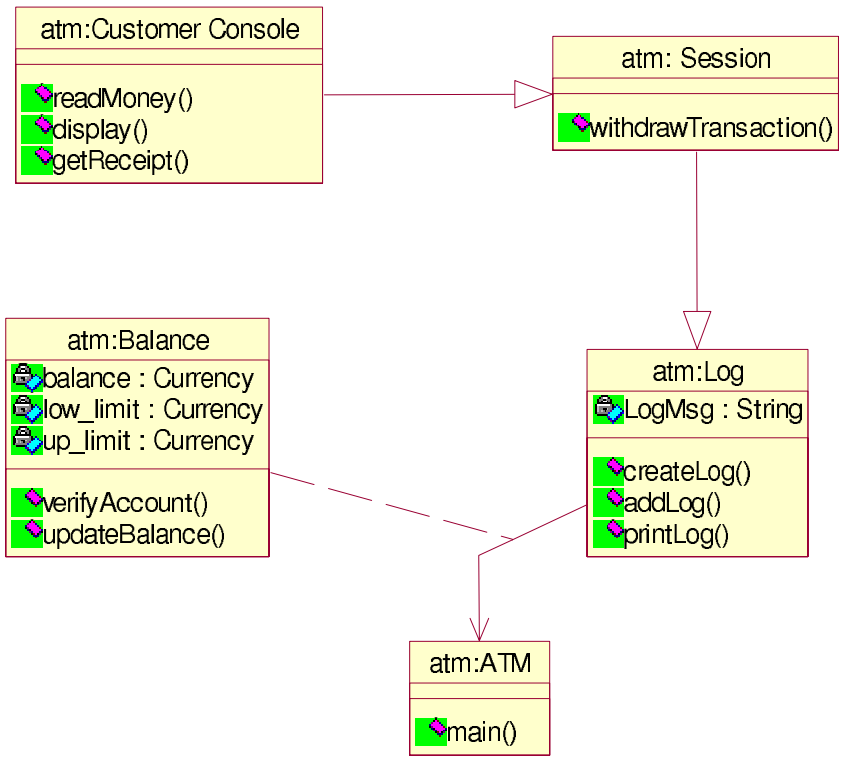

Figure 5.3: Class diagram for ATM money withdrawal

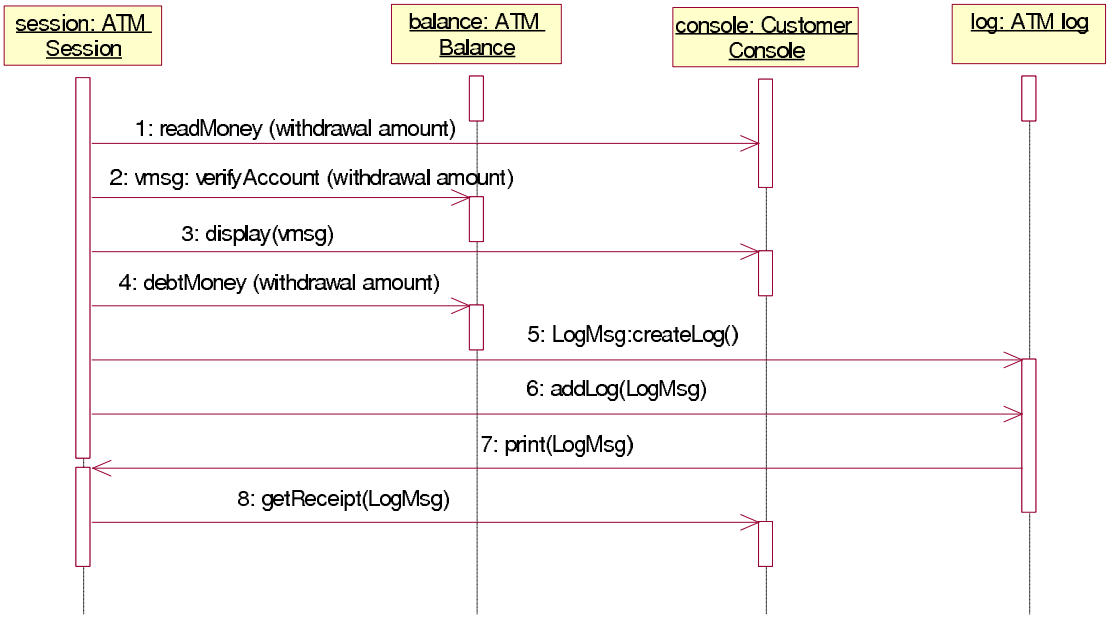

Figure 5.4: Sequence Diagram for money withdrawal from ATM

2. *Test Unit definition:* Each object inside a sequence diagram is considered as a Test Unit, since it can be separately tested and it represents and defines a possible use of system.

3. *Search of setting and interaction categories:* Interaction categories are the interactions that an object has with other objects involved in the same sequence diagram. Settings categories are attributes of a class (and corresponding sequence diagram's object), like input parameters used in messages or data structures.

4. *Test Case construction:* After both the categories are identified for each test unit, significant values were chosen. For each found category, its possible values and constraints are generated. For this purpose class diagram is used, where a preliminary description of a method implementation and its possible input values (or the description of an attribute used and its significant values) are found. By considering all the potential combinations of compatible choices, we derive the test cases. Finally, for each test scenario, all the possible test cases are generated.

5.3 Working of our algorithm

This section discusses the results obtained by implementing the proposed Approach. We have implemented the complete approach using JAVA Swing and Rational Rose Version 7.0. We have explained our approach by taking ATM (Automatic Teller Machine) as the Case Study. By applying our approach, we obtained 9 most prioritized test scenarios, only few are represented in Table. 5.2. After obtaining the scenarios, we generated the sequence diagram and class diagrams for each test scenario and the two categories. By identifying significant values for each of the categories, we have obtained the final test cases. Both the positive and the negative test cases are generated for each of the generated test scenario using boundary-value analysis. Generated test cases are given in Table 5.2 and Table 5.3.

5.4 Conclusion

We proposed an approach to generate test cases from UML behavioral diagram, where the design is reused. In this approach we have considered multiple UML diagrams such as *Activity, Class and Sequence diagram* to handle different kind of errors like *work flow errors, state based errors and etc.* Because different diagrams of UML shows the

Table 5.2: Generated Test Cases with test scenario

TC-ID	Test Scenario	PIN	Amount Entered	Total Amount(in A/C)	Amount(in ATM)	Expected result
1	Successful	4987	200	1000.00	5000.00	Withdraw success
2	Incorrect Pin (= 1 try left)	4976	n/a	800.00	5000.00	Apology Message (Incorrect PIN)
3	Incorrect Pin (= 0 try left)	4956	n/a	800.00	5000.00	Apology Message (warning card retained)

Table 5.3: Generated Test Cases for ATM Withdrawal

TC-ID	Test Scenario	Input	Expected result	Observed Output
1	$<a_0, a_1, a_2, a_4, a_5, a_6, a_7, a_9>$	<4987, 200>	Withdraw success	Successful withdraw
2	$<a_0, a_1, a_2, a_3, a_8, a_9>$	<4976, na>	Apology Message (Incorrect PIN)	Apology Message (Incorrect PIN)
3	$<a_0, a_1, a_2, a_3, a_8, a_9>$	<4956, na>	Apology Message (warning card retained)	Apology Message (warning card retained)

different views and each of these views produces different kind. By using our approach defects in the design model can be detected during the analysis of the model itself. So, the defects can be removed as early as possible, thus reducing the cost of defect removal. The major advantage of our approach is that it handles the complicity of nested fork-join pair which is more often overlooked by other approaches [37]. It overcomes the limitations of the existing approach such as nested fork-join and loops. Test coverage criteria achieved is another advantage of our approach. However, the overall approach is not fully automated. An automated tool can be developed for the proposed approach. This approach can further be extended by generating test cases for the complete system i.e. by implementing the approach for integration testing as interactions between different components are obtained from sequence diagrams. But for the complete system the test case size may pose as a challenge to the tester, as more manual analysis is involved in this approach. So an sub-optimal solution is required to address the problem. The ultimate goal will be to address testability, coverage criteria and automation issues, in order to fully support system testing activities.

Chapter 6

Optimized Test Case Generation

A good test case should have the quality to cover more features of test objective. In other words effectiveness of testing process relies on the quality of test cases not in the quantity of test cases, which in turn lingers the testing time. We can get an appropriate amount (or optimal) number of test cases of better quality, by eliminating redundant (or unnecessary) test cases. So the problem of more time consumption in testing phase can be reduced. But getting all those test cases in a rush time (time to deliver the software to the client) is a cumbersome task. Therefore, automatic generation of test cases reduces the effort of a tester and developer and so cost and time [71]. There are so many approaches used for automatic test case generation by using evolutionary computation algorithms, but they are unable to deal with the *exemplary behavior* of test case. An exemplary test case should test more than one thing, thereby reducing the total number of test cases required. Our proposed approach is more effective by covering not only *what it is intended to do but also what it is not intended to do*, by making the difference between these two activities.

In this chapter, we use UML activity diagrams and Collaboration diagrams as design specifications and develop an approach for test case generation. UML diagrams describe the different aspects of software systems depending on the activity to be performed i.e. whether they are intended to describe the structural or behavioral aspects of systems. And possibility is there that each of these aspects or views of the system may produces different kind of errors. So it will be very much useful to use the both of the diagram to tackle each of these errors. Our approach uses genetic algorithm for generation of optimized test cases, due to its simplicity and effectiveness. We, first select the

most prioritized scenario and then construct the corresponding collaboration and activity diagrams for the given problem. Then, we construct the various sequences of events for a SUT. We pass these events (as input) to the test case generator, which works using genetic algorithm with some constraint so we called it as *genetic algorithm* and we named it as (C-GA) and finds the optimized test cases, which results in the presence of minimum percentage of errors on execution. Here, we have considered only the higher prioritized scenarios to generate test cases and it is understood that the same procedure will be followed for subsequent prioritized scenarios until the resources get exhausted.

The rest of the chapter is organized as follows: Section 6.1 illustrates the basic concepts and definitions. Proposed frame work is described in Section 6.2. In Section 6.3, our proposed approach is substantially discussed. Section 6.4 presents working of our algorithm with result produced. In Section 6.5, we conclude the chapter.

6.1 Basic Concepts and Definitions

In the following subsections, we describe few basic concepts and definitions related with this chapter.

6.1.1 Collaboration Diagram

Collaboration diagram is an interaction diagram and emphasizes on structural organization of the objects that sends and receives messages. This gives the user a clear visual clue to the flow of control in the context of the structural organization of objects that collaborate.

6.1.2 Test Adequacy Criteria

Identification of good test cases not only depends upon what errors it finds, but also how errors have been defined. Based on test adequacy criteria, we can evaluate our test case. There are so many test adequacy criteria (or test coverage criteria). Some of them have been defined by Abdurazik, et al. [4] and Mingsong, et al. [37] such as message path execution (for collaboration diagram), activity coverage, transition coverage and simple path coverage (for activity diagram). Our approach uses transition coverage

as test adequacy criteria, which suggests that all the transitions triggered by an event, should be covered at least once.

6.1.3 Prioritized Scenario

The high prioritized scenario is those, which involves more number of transitions. If number of transitions are same in between two scenarios, then scenario of high priority is calculated, on the basis of complexity involves. The complexity is defined in terms of number of fork-join pairs and number of guard conditions it executes. So the prioritized scenario (SP) is calculated as:

$$SP = \sum(T + J + C) \tag{6.1}$$

where, T is the number of transitions, J number of fork-join pairs and C is the number of guard conditions.

6.1.4 Genetic Algorithm

Our approach uses genetic algorithm for generation of sub-optimal test cases and we called this algorithm as *genetic algorithm (C-GA)*. Here, we use a constraint to satisfy the transition coverage as test adequacy criteria in genetic algorithm. The test adequacy criteria is *all transition should be covered atleast once*, which is used as the stopping criterion for GA. So we called it as GA. But we have considered a special case in our approach by illuminating an ideal system like. Our illuminate ideal system is designed is like this: < Off State (Transition1) Start or Initializing State (Transition2) Active State (Transition3) End State>. The illuminated system is an generalized view of every system, as every system should have atleast these states and transitions. Thats why We called it as an ideal system. In this system, there are four states like *Off State, Start or Initializing State, Active State, End State* and three transitions named as *Transition1, Transition2, Transition3*. So these three transitions should be covered atleast once, which is used as the stopping criterion.

6.2 Proposed Frame work

In this section, we explain the important components of our frame work for optimized test case generation. Our proposed framework is given in Fig. 6.1.

Event generator

This module/device generates the possible set of events for an operation. Ex: for successful ATM withdrawal, possible sets of events from the above activity diagram are: {insert card, verify access code, ask for amount, dispense cash, prepare to print receipt, finish transaction and print receipt}

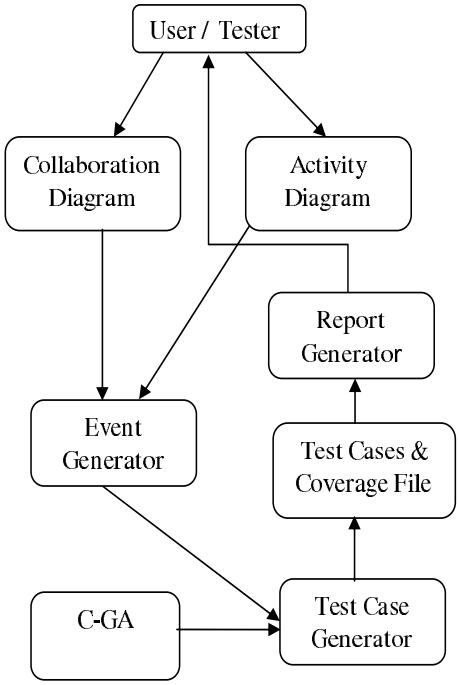

Figure 6.1: Proposed framework for optimized test case generation

Test case generator

The test case generator produces new test cases by taking set of sequence of transitions from event generator and *genetic algorithm (C-GA)* as input. Generally we have defined our test case as < *input* >, < *sequence of transitions* > and < *observed output* >. The best test cases are generated by applying the *genetic algorithm (C-GA)* and continue until reaching the stopping condition.

Test cases and coverage file

This module stores all test cases generated so far and their corresponding coverage information. A stack is used in this module to store the test cases and their coverage information. So the best test case generated so far by *test case generator* will be at the top of the stack.

Report generator

The report generator prints the result which includes the generated best (sub-optimal) test case, condition and transition coverage and percentage of error minimization. At last this printed report is submitted to user as a reference about the best test case in detail for future correspondence.

6.3 Proposed Approach

In this section, we discuss our work to generate optimized test cases by taking specification document (defined in formal method i.e. using UML collaboration and activity diagram) as input. As it is discussed in [16, 71], the automatic approach of test case generation can downplay the problem of cost and time for development of large systems. But the problem is also more cumbersome if undesirable test cases execute. Hence, we have to optimize the number of test cases and that should guarantee the presence of minimum errors.

6.3.1 Our Objective

Our approach aims to develop an algorithm to generate test cases which would be optimal and effective (high rate of error detection). Maximum percentage of I/O specification should be matched on test case execution. Testing procedure should be of low cost and effective in time consuming.

6.3.2 Methodology

Our test case generation approach consists of four parts such as: (1) I/O Specification (2) Designing the document (UML Diagram) (3) Sub-Optimal Test Case Generation (4)

Test Case Evaluation

I/O Specification

I/O Specification shows the Input and Output of the projected software in detail. This document is used as primary source of input for test case generation.

Designing the Document

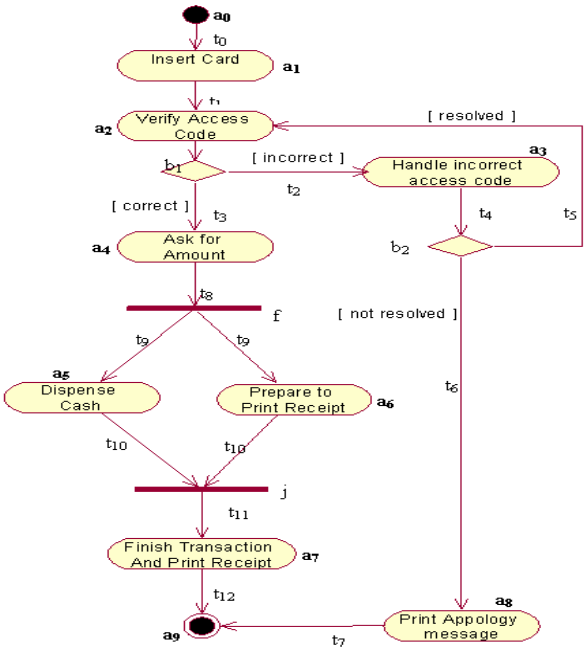

Figure 6.2: Activity diagram for cash Withdrawal in ATM

For our approach we have considered both Activity diagram and Collaboration diagram and we call them *"AC diagram"*. We design these two diagram only for the scenario of higher priority. We use collaboration diagram, because unlike sequence diagram, it shows the links among objects and sequence number of a message explicitly. Collaboration diagram is also capable of handling more complex branching. The activity diagram is used because of its dynamic behavior of modeling. We can visualize, construct, specify and document the dynamic aspects of an object. It is modeled to show the control flow of an operation (or from activity to activity). Activities ultimately result in some *action*, which is some set of pure computation. An important fact about the collaboration and activity diagrams is that they are most useful for constructing executable systems through forward and reverse engineering [14]. Fig. 6.2 and Fig. 6.3

represent the activity diagram and collaboration diagram, respectively for ATM cash withdrawal.

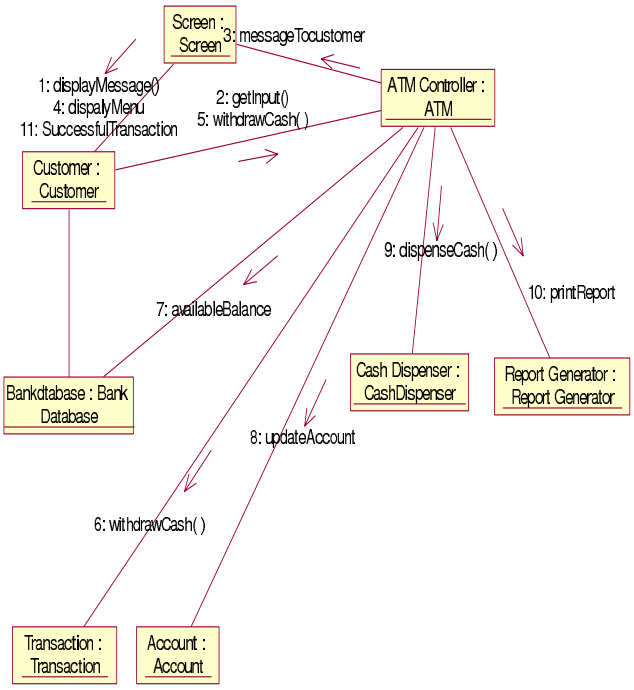

Figure 6.3: Collaboration Diagram for cash Withdrawal in ATM

Sub-Optimal Test Case Generation

The optimized test cases are generated by applying *Genetic Algorithm (C-GA)* technique on the input domain. We have considered transition coverage, a test adequacy criteria, which is defined in the Subsection 6.1.2, as the constraint. The input values are defined in the form of set of sequence of events. An event consists of a name and a list of possible arguments, which when triggered, generate transition from one activity to another. There are lots of input values for an operation. First, all the possible input values are taken in to consideration and then the C-GA is applied on these input domain, so as to minimize the input range. The obtained test cases will be used for further processing. Genetic algorithm is used for generation of better (sub-optimal) test cases. The fitness function is defined to generate the sub-optimal test case. Minimum number of errors detected measure the quality of the test case. The proposed *error minimization technique* is used to minimize the presence of errors, as we cannot guarantee the complete absence of error but we can minimize the percentage of presence of errors.

The above said technique (error minimization) is defined as *the difference between the weightage value of expected transition coverage and actual transition coverage*.

Test Case Evaluation

A pass/fail (Boolean variable such as *1* for Pass and *0* for fail) technique is deployed to evaluate the test cases. The input for this step is the optimized test case, obtained from the previous step. The test cases those will only meet all the I/O specification are considered as passed test cases, others will be treated as failed test cases.

Genetic Algorithm for our approach

Set of sequence of all events (solution/chromosome) are considered as input domain for the problem. So we have assigned a fitness (or weightage) value to each and every event or transition based on the intended activity to be performed. We have given more weighted value to those events, that involves more branches or decision. Transitions producing simple transitions are given with weightage value 1 and 0 for transitions not producing any transitions. Whereas transitions producing branches or fork and joins are assigned the more weightage value that is 2. Initially, we select randomly a valid set of transitions for the given activity. Then, we generate new solution in the next generation by performing some basic GA operations i.e. selection, crossover and mutation. The best fit test case is selected based on the calculated fitness value. The process is continued until reaching the stopping condition as defined by the user. Any successful test cannot guarantee the absence of error, rather it detects the error. So, we have deployed an error minimization technique to minimize the percentage of error presence.

The fitness value is given as:

fitness value = {*Min(error)*| *When all the transitions are covered at least once*} Eqn 1.

Min(Error) = $\sum WT_{exp}$ - $\sum WT_{act}$

Where, WT_{exp} = Weightage value of expected transitions

WT_{act} = Weightage value of actual transitions.

We have named our algorithm **OTCG-AC (Optimized Test Case Generation using Activity and Collaboration Diagram)** to generate the optimized test cases from activ-

ity and collaboration diagram using GA. Now, we present our algorithm OTCG-AC in pseudocode form.

Algorithm: **OTCG:AC**

We have taken following two assumptions:

- Activity and Collaboration diagram is given.
- Pass and Fail used here, are two Boolean variables, *1* for pass and *0* for fail. OptimalTCase and IO Specifications are represented in adjacency matrix format.

```
begin TCG // Test Case Generator
    {
    Input I/O Specification and AC Diagram; /* AC Diagram: Activity and
    Collaboration Diagram, based on high prioritized scenario */.
    Construct the set of sequence of events from the given diagram
    Level 1:Apply the C-GA and get the OptimalTCase; /* Constraint: All transitions
should be covered at least once */
    Evaluate (OptimalTCase, Input)
        {
        if (OptimalTCase = = Exact I/O Specification)
            Pass
        else
            Fail
        }
         While (TCG! = Pass or i=(n-1))
              {
              Go To Level 1;
              i++; /* i is the no. of iteration and Maximum value of i is number of events
              or transitions (n) */
              }
    TCS= OptimalTCase; /* TCS= Test Case Set, which is initialized to Null */
end TCG;
```

6.4 Working of our algorithm

We have illustrated our approach by considering four scenarios of ATM Banking system (ABS). As we have already mentioned that only scenarios of high priority, which is calculated by our definition given in Eqn. 6.1 is considered at first instant and the process will be continued until all the scenarios of the system are covered. So in this approach we have considered only four scenarios of ABS, such as ATM withdrawal, Balance Enquiry with receipt and Without receipt and PIN Verification. We presented the activity and collaboration diagrams for above said systems in Fig. 6.2, 6.4, 6.5, 6.3, 6.6, 6.7 and 6.8. We presented the result obtained by our approach by considering the above said problems such as ATM withdrawal, Balance Enquiry in an ATM and PIN verification in ATM system in Table 6.1 and Fig. 6.9 and Fig. 6.10.

Now consider the scenario of successful ATM withdrawal (from Fig. 6.2), there are many possible sequence of transitions, one of them is $< t_0, t_1, t_3, t_8, t_9, t_10, t_11, t_12 >$. We pass these transitions as an input to the TCG system, which works using C-GA. As we are going to minimize the presence of errors, only system state or sequence of events yield the events, is mostly responsible for it. In this context collaboration diagram is most essential to analyze properly the sequence of states. We have presented our chromosome (or solution, which is nothing but a system state presented in the sequence of transitions) in binary form. Initial population is derived randomly by considering some random inputs. Based on the fitness function defined in Eqn. 1, GA proceeds further in this regard for generation of sub-optimal test case. We have used Crossover Probability (P_c) as 0.5 and mutation probability (P_m) as .05. The GA runs for minimum three generation as per the definition for C-GA in 6.1.4.

Table 6.1: Result produced by our approach

SUT	Transition Covered	Error Detected	Error Recovered
PIN	6	4	4
Balance	8	5	5
Withdrawal	11	8	6
Balance(with receipt)	13	9	7

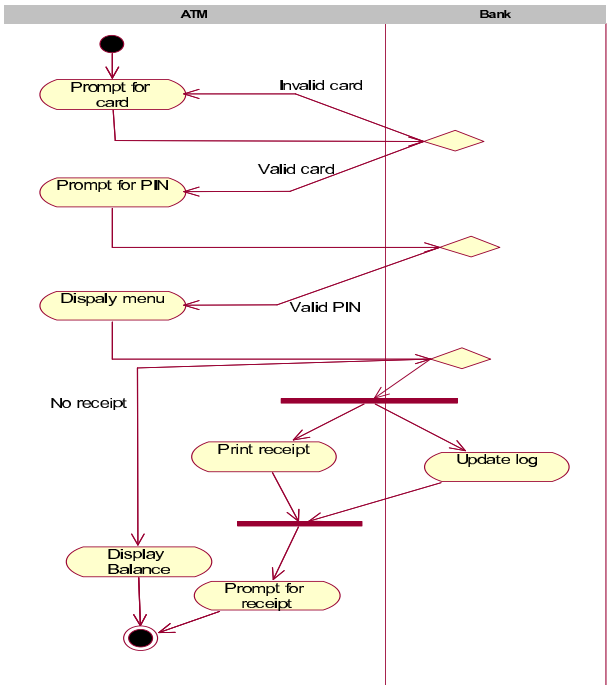

Figure 6.4: Activity Diagram for Balance Enquiry

6.5 Conclusion

We have proposed an approach to generate the test cases for *object oriented programs* from the UML collaboration and activity diagrams. We have used a genetic algorithm approach to obtain the sub-optimal (best fittest) test cases, which satisfy the test case adequacy criteria. Our approach guarantees the minimum presence of error, in the generated test case. Our approach can be further extended, to simulate our approach for real world problem along with development of test cases involving nested fork-joins and branch nested fork-joins. Though, we have proposed an automated approach for test case optimization, but it does not support fully automated approach. Some manual intervention is also essential.

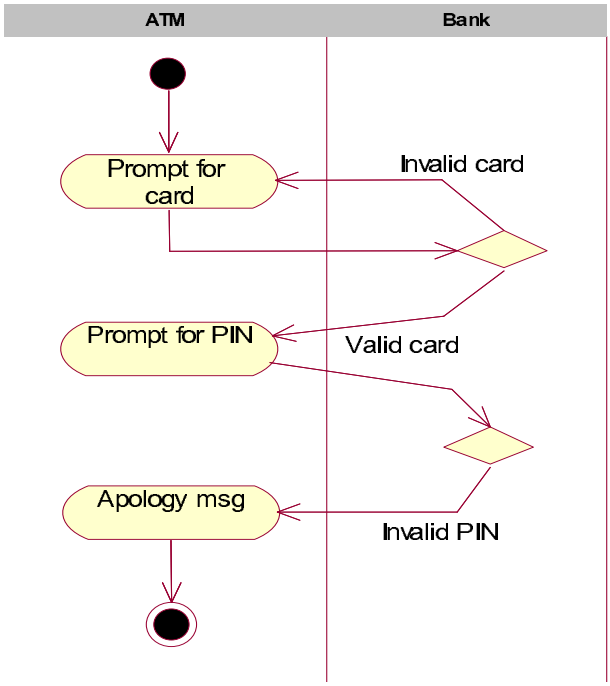

Figure 6.5: Activity Diagram for PIN Verification

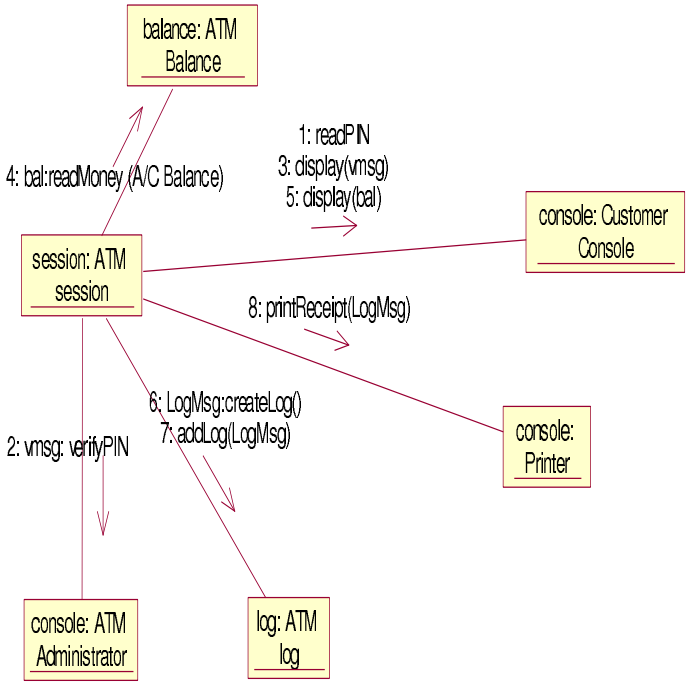

Figure 6.6: Collaboration diagram for Balance Enquiry with print receipt

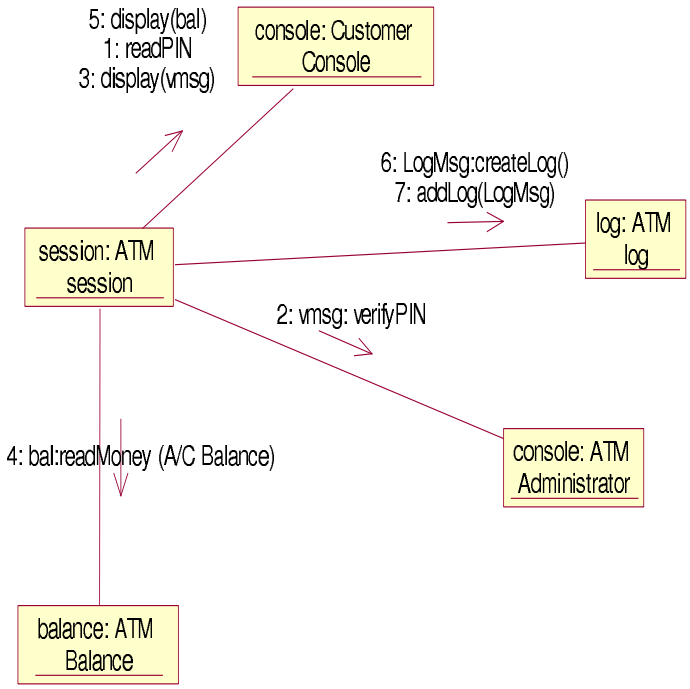

Figure 6.7: Collaboration diagram for Balance Enquiry without print receipt

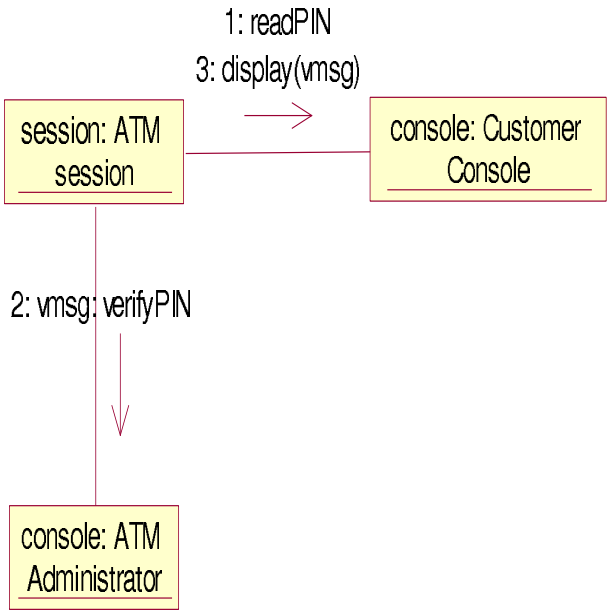

Figure 6.8: Collaboration diagram for PIN Verification

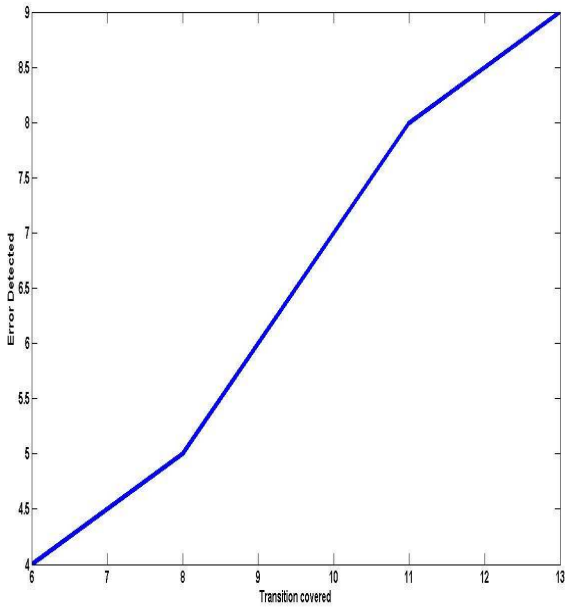

Figure 6.9: Transition Coverage Vs Error Detected

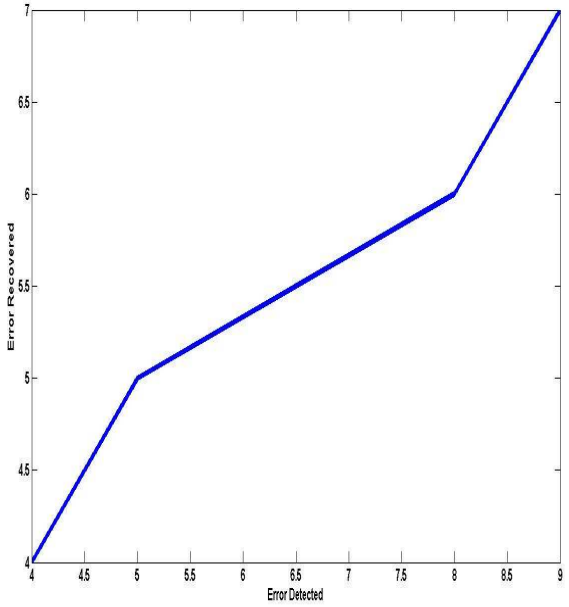

Figure 6.10: Errors Detected Vs Errors Recovered

Chapter 7
Conclusion

In this thesis, we have primarily focussed on test case generation of object-oriented software automatically. We have also explored the technique for application of evolutionary algorithm like genetic algorithm to the automatic approach of testing. Our thesis immensely encourages the approach of model based testing, which is on demand for large scale software development. The work reported in this thesis is summarized in this chapter. In Section 6.1, we summarized the chapter wise contributions of our work. Scope for future development of our work is discussed in Section 6.2.

7.1 Contributions of our work

In Chapter 4, we have proposed an approach to generate the test cases for *object oriented programs* from the UML activity diagrams. Our approach first constructs the activity diagram for the given problem and then randomly generates initial test cases, for a PUT. Then, by running the program with the generated test cases, we can get the corresponding *PET*. Next, we compare these traces with the constructed activity diagram according to the specific coverage criteria. We have used a heuristic rule to obtain the reduced test cases, which satisfy the test case adequacy criteria. We have considered only the path (simple) coverage as test adequacy criteria for automatic test case generation. Our approach achieves the maximum path coverage, which is an added advantage.

We generate test cases directly from UML behavioral diagram, where the design is reused, in Chapter 5. By using our approach defects in the design model can be detected during the analysis of the model itself. So, the defects can be removed as early

as possible, thus reducing the cost of defect removal. First we generate test scenarios from the activity diagram and then for each scenario the corresponding sequence and class diagrams are generated. After that we analyze the sequence diagram to find the interaction categories and then use the class diagrams to find the settings categories. After analyzing each category, its significant values and constraints are generated and respective test cases are derived. The major advantage of our approach is that it handles the complicity of nested fork-join pair which is more often overlooked by other approaches. It overcomes the limitations of the existing approach such as nested fork-join and loops.Test coverage criteria achieved is another advantage of our approach. However, the overall approach is not fully automated. An automated tool can be developed for the proposed approach. This approach can further be extended by generating test cases for the complete system i.e. by implementing the approach for integration testing as interactions between different components are obtained from sequence diagrams. But for the complete system the test case size may pose as a challenge to the tester, as more manual analysis is involved in this approach. So an optimal solution is required to address the problem. The ultimate goal will be to address testability, coverage criteria and automation issues, in order to fully support system testing activities.

In Chapter 6, we have proposed an approach to generate the test cases for *object oriented programs* from the UML collaboration and activity diagrams. We have used a genetic algorithm approach to obtain the sub-optimal (best fittest) test cases, which satisfy the test case adequacy criteria. Since we can not rule out the possibility of presence of error, however we can minimize this chance and our approach deals with this, in the generated test case. We have used transition coverage as test adequacy criteria, which is better than others like path coverage and branch coverage. Our approach is suitable for simple problems with less complicity, however it needs adequate improvement to emulate with real world problems involved with complicity like development of test cases involving nested fork-joins and branch nested fork-joins.

7.2 Scope and Future Work

We conclude this thesis with its scope for further extension, which are discussed below:

- We works primarily on test case generation and its automation process but recent development does not support fully automation and hence further development is essential.
- It will lead to development of automation associated technologies.
- We have discussed on optimized test case generation, which is suitable only for simple problems i.e. without fork-join and nested fork-join activities. So, the optimization problem in software testing is required more attention of researchers, in particularly for model based testing approach.

Bibliography

[1] L. Luo, "Software testing techniques technology maturation and research strategy," Institute for Software Research International, Carnegie Mellon University, Pittsburgh, PA15232, USA, Tech. Rep. 17939, 2001.

[2] E. F. Miller, "Introduction to software testing technology," *Software Testing & Validation Techniques*, pp. 4 – 16, 1981.

[3] M. Fewster and D. Graham, *Software Test Automation Effective use of test execution tools*, 2nd ed. New York: ACM Press, 1994.

[4] A. Abdurazik and J. Offutt, "Using uml collaboration diagrams for static checking and test generation," in *Proceedings of the third International Conference on the UML*. York, UK: Lecture Notes in Computer Science, Springer-Verlag GmbH, 2000, pp. 383 – 395.

[5] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from uml activity diagram based on gray-box method," in *11th Asia-Pacific Software Engineering Conference (APSEC'04)*. IEEE Computer Society, 2004, pp. 284 – 291.

[6] "Object management group," available at http://www.omg.org/uml, 2003.

[7] Specification and D. L. (SDL), available at http://www.iec.org/online/tutorials/sdl/.

[8] J. Bowen, *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, International Thomson Publishing, 1996.

[9] J. Davies and J. Woodcock, *Using Z: Specification, Refinement and Proof*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[10] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, 1993, pp. 268 – 284.

[11] S. Helke, T. Neustupny, and T. Santen, "Automating test case generation from z-specifications with isabelle," in *Proceedings of ZUM'97: The Z Formal Specification Notation*, LNCS 1212. Springer-Verlag, 1997, pp. 52 – 71.

[12] R. M. Hierons, S. Sadeghipour, and H. Singh, "Testing a system specified using statecharts and z," *Information and Software Technology*, vol. 43, no. 2, pp. 137 – 149, 2001.

[13] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell*. NY, USA: O'Reilly, 2005.

[14] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 2001.

[15] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, "A uml-based approach to system testing," *Journal of Innovations System Software Engineering*, vol. 1, pp. 12 – 24, 2005.

[16] L. Briand and Y. Labiche, "A uml-based approach to system testing," *Journal of Software and Systems Modeling*, vol. 1, pp. 10 – 42, 2002.

[17] A. Bertolino and S. Gnesi, "Use case-based testing of product lines," in *Proceedings of the ESEC/SIGSOFT FSE*, 2003, pp. 355 – 358.

[18] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," *IEE Proceedings of Software*, vol. 146, no. 4, pp. 187 – 192, August 1999.

[19] M. Riebisch, I. Philippow, and M. Gotze, "Uml-based statistical test case generation," in *Proceedings of ECOOP 2003*, S. Verlag, Ed. LNCS 2591, 2003, pp. 394 – 411.

[20] L. C. Briand, Y. Labiche, and J. Cui, "Automated support for deriving test requirements from uml statecharts," *Journal of Software and System Modeling*, vol. 4, no. 4, pp. 399 – 423, 2005.

[21] C. Nebut, F. Fleurey, Y. L. Traon, and J. Jean Marc, "Automatic test generation: A use case driven approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 3, pp. 140 – 155, 2006.

[22] F. Basanieri, A. Bertolino, and E. Marchetti, "The cow-suite approach to planning and deriving test suites in uml projects," in *Fifth International Conference on the UML*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2002, pp. 275 – 303.

[23] R. Mall, B. R. Kar, and J. Lalchandani, "Model-based testing of object-oriented software," *CSI Communications*, 2008.

[24] R. V. Binder, *Testing Object-Oriented System Models, Patterns, and Tools*. NY: Addison-Wesley, 1999.

[25] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493 – 510.

[26] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for uml design models," *Software Testing, Verification and Reliability*, vol. 13, pp. 95 – 127, 2003.

[27] C. M. Chung, "Software development techniques - combining testing and metrics," in *Proceedings of IEEE Region 10 Conference on Communication Systems*, Hong Kong, September 1990.

[28] ——, "A family of testing path selection criteria," *International Journal on Mini and Micro Computers*, August 1991.

[29] R. Mall, *Fundamentals of software engineering*, 2nd ed. New Delhi: Prentice-Hall of India Ltd, 2008.

[30] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Addison-Wesley, 2001.

[31] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[32] H. Zhong, L. Zhang, and H. Mei., "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534 – 546, May 2008.

[33] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 2000.

[34] P. Jalote, *An Integrated Approach to Software Engineering*, 3rd ed. Springer/Narosa, 2006.

[35] G. Myers, *The art of software testing*, 2nd ed. Hoboken, New Jersey: John Wiley & Son, 2004.

[36] L. Owterweil, "Strategic directions in software quality," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, December 1996.

[37] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for uml activity diagrams," in *Proceedings of the 2006 international workshop on Automation of software test*, Shanghai, China, 2006, pp. 2 – 8.

[38] P. Samuel, R. Mall, and P. Kanth, "Automatic test case generation from uml communication diagrams," *Information and Software Technology*, vol. 49, no. 2, pp. 158 – 171, 2007.

[39] D. C. Kung, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao, "Object-oriented software testing-some research and development," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*. Washington DC, USA: IEEE Computer Society, 1998, pp. 158 – 165.

[40] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. S. Kim, and Y. Song, "Developing an object-oriented software testing and maintenance environment," *Communications of the ACM*, vol. 38, no. 10, pp. 75 – 87, October 1995.

[41] A. S. Parrish, R. B. Borie, and D. W. Cordes, "Automated flow graph-based testing of object-oriented software modules," *Journal of Systems and Software*, vol. 23, no. 2, pp. 95 – 109, November 1993.

[42] P. Jorgensen and C. Erickson, "Object-oriented integration testing," *Communications of the ACM*, vol. 37, no. 9, pp. 30 – 38, September 1994.

[43] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structure," in *Proceedings of 14th International Conf. on Software Engineering*, 1992, pp. 68 – 80.

[44] C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," in *Proceedings of IEEE Conference on Software Maintenance*, 1993, pp. 302 – 310.

[45] P. Frohlich and J. Link, "Automated test cases generation from dynamic models," in *Proceedings of the European Conference on Object Oriented Programming*. Springer-Verlag, 2000, pp. 472 – 491, lNCS 1850.

[46] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *SoftwareŮPractice & Experience*, vol. 25, no. 1, pp. 97 – 108, 1995.

[47] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France, "Testing uml designs," *Information and Software Technology*, vol. 49, no. 8, pp. 892 – 912, 2007.

[48] Y. Wu, M. Chen, and J. Offut, "Uml-based integration testing for component-based software," in *Proceedings of the Second International Conference on COTS-Based Software Systems*. London, UK: Springer-Verlag, 2003, pp. 251 – 260.

[49] F. Basanieri and A. Bertolino, "A practical approach to uml-based derivation of integration tests," in *4th International Software Quality Week Europe*, November 2000.

[50] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676 – 686, June 1988.

[51] F. Fraikin and T. Leonhardt, "Seditec-testing based on sequence diagrams," in *Proceedings of the 17th IEEE international conference on Automated software engineering*, 2002.

[52] A. Rountev, S. Kagan, and J. Sawin, "Coverage criteria for testing of object interactions in sequence diagrams," in *8th International Conference(FASE 2005)*, M. Cerioli, Ed., LNCS-3442. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 289 – 304.

[53] P. Samuel and R. Mall, "A novel test case design technique using dynamic slicing of uml sequence diagrams," *e-Informatica Software Engineering*, vol. 2, no. 1, pp. 71 – 92, 2008.

[54] H. Q. Nguyen, *Testing Application on the Web:Test Planning for Internet-Based Systems*. John Wiley & Sons, 2003.

[55] A. von. Mayrhauser, R. France, M. Scheetz, and E. Dahlman, "Generating test-cases from an object-oriented model with an artificial intelligence planning system," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 26 – 36, March 2000.

[56] A. Cavarra, C. Crichton, and J. Davies, "A method for the automatic generation of test suites from object models," *Information and Software Technology*, vol. 46, no. 5, pp. 309 – 314, 2004.

[57] R. M. Patton, A. S. Wu, and G. H. Walton, "A genetic algorithm approach to focused software usage testing," School of Electrical Engineering and Computer Science, University of Central Florida, Tech. Rep., 2002.

[58] P. McMinn, "Search-based software test data generation: a survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105 – 156, 2004.

[59] R. Lefticaru and F. Ipate, "Automatic state-based test generation using genetic algorithms," in *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2008, pp. 188–195.

[60] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transaction on Software Engineering*, vol. 4, no. 3, pp. 178 – 187, 1978.

[61] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," in *ASE*, 1998, pp. 285 – 288.

[62] R. P. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263 – 282, 1999.

[63] 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*, vol. 2, no. 3, pp. 270 – 285, July 1993.

[64] D. Jeffrey and N. Gupta., "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 108 – 123, February 2007.

[65] S. Yoo and M. Harman., "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 140 – 150.

[66] K. H. Chang, W. H. Carlisle, J. H. C. II, and D. B. Brown, "A heuristic approach for test case generation," in *Proceedings of the 19th annual conference on Computer Science*. San Antonio, Texas, United States: ACM New York, New York, USA, 1991, pp. 174 – 180.

[67] R. E. Prather and J. P. M. Jr., "The path prefix software testing strategy," *IEEE Transactions on Software Engineering*, vol. 13, no. 7, pp. 761–766, July 1987.

[68] W. H. Deason, D. B. Brown, K. H. Chang, and J. H. C. II, "A rule-based software test data generator," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 1, pp. 108 – 117, March 1991.

[69] H. Zhu and X. He, "A methodology of testing high-level petri nets," *Information and Software Technology*, vol. 44, no. 8, pp. 473 – 489, 2002.

[70] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture Ű practice and promise*. Addison-Wesley, 2003.

[71] J. Edvardsson, "A survey on automatic test data generation," in *Proc. of the Second Conference on Computer Science and Engineering in Linkoping(ECSEL-99)*, October 1999.

BIO-DATA

Baikuntha Narayan Biswal

Date of Birth: 13th June 1984

Correspondence:

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

Ph: +91 94385 72927

E-mail: baikunthanarayan@gmail.com, baikuntha@nitrkl.ac.in

Qualification

- M.Tech. (CSE)
  NIT Rourkela, [Continuing]
- M. Sc (Computer Science)
  Ravenshaw University, Cuttack, Odisha [First division]
- B.Sc (Physics)
  Utkal University, Bhubaneswar, Odisha [Second division]
- +2
  CHSE, Bhubaneswar, Odisha [First division]
- 10th
  BSE, Cuttack, Odisha, [First division]

Professional Experience

Project Fellow, NIT Rourkela, October 2007 – 31st April 2010

Publications

- 02 Journal Articles
- 04 Conference Articles

Permanent Address

At/Po: Bandhapatna, Via: Madanpur
Dist: Kendrapara, Odisha-754246.