

Test Case Generation using UML Activity Diagram & Composite Structure Diagram

Kishan Bhukya

(Roll No: 213CS3180)



Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, Odisha, India

May 2015

Test Case Generation using UML Activity Diagram & Composite Structure Diagram

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

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Kishan Bhukya

(Roll No.- 213CS3180)

under the supervision of

Prof. D.P. Mohapatra



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

May 2015



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

Certificate

This is to certify that the work in the thesis entitled *Test Case Generation using UML Activity Diagram & Composite Structure Diagram* by *Kishan Bhukya* 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 with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: May 23, 2015

(Prof. D.P. Mohapatra)
Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Prof. Durga Prasad Mohapatra for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction to the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.

I am very much indebted to Prof. Santanu Ku. Rath, Head-CSE, for his continuous encouragement and support. He is always ready to help with a smile. I am also thankful to all the professors at the department for their support.

I would like to thank all my friends and lab-mates for their encouragement and understanding. Their help can never be penned with words.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Kishan Bhukya

Roll-213CS3180

Abstract

In software development, Quality is an important thing. We get the best quality of software when we test it properly. In present day the size and complexity in object oriented software are increased , Due to this manual testing become very resource consuming and not able to provide proper results. To handle this we need automatic test case generation which helps in finding the errors and bugs. Test cases we generate in design phase only, which is very early in software development process. We use unified modelling language (UML) to visualise the designs and structure of object-oriented software. From UML diagram, we generate efficient test cases which gives proper results during testing.

We first focus on the generation of test cases from the Activity diagram. Activity Diagrams are used to describe the behaviour of the models i.e. dynamic aspects of model. In activity diagram we describe the flow of activity from one to another. Every activity represents a different operation. We first use Rational Software Architect (RSA) to develop the Activity Diagram. From here we generate XMI (XML metadata interchange). We generate the intermediate graph by parsing the XMI code automatically. After developing the intermediate graph , by using it, in our proposed algorithm topdown test case generation algorithm we generate test cases. The generated test cases are used in testing to find out the errors, by removing erros we increase the quality of the software.

In our next work, we focus on the UML composite structure diagram (CSD). CSD defines the interaction between components because of this we use these generated test scenarios in the integration testing. We first use Rational Software Architect (RSA) to develop the CSD. From here we generate XMI (XML metadata interchange). We generate the intermediate graph by parsing the XMI code automatically. By using intermediate graph, in our algorithm we generate test scenarios. The generated test scenarios are used in integration testing to find out the errors.

Contents

Certificate	ii
Acknowledgement	iii
Abstract	iv
List of Figures	vii
1 Introduction	1
1.1 Overview of UML diagrams	2
1.2 Motivation	3
1.3 Problem Statement and Objectives	3
1.4 Thesis Organization	4
List of Tables	1
2 Basic Definitions and Concepts	5
2.1 Test Case	5
2.2 Test Scenario	5
2.3 Testing Techniques	6
2.3.1 Black Box Testing Technique	6
2.3.2 White Box Testing Technique	6
2.3.3 Grey Box Testing Technique	6
2.4 Model Based Testing	6
2.4.1 Benefits of Model Based Testing	8
2.5 Coverage Criteria	8
2.6 Integration Testing	9
2.7 Equivalence Class Partitioning	10
2.7.1 Boundary Value Analysis	11

2.8	XML Metadata Interchange (XMI)	11
3	Review of Related Work	12
3.1	Test Case Generation for Integrating testing Using UML diagrams .	12
4	Test Case generation from Activity Diagrams	16
4.1	Basic Concepts and Definitions	16
4.2	Proposed Approach	20
4.3	Case Study	21
4.3.1	Working of proposed model	22
5	Test Scenario Generation from UML Composite Structure Diagram	27
5.0.2	Black Box Testing Technique	28
5.0.3	White Box Testing Technique	28
5.1	Basic Concepts and Definitions	28
5.1.1	Composite structure diagram	28
5.2	Proposed Approach	30
5.2.1	Proposed Algorithm	31
5.2.2	Description of Algorithm	31
5.2.3	Case Study	32
5.2.4	Working of Algorithm	33
6	Conclusion and Future Work	37
6.1	Test Scenario Generation from UML Activity Diagram	37
6.2	Test Case generation from Composite Structure diagrams	37
	Bibliography	39

List of Figures

2.1	Model Based Testing Process	7
4.1	Intial node	16
4.2	Final node	17
4.3	Activity	17
4.4	Flow	17
4.5	Fork and Join	18
4.6	Decision	19
4.7	Merge	19
4.8	Proposed framework for test cases and test sequences generation . .	20
4.9	Activity diagraph for withdarw money from ATM	22
4.10	Activity diagraph for withdarw money from ATM	23
4.11	Java parser	24
4.12	interediate graph	25
4.13	test cases	26
5.1	Basic Symbols of Composite Structure Diagram	29
5.2	Block diagram of proposed approach	30
5.3	online shopping composite structure diagram	33
5.4	XMI	34
5.5	java parser	34
5.6	intermediate graph	35
5.7	generated test scenarios	36

Chapter 1

Introduction

In software testing we generally prefer that we execute the software on a certain conditions and we then compare the expected output and the output which we actually needed i.e. actual output [1]. In testing we try to find out the errors, bugs etc. to free the system from the failure and to increase the quality of the software. We do the testing in three stages: generating of test cases, executing the these generated testcases and test results evolution [2]. In these three stages the first one i.e. generating the test cases is the difficult task compared to the other two tasks. In present day the size and complexity in object oriented software increased, due to this manual testing become very resource consuming and not able to provide proper results. To handle this we need automatic test cases generations which helps in finding the errors and bugs. Test cases we generate in design phase only, which is very early in software development process which helps in reducing the development cost of the software. Additionally measures the product quality as far as its ability for dependability, accuracy, practicality, testability, ease of use and re-convenience. A percentage of the goals of testing are as per the following:

- A quality test case ought to have high likelihood of discovering a error
- Meets the necessities that guided its design and advancement
- It guarantees quality of the software
- testing of software minimizes the failure of the product

1.1 Overview of UML diagrams

Unified modelling language(UML) is explains the graphical structure for system for imagining, identifying, developing and documenting of the artifacts of product. UML is a blueprint of the product which we want to develop and it helps in the making of the documentation of the product [3]. By using Unified modelling language we can make any system easily understandable to various developers who are participating in the developing that product and working on the various types of platforms. The main advantage of the unified modelling language is that in not depends on system or platform. Modeling is an essential piece of the software product, which also helps in the development medium and small products. In UML 2.0 we in total have 14 diagrams which are used to model different artifacts of the project. The use of these diagrams are increasing day by day which encourages us in using these diagrams to develop test cases for the product.

we mainly have three types of UML diagrams

1. **Structure diagram:** The structure diagram maily deals with the static parts of the software. These diagrams basically highlights the important things which are present in product being modelled. Various types of structural diagram are Object diagram, Component diagram, Class diagram, Package diagram, Composite structure diagram and Deployment diagram.
2. **Behavioral diagram:** These behavioral diagrams mainly deals with how the system is interacting i.e. how the interaction will occurs in the system. These diagrams deals with the dynamic behaviour of the system. Various types of the Behavioral diagram are Activity diagram, Use case diagram, State machine diagram.
3. **Interaction diagrams :** These diagrams deals with the how the data will flow and control handling among the various parts of the system which we are going to modelled. Various types of the interaction diagrams are Interaction overview diagram, Sequence diagram, Communication diagram, Timing diagram.

1.2 Motivation

Present day situations and requirements are demanding the project to be developed in object oriented method. In actual these object oriented methods are huge in nature and compare to other methods it is some what complex because of its characters like polymorphism and inheritance. Because of there characteristics object oriented methods , generating the test cases automatically from object oriented methods is very difficult. To do this task we use the unified modelling language which is easy to generate test cases from object oriented methods. Now a days we have different type of unified modelling diagrams which are acting as main source of developing the required test cases of the product. By using the unified modelling language we can find out the errors in the design phase only rather than finding after developing phase, which will reduce the development cost and it helps in reducing the wastage of resources. In our paper, we are going to generate test cases from activity diagram and composite structure diagram which helps in integration testing.

1.3 Problem Statement and Objectives

In software development most important and difficult task is testing. The most expensive part of the software life cycle is testing. In starting days testing is conducted after completion of the code for the product. After performing the test on that code and if they found the errors they will go back to design phase and starts preparing of another documentation which will lead to wastage of the valuable resources and money. But in model based testing, using unified modelling language we can develop the test cases at desing phase which is the second step in SDLC. Unified modelling language is very highly used by different persons to show the models of the system. Form these models we can develop test cases easily. Objectives,

- To propose a methodology to automatically generate the test scenarios using the UML Activity diagram

- To propose an methodology to automatically generate test cases using composite structure diagram.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter-2, In this chapter we discussed basic terms which we are used in the paper. The terms are like test case, test scenarios, testing etc In this chapter we discussed basic terms which we are used in the paper. The terms are like test case, test scenarios, testing etc

Chapter-3, In this chapter we discussed about some work which is used in our contribution. We discussed about integration testing , designing methods of test cases.

Chapter-4, In this we discuss how to design the the test cases from Activity Diagram. In starting we explain basic conept and used terms. Secondly we explain how to develop test cases with example

Chapter-5, In this we discuss how to design the the test cases from Composite Structure Diagram. In starting we explain basic conept and used terms. Secondly we explain how to develop test cases with example.

Chapter-6, concludes the thesis with a summary of our contributions. We also briefly discuss the possible future extensions to our work.

Chapter 2

Basic Definitions and Concepts

To understand thesis we are discussing some terms which we used throughout in the paper and discussed the very basic concepts which are used in the paper. By explaining like this it is easy to understand the paper without much effort.

2.1 Test Case

Generally we develop the required test cases based on the specifications of the product and the requirements of the product. These both specifications , requirements are stored in the SRS document on which we develop the required test cases. Test case is combination of three units. First one is input, second one is system conditions i.e. state to which we are taking the input and finally third one is, after performing the required calculations we get an output from the system [4] i.e. expected output. We can form the test suite by combining the test cases [5].

2.2 Test Scenario

Test scenarios are nothing but combination of the test cases, which executes in a specific order to give some required result [6]. So, test scenarios are combination of test cases which will test the flow of data from one end of the product to the other end. Test scenarios are nothing but set of test cases which arranged in proper order to execute , to check the flow of the system.

2.3 Testing Techniques

Testing techniques are mainly divided into three categories: We Generally found the three techniques for testing. Those are Black box testing White box testing Graybox testing

2.3.1 Black Box Testing Technique

In this type of testing as name suggests we cant consider the internal structure of the product. In this testing we test the product by giving the required input and we wait for the output and we compare the actual output and real output. During the output generation we not consider the how the output is generating. We normally call this test as functional testing

2.3.2 White Box Testing Technique

Compare to the black box testing the white box testing talks about the internal structure of the product. So, in this testing we discuss about input , output and the most important the internal coding structure of the product i.e. we discuss how we are getting the output. We also known this testing as glass or open box or structural testing.

2.3.3 Grey Box Testing Technique

This testing is combination of the black box and white box testing techniques. In this type of method we generally observe the important the internal coding structure of the product i.e. we discuss how we are getting the output. As said it is a combo of both techniques we also observe the outer structure of the system and we discuss what are the inputs and what outputs we expecting and we compare the outputs we obtained.

2.4 Model Based Testing

Model based testing, as name suggests we mainly depends on the models to test the product we designed. In this test cases we develop the test cases using models

of the system. Model is nothing but which explains some parts of the product. compare to total system , models are easier to understand the product and very easy to develop the test cases which are used in the testing. By using These models we can understood the actually what the product is. In this testig we have three main stages:

1. Designing the model for system requirements which are used in testing.
2. Developing the test cases from the designed model considering the requirements.
3. Developing the test cases from the designed model considering the requirements.

In companies deployment of model based testing conducted mainly in following steps. 2.1 The model is generally created from the requirement specification

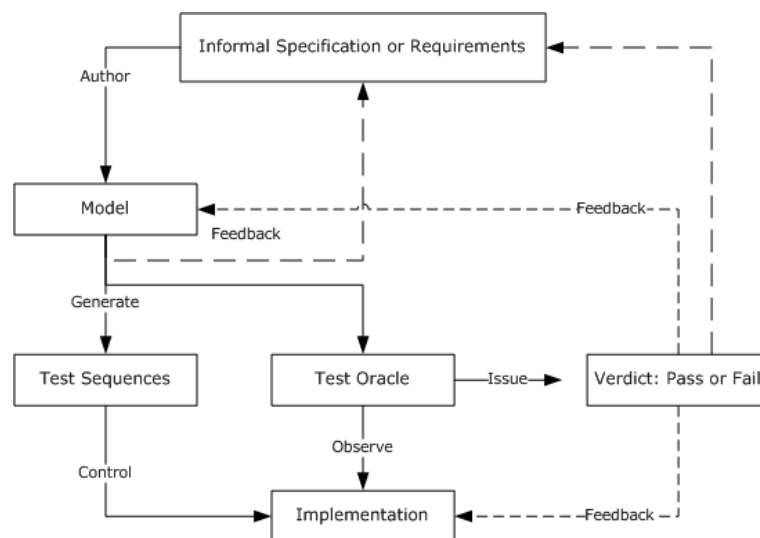


Figure 2.1: Model Based Testing Process

document. The model is then used to generate the test suites. These test suites contain both the test oracle and test sequence. Test sequence is used to control the system under test. Test Oracle is used for determining whether a test has failed or passed. A failure indicates that the system does not perform according to user requirement. The required model is developed on the basis of the specifications

and requirements of the system i.e. SRS documents. From these developed models we develop the test suites which contains test cases which are used for testing. By using these test suites we generally check how the flow is going in the system and the result of the product i.e. whether the product is working properly or not.

2.4.1 Benefits of Model Based Testing

1. The main benefit of the MBT is it is easy to understand for clients and the software developers groups.
2. Model-based testing separates business process and testing of the code .
3. By using MBT we can use automated testing method very quickly.
4. We mainly concentrate on the coverage of requirement in MBT.
5. In MBT we have design work is more compared to code .

2.5 Coverage Criteria

Coverage code is nothing but how effectively the program source code is tested by the particular test suit. We do it because if the coverage of code is very high then there is a chance of reducing the bugs in the product, by that we get quality product.

Some of the coverage criteria are.

1. **Statement coverage:** In code we encounter number of statements. This coverage will explain about the all present statements in the code must execute minimum one time
2. **Branch Coverage:** This coverage will explain about the all present decision in the code must execute minimum of one time with all outcomes.
3. **Condition Coverage:** This coverage will explain about the all present condition in the code must execute minimum one time with all outcomes.

4. **Decision/Condition Coverage:** This coverage will explain about the all present decision in the code must execute minimum of one time with all outcomes. All present condition in the each decision must execute minimum one time with all outcomes.
5. **Path Coverage:** This coverage will explain about the all present paths in the code must execute minimum of one time.

2.6 Integration Testing

As name suggests we combine all components and perform the test on the newly developed component [7]. This testing will start when every component finishes its own testing i.e. individual. i.e. nothing but after the unit testing is terminated. We do integrating the components and performing the test because when single component is tested separately it may perform the correct behaviour but when we integrate that component with the other component then there is no guarantee that the component once performed correctly will perform once again correctly. So we do this testing because we check the components working properly or not even after the combining those components with other components [8]. In this testing we encounter with stub and driver. Some times some of the components are not built properly at that time for integration, so at that time use stub and driver concept

Stubs: To check the behaviour of the lower level components we add the stub i.e. we use the stub .

Driver: To check the behaviour of the upper level components we add the driver i.e. we use the driver.

We generally have following type of integration techniques for testing:

1. **Top-Down:** In this testing techniques we first perform the testing on the higher stage components by integrating them. After completing higher stage components we then go for lower stage components one by one. In this technique we create stub to test the lower stage components [9].

2. **Bottom-Up:** In this testing techniques we first perform the testing on the lower stage components by integrating them. After completing lower stage components we then go for higher stage components one by one. In this technique we create driver to test the higher stage components.
3. **Sandwich:** Sandwich integration testing is the combo of Top Down and Bottom Up integration testing.
4. **Big Bang:** In this testing we integrate all components or part of components and we perform the testing on the integrated new component.

In all these four techniques we mostly use the top down or bottom up techniques in today's companies. The big bang technique is we can integrate components in less time and from integrating these components its cost is also low but disadvantage of this technique is we can not find out in which component the error exists. After completing of this integration testing next stage is system testing. In this paper we worked with top down and bottom up approach.

2.7 Equivalence Class Partitioning

In the approach of equivalence class partitioning, we divide the inputs into sets of equivalence classes, which we are going to use those inputs in the system which is under test. After dividing the equivalence classes, we generally form one test case for the one equivalence class. We develop these equivalence classes of input based on the system requirements for the data which we are giving as input. By using this technique we diminish the time required for testing the product with help of effective test cases.

This testing is very useful because we can use this method at any stage of the testing and it is generally a good technique to utilize first. For each class partition we try a one condition because we agree that the conditions present in one partition will behave as same. We know that in one partition if one condition performs well, then the other condition also performs well. Like this, if one condition not

performs well, then the other remaining conditions in that partitions also not perform as expected.

- Suppose you have a programme which takes input between 100-200, so the valid partition will be (100-200), equivalence partitions will be like:

Invalid partition below to 99, valid partition 100-200, invalid partition above 200 201 and above

2.7.1 Boundary Value Analysis

In this we develop the test cases depending on the values present at end of the equivalence classes. One good test case requires that test cases should developed in such way that it will find the more number of errors. 2.7.

- **Test Cases 1:** Designing the test cases at the extremes of given domain i.e. 200 , 300.
- **Test Cases 2:** Designing the test cases below the given domain i.e. 199 and 299.
- **Test Cases 3:** Designing the test cases above the given domain i.e. 201 and 301.

2.8 XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) , an interchange structure for the metadata which is discussed as Meta-Object Facility (MOF) standard [10]. XMI plays a very important role in IBM Rational Software Architecture to design UML models in XMI metadata structure [11].

Chapter 3

Review of Related Work

In this chapter we discuss about previous existed related work done by different research people in the area of generating test cases from using UML diagrams for integration testing and next we given an overview about the present related work which we did in the generating the test scenarios using UML diagrams. We worked on the developing of test cases from the activity diagram and developing test scenarios from the composite structure diagram.

3.1 Test Case Generation for Integration testing Using UML diagrams

Number of researchers are working on the designing the test cases using various UML diagrams. Here we discuss the existing work which is related to our work. Sarma et al [12]. have proposed a model how to design the test cases using UML sequence diagram. In his method first he designed the sequence diagram , from it he developed the sequence intermediate graph which is used for developing the test cases. These test cases which he designed are used to test the system to find out the errors and to check the correctness of the system.

Traon et al [7]. given a procedure to use the object oriented model for regression testing and also for integration testing. By using the refinement process in object oriented model, he developed a model for structural system dependency graph which helps in the ordering of methods and classes to test for the purpose of the integration testing and regression testing. He reduced stubs count which are used

in the testing process.

Hartmann et al [13]. given an methodology for designing the components and how those components will interact with each other. They took the help of the statecharts to desing the dynamic character of the components and they also used the these statecharts to explain how the components will interact with each other. By using the messages, the interaction occurs between the components where the messages containing no parameters and required values. In our methodology there is no constraint on the messages ,they can use parameters and required values.

Hanh et al. [8] given two testing techniques for integration testing. The first method is depends on the deterministic procedure and the second method depends on the particular algorithms i.e. genetic algorithm. They first developed the class diagram and also developed the package diagram to find out the what dependency exist between the different components. The data received till now is used to develop a test intermediate graph. Their main motto is to reduce the count of the stub. Compare to this approach in our approach , finding the error contained component is more.

Wu et al. [14] given an approach for testing the product which is dependent on the component based with help of the UML diagrams. They took the help of the statecharts to desing the dynamic character of the components and they also used the these statecharts to explain how the components will interact with each other.

Wang et al. [15] proposed a methodology to generate the test case from a class diagram and an interaction diagram. The test adequacy criteria they used is the coverage of the design model elements, also called the building blocks in the class diagrams and the interaction diagrams. They have adopted the category partition approach to get the function units, then for each function unit, generate test cases from class diagram criteria. The method sequence from the interaction diagram is used to generate sequence of the signals in the test case. The generated test cases are able to meet all message path criteria.

A methodology is proposed by Swain et al. [16] to prioritize test scenario from UML communication and activity diagrams. They presented an integrated ap-

proach and a prioritization technique to generate cluster-level test scenarios from UML communication and activity diagrams. First, they convert the communication and activity diagrams into a tree representation respectively. Then combines the tree representation of diagrams into intermediate tree named as COMMACT tree. The COMMACT tree is then traversed to generate the test scenarios. They have proposed a prioritization metric considering the coupling or impact or influence of activity and methods. They considered the criticality of guard conditions to perform those activities and methods. Their approach generates prioritized test scenarios and test scenarios are not redundant.

Pilskalns et al. [17] presented a graph based approach to combine the information from sequence diagrams and class diagrams. In this approach, first sequence diagram is transformed into an object-method directed acyclic graph (OMDAG). The values of variable in class diagram are then associated with objects in OMDAG during path traversal. The execution sequence and attribute value of generated test cases is stored into an object method execution table (OMET). This approach achieves the All message paths and Attribute criteria.

Boghdady et al. [18] given an approach to develop test cases using the UML activity diagram. They given an algorithm which will generates the activity dependency table automatically. The generated table is then used to develop the intermediate graph which is activity dependency graph. With help of the generated table and the generated graph they designed the test cases using the UML activity diagram. The final designed test cases are used in testing the product.

Kansomkeat et al. [19] have given an approach which is know as Condition-Classification Tree Method to desingn the test cases using UML Activity diagrams. They first developed the condition classification trees from activity diagram and from there the developed the test case table. Finally with help of these test case tables they designed the test cases. They first developed the input output explicit activity diagram with help of a basic activity diagram and they generated intermediate graph, with using this graph they produced the test cases.

Kundu et al. [20] designed the test cases using the UML activity diagrams in

UML 2.0 notations. Their first step includes augmenting the developing activity diagram with required test data. Secondly they are designing the an intermediate graph from the augmented activity diagram later which they using that graph in designing the test cases. The developed test cases are used in testing the product and there particularly very helpful in the finding faults in the area of synchronization and loop faults

Kim et al. [21] proposed a method in which they first developed the input output explicit activity diagram with help of a basic activity diagram and they generated intermediate graph, with using this graph they produced the test cases. The conditions for conversation are depends on the single stimulus principle , we resolves the state explosion problem. They mainly concentrated on the behaviour relevant data and designed the model accordingly. They used all-paths test coverage technique.

Chapter 4

Test Case generation from Activity Diagrams

Generally we use activity diagrams to show how the different activities are executing in a sequence flow. These diagrams displays the path from initial point to last point which includes number of decisions, which will handled the different situations of the system. These diagrams are also used to represent the parallel execution of the activities. We mainly use these diagrams in the modelling of the business related works

4.1 Basic Concepts and Definitions

1. **Initial node:** The initial node is nothing but starting node in the diagram. We represent this initial node with filled in circle. Actually we dont need this starting node but for our convenience to understand the diagram or any logic which we represented using the activity diagram are easy to understand

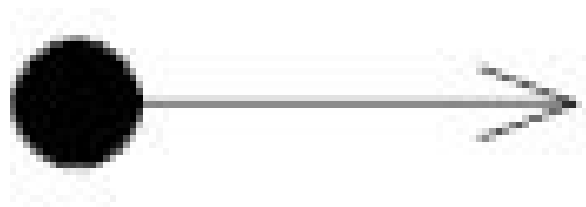


Figure 4.1: Intial node

2. **Activity final node:** In Activity diagram we represent final node to indicate the diagram is completed there. We represent the final activity node

by the filled circle with a circle border is the symbol. In activity diagram we may have zero final nodes or we may have more than one or more than final nodes in a single diagram



Figure 4.2: Final node

- 3. Activity:** The rounded rectangles represent activities that occur. An activity may be physical, such as Inspect Forms, or electronic, such as Display Create Student Screen. In activity diagram, we use the activities to display the activities that are happening or to show the actions which we are using in the diagram. We represent the activity in the diagram by rounded rectangle.



Figure 4.3: Activity

- 4. Flow:** We see lots of arrows in the diagram. Those arrows shows the flow the process or control in the sytem. Flow designing is very important because if we not provide proper flow , it may ruin the whole sytem resulting in the lot of wastage of resources.



Figure 4.4: Flow

5. **Fork and Join:** We use this option in the diagram to show the parallel activity is happening in the process. We represent this fork with a black bar which containing one in flow and number of out flows. Fork is widely used because generally in a system there happens lot of parallel activities. To understand them easily we are using the fork. Join completely opposite to the fork which we discussed in the above section. If we are using the join in the diagrams it says that the parallel activities are completing now. We represent join in the diagram as a black bar in which there are number of inflows and only one single outflow.

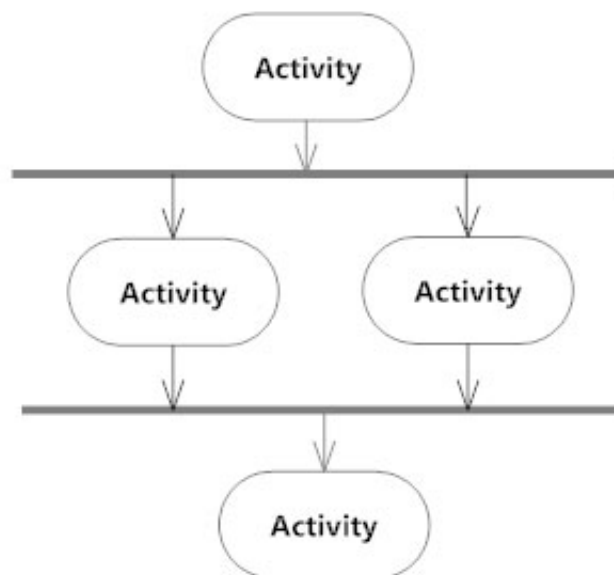


Figure 4.5: Fork and Join

6. **Decision:** Decision which we use frequently in the activity diagram to take the decision. When we use the decision, if the condition is true then the control takes one path and if it won't satisfy then the control takes the different path other than previous. We represent the decision with a diamond shape which contains one incoming flow and number of leaving flows.

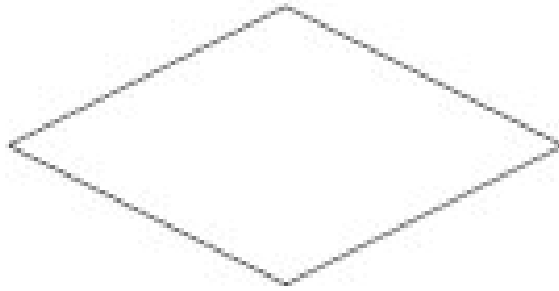


Figure 4.6: Decision

7. **Merge:** Merging is just opposite to the decision. In this merge option we observe that there are several incoming flows from other sources and only one flow is leaving the merge option. We represent the merge option with a diamond which includes the several incoming flows from other sources and only one flow is leaving.

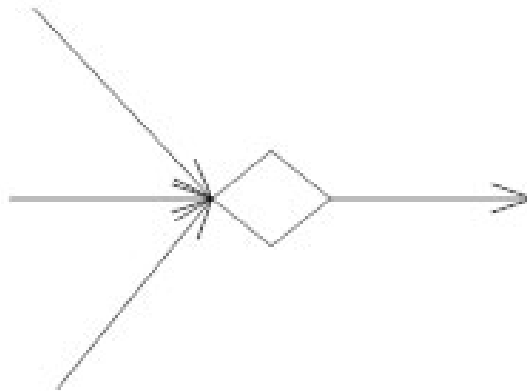


Figure 4.7: Merge

8. **Partition:** Some times we use these partitions to represent who actually participating in the execution of the activities. By using the partitions we easily understood the which activities are coming under whom and who executing those activities. This is very useful to understand the process of the system. We also call these partitions as swimlanes

9. **Flow final:** The circle with the X through it. This indicates that the process stops at this point. Flow final is used to say that the process is stopped at this time. We represent the flow final using a circle which contain the X

4.2 Proposed Approach

Here, we describe our approach which we proposed to develop test scenario using UML composite structure diagram. The process we used shown in a diagram below:

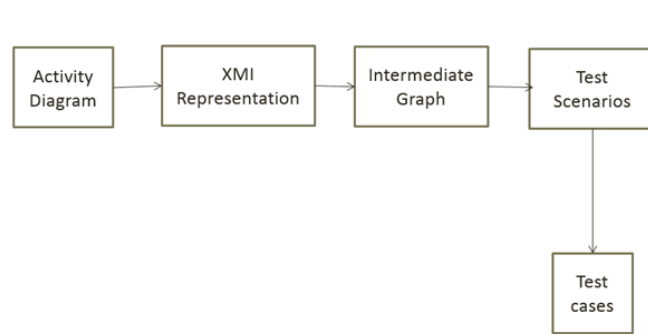


Figure 4.8: Proposed framework for test cases and test sequences generation

By using IBM Rational software Architect(RSA) we develop the activity diagram. In RSA we have a option to export the XMI of UML diagram. So we here extracted the XMI of the developed activity diagram using that option from in RSA. The developed java code the extractor will extract the required data from the exported XMI of activity diagram like name of the activities, linkage between the different activities. Using this extracted data by extractor a graph is generated which is intermediate graph. Using Gvedit tool we can see the how the graph is and how the activities are represented as nodes. Each node present in the intermediate graph is one activity in the diagram. The linkages we used in the diagram are shown as edges in the intermediate graph which is generated through extractor. In one pair of nodes we find two types of nodes. One node is represented as callee node and the second node is known as called node. Callee node are the activities which obtain the services from other nodes whereas called node are the

activities which give services to other activities. The starting node which we call as root node, wont give any service to other nodes and not require any services to other nodes. Leaf nodes in the intermediate graph shows the activities in diagram which are not requesting for any services. We made changes in the breadth first search to develop the test cases from the intermediate graph.

4.3 Case Study

We considered the withdraw of money from the ATM as the example to implement the our proposed approach. In starting we developed the an activity diagram for the withdraw of money from the ATM in the RSA. In this example of the withdraw of money from the ATM we considered the 8 activities such as entercard, enterpin, pinvalidation, showmenu, enteramount, amountverification, despense-cash, takemoney. Along with we used starting point and end point also. In this diagram we used the three partitions i.e. swimlanes like first one user, second one atm, third one is bank. After developing this diagram in the RSA we export the XMI of the activity diagram which is used as the input in the extractor. The extractor i.e. java code will take the xmi as the input and develops the intermediate graph which show the different activities present in the activity diagram as nodes and the flow between those activites as the edges between these new generated nodes. By applying the proposed algorithm we generate the test cases. The procedure is shown below step by step with diagram.

4.3.1 Working of proposed model

Step1: Developing the activity diagram in RSA

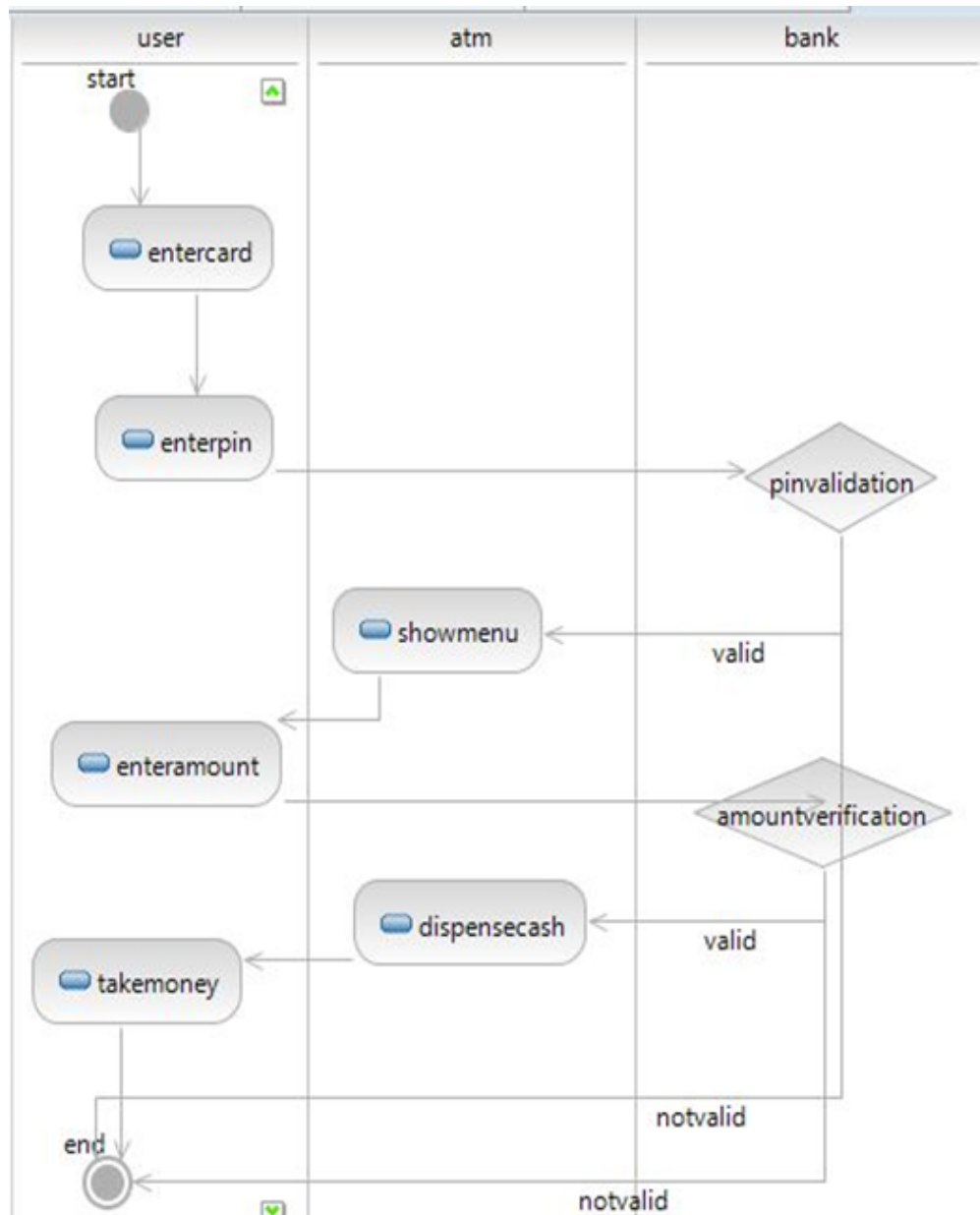


Figure 4.9: Activity diagram for withdraw money from ATM

step2: extracting XMI

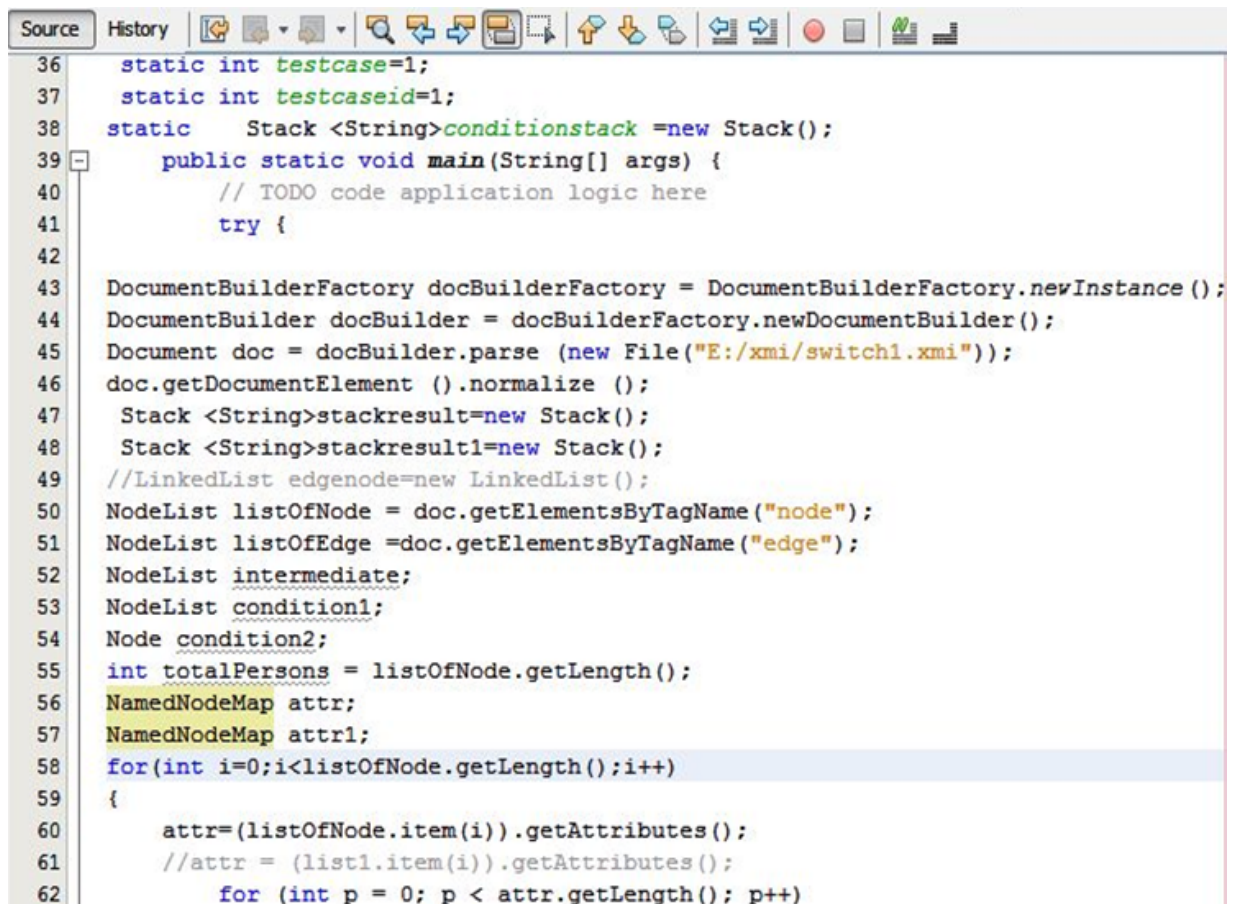
```

<uml:Model xmi:version="2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns:uml="http://schema.omg.org/spec/UML/2.1.1"
xsi:schemaLocation="http://schema.omg.org/spec/UML/2.1.1
http://www.eclipse.org/uml2/2.0.0/UML"
xmi:id="_ZiNKEHYfEeSm5tOeKeyZ_g" name="Blank Model">
  <packageImport xmi:type="uml:PackageImport"
xmi:id="_ZiNKEXYfEeSm5tOeKeyZ_g">
    <importedPackage xmi:type="uml:Model"
href="http://schema.omg.org/spec/UML/2.1.1/uml.xml#_0"/>
  </packageImport>
  <packagedElement xmi:type="uml:Activity"
xmi:id="_ZiNKEnYfEeSm5tOeKeyZ_g" name="Activity2">
    <node xmi:type="uml:InitialNode"
xmi:id="_ZiNKE3YfEeSm5tOeKeyZ_g" name="start"
outgoing="_ZiNKHXYfEeSm5tOeKeyZ_g"/>
    <node xmi:type="uml:OpaqueAction"
xmi:id="_ZiNKFHfEeSm5tOeKeyZ_g" name="entercard"
outgoing="_ZiNKIHfEeSm5tOeKeyZ_g"
incoming="_ZiNKHXYfEeSm5tOeKeyZ_g"
inPartition="_ZiNKPnYfEeSm5tOeKeyZ_g"/>
    <node xmi:type="uml:OpaqueAction"
xmi:id="_ZiNKFXfEeSm5tOeKeyZ_g" name="enterpin"
outgoing="_ZiNKI3YfEeSm5tOeKeyZ_g"
incoming="_ZiNKIHfEeSm5tOeKeyZ_g"
inPartition="_ZiNKPnYfEeSm5tOeKeyZ_g"/>
    <node xmi:type="uml:DecisionNode"

```

Figure 4.10: Activity diagram for withdraw money from ATM

step3: giving this XMI as input to the Java Parser



```
Source History
36 static int testcase=1;
37 static int testcaseid=1;
38 static Stack <String>conditionstack =new Stack();
39 public static void main(String[] args) {
40     // TODO code application logic here
41     try {
42
43     DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
44     DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
45     Document doc = docBuilder.parse (new File("E:/xmi/switch1.xmi"));
46     doc.getDocumentElement ().normalize ();
47     Stack <String>stackresult=new Stack();
48     Stack <String>stackresult1=new Stack();
49     //LinkedList edgenode=new LinkedList();
50     NodeList listOfNode = doc.getElementsByTagName ("node");
51     NodeList listOfEdge =doc.getElementsByTagName ("edge");
52     NodeList intermediate;
53     NodeList condition1;
54     Node condition2;
55     int totalPersons = listOfNode.getLength();
56     NamedNodeMap attr;
57     NamedNodeMap attr1;
58     for(int i=0;i<listOfNode.getLength();i++)
59     {
60         attr=(listOfNode.item(i)).getAttributes();
61         //attr = (list1.item(i)).getAttributes();
62         for (int p = 0; p < attr.getLength(); p++)
```

Figure 4.11: Java parser

Step4: generatign the intermediate graph

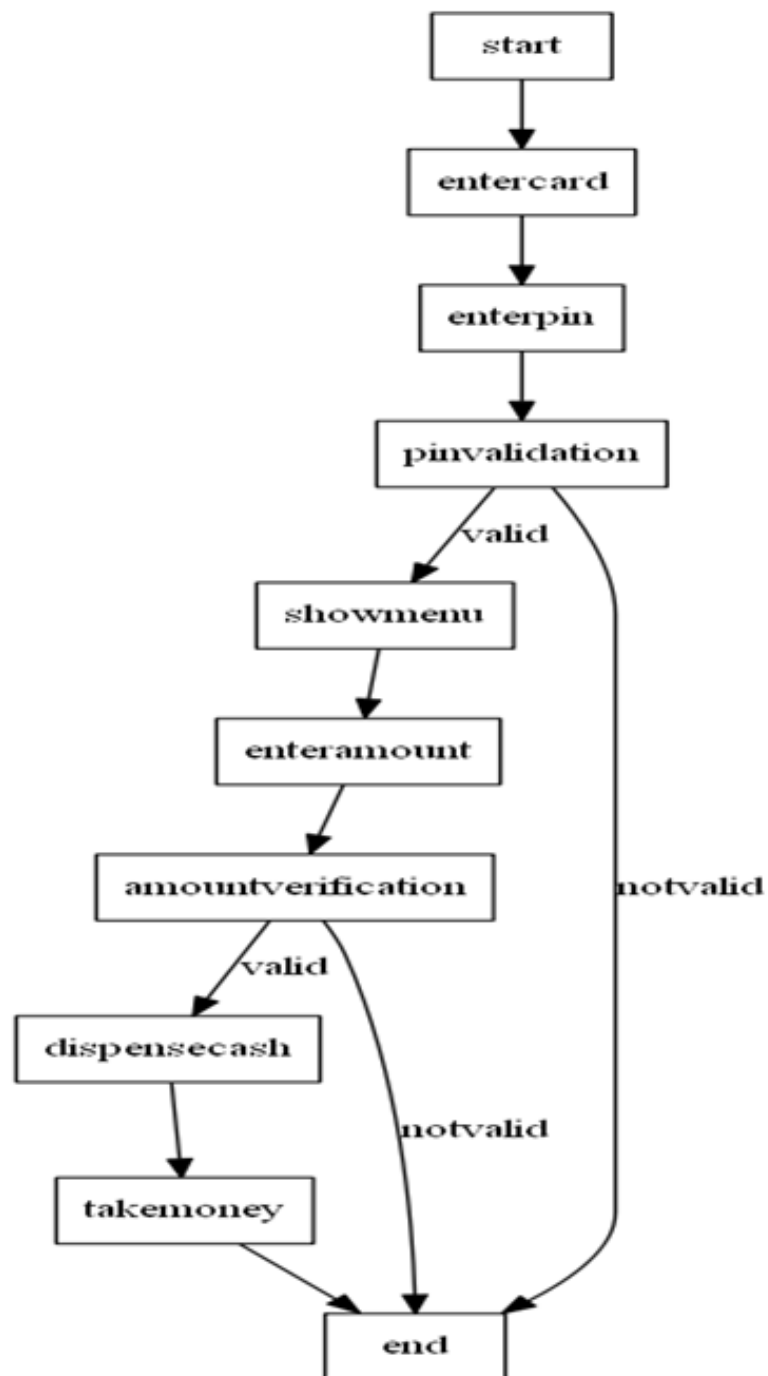
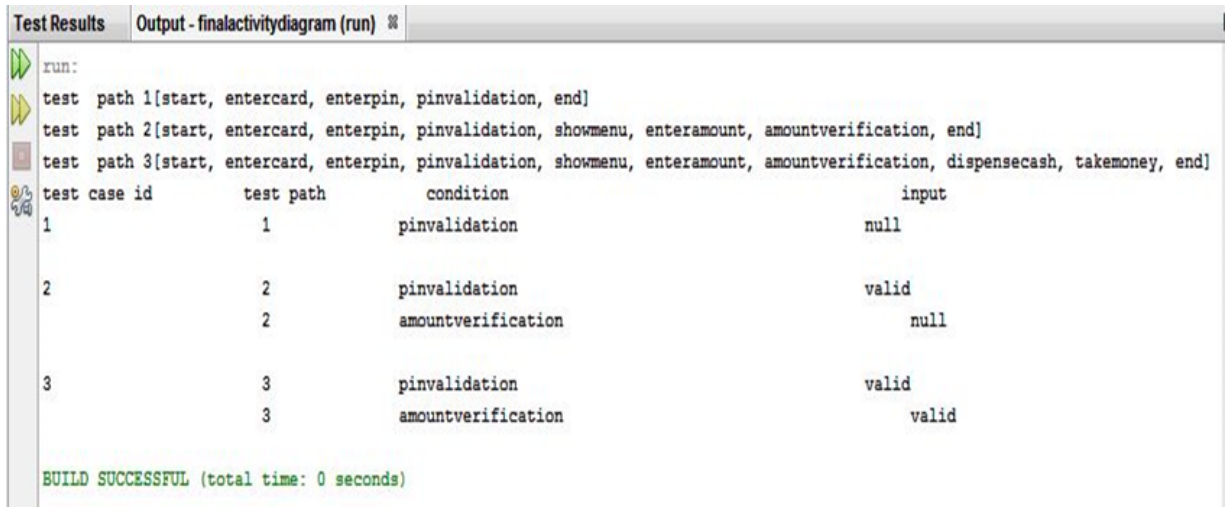


Figure 4.12: interediate graph

step5: generating the test cases



```
Test Results Output - finalactivitydiagram (run)
run:
test path 1[start, entercard, enterpin, pinvalidation, end]
test path 2[start, entercard, enterpin, pinvalidation, showmenu, enteramount, amountverification, end]
test path 3[start, entercard, enterpin, pinvalidation, showmenu, enteramount, amountverification, dispensecash, takemoney, end]
test case id    test path    condition    input
1               1           pinvalidation    null
2               2           pinvalidation    valid
                2           amountverification    null
3               3           pinvalidation    valid
                3           amountverification    valid
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4.13: test cases

Chapter 5

Test Scenario Generation from UML Composite Structure Diagram

In software testing we first perform the unit testing i.e. individual testing of each component. Unit testing guarantees that components individually performs and developed in proper way. In testing, we find several errors even though we perform unit testing correctly. After unit testing is completed, we go for integrating the components and we perform the testing and the components not perform in a proper manner because of many reasons , and main reason is that interface between components designed is not in a flexible manner. So, to find out the errors after integrating the components we need sufficient number of test cases. After unit test is completed, then the components are integrated and the product now goes under the integrating testing to find out the errors in the product. The UML composite structure diagram gives the details how the components behave and how they pass their data between components once they are integrated.

Different components will be developed in different programming languages and those developed components will run on different types of platforms [22]. By passing control and data the components will interact. Some components in unit testing show no error but when it comes to the integrated components they are not sure to show correct results. We have two main types of techniques black box and white box testing. Black box testing does not concern about the internal structure [23] whereas white box testing concerns about the internal structure of

the diagram [24].

5.0.2 Black Box Testing Technique

In this type of testing as name suggests we cant consider the internal structure of the product. In this testing we test the product by giving the required input and we wait for the output and we compare the actual output and real output. During the output generation we not consider the how the output is generating. We normally call this test as functional testing

5.0.3 White Box Testing Technique

Compare to the black box testing the white box testing talks about the internal structure of the product. So, in this testing we discuss about input , output and the most important the internal coding structure of the product i.e. we discuss how we are getting the output. We also known this testing as glass or open box or structural testing.

5.1 Basic Concepts and Definitions

Here, we discussing some terms which are used throughout the paper mainly when generating the test scenarios from the composite structure diagram.

5.1.1 Composite structure diagram

This diagrams shows the internal design of the component and also shows how the data is exchanging between the different components , how the control is passing between the different components which are connected to one another [25]. This diagram is one diagram which represents the system and used to model the product. Different symbols which used during the developing of the composite structure diagram.

1. **Component:** Component describes a particular part of the code , which gives servies to other components using some interfaces [26].






Symbols	Names
	Port
	Component
	Interface
	Provided Interface
	Required Interface

Figure 5.1: Basic Symbols of Composite Structure Diagram

2. **Port:** Port describes the set of messages and some operations which goes either in the component or it goes from the component.
3. **Interface:** As name suggests, interface gives a chance or medium to the components, with help of this the components will exchange the messages between them i.e. they can interact using the set of interfaces.
4. **Provided interface:** Provide interface when situated on particular component then that component sends the services to the other components which requesting that component to send services.
5. **Required interface:** A component with a required interface port receive services that are implemented by other components. Required interface when situated on particular component then that component receive the services from the other components to which it requested to send services.

Test Criteria: In top-down procedure we create (n-1) stubs. Here n represents the number of nodes present in the diagram [27].

Composite Structure Graph (CSG): This Graph is generated as intermediate graph during the execution. In this graph we have set of nodes and set of edges which connecting those edges. We developing a intermediate graph of online hoping system.

5.2 Proposed Approach

Here, we describe our approach which we proposed to develop test scenario using UML composite structure diagram. The process we used shown in a diagram below: 5.2.

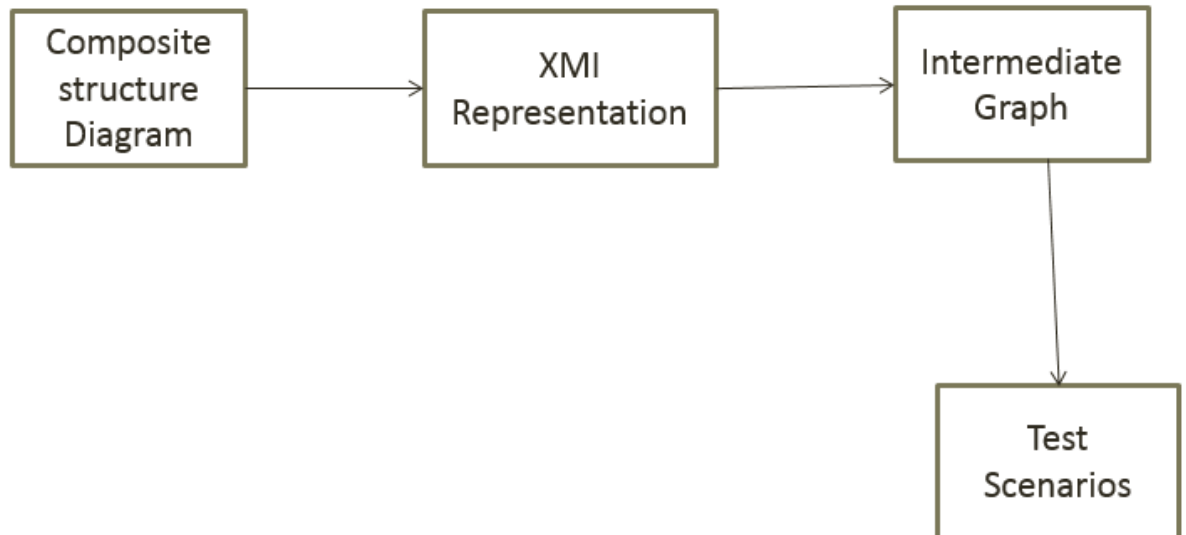


Figure 5.2: Block diagram of proposed approach

By using IBM Rational software Architect(RSA) we develop the composite structure diagram. In RSA we have a option to export the XMI of UML diagram. So we here extracted the XMI of the developed composite structure diagram using that option from in RSA. The developed java code the extractor will extract the required data from the exported XMI of composite structure diagram like name of the components, linkage between the different components. Using this extracted data by extractor a graph is generated which is intermediate graph. Using Gvedit tool [28] we can see the how the graph is and how the components are represented as nodes. Each node present in the intermediate graph is one component in the diagram. The interfaces we used in the diagram are shown as edges in the intermediate graph which is generated through extractor. In one pair of nodes we find two types of nodes. One node is represented as callee node and the second node is known as called node. Callee node are the components which obtain

the services from other nodes whereas called node are the components which give services to other components. The starting node which we call as root node, wont give any service to other nodes and not require any services to other nodes. Leaf nodes in the intermediate graph shows the components in diagram which are not requesting for any services. To implement the our work we developed the following algorithm. In this we following the top-down approach. We made changes in the breadth first search to develop the following proposed algorithm.

5.2.1 Proposed Algorithm

Input: intermediate graph

Output: test scenarios.

- 1: Stack st= null
- 2: Queue Q= null
- 3: s= null
- 4: testscenario ts: null
- 5: enqueue(Q, root)
- 6: repeat
- 7: s= Deque(Q)
- 8: push(st, s)
- 9: for j=1 to n do
- 10: Enque(Q,xi)
- 11: end for
- 12: ts= elements of Stack s U elements of Q
- 13: until q is not empty
- 14: empty(s)
- 15: exit

5.2.2 Description of Algorithm

This algorithm takes intermediate graph as input and gives back as output, a set of test scenarios. The algorithm maintain one queue Q, one stack st, one testscenario ts variable and x variable. At starting of the algorithms the , it maints queue ,

stack and testscenarios as null. First of all the root node from the intermediate graph is pushed into the Queue Q. Next the element is popped out from the queue and it is inserted into the stack st. Now all the nodes which are linked to the deleted node which we deleted from in queue are pushed into the Queue Q. Now, the nodes present in stack and queue are combined and they are inserted in the testscenario variable to generate the test scenarios. the testscenarios indicates, the nodes available in stack st , tested using integrating them with the stub nodes present in the queue Q. to exit the algorithm, if it find q is empty in terminates. If the queue Q in not yet empty, an node is removed from the queue and the procedure repeats otherwise s is emptied

5.2.3 Case Study

To implement our proposed algorithm, we have taken the example of online shopping. In this composite structure diagram we have eleven components and number of interfaces to connect them. The components are mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout, addtocart, orderitem, deleteitem and bankgateway. The interfaces are interface1, interface2, interface3, interface4, interface5, interface6, interface7, interface8, interface9, interface10 and interface11. We have following ports, port1, port2, port3, port4, port5, port6, port7, port8, port9, port10, port11, port12, port13, port14, port15, port16, port17, port18, port19, port20. The components are connected using the interfaces, required interface and provided interface. The component which requires data, uses the required interface and the component which gives services uses the provided interface. We develop this composite structure diagram in the RSA and we export its XMI. The java parser will take it as the input and generates the intermediate graph which contains the dependencies between the components. Form the intermediate graph, by using algorithm we generate the test scenarios.

5.2.4 Working of Algorithm

Step1: Developing the composite structure diagram in RSA

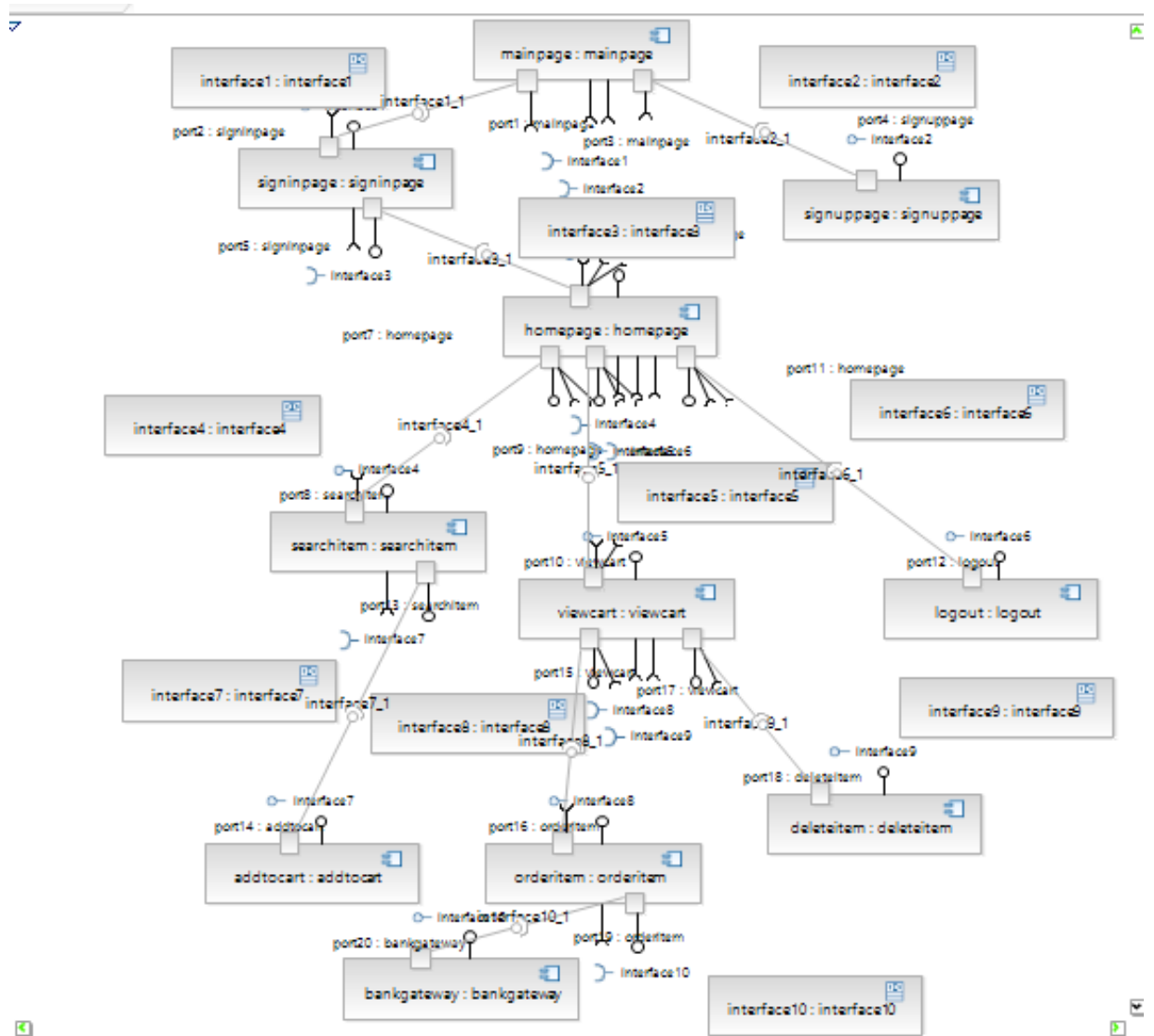


Figure 5.3: online shopping composite structure diagram

step2: Extracting the XMI of the diagram in RSA

```

<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQwF8LEeSpA7YFA9YLWQ" name="signuppage" visibility="private" type="_grCQoP8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQw8LEeSpA7YFA9YLWQ" name="addtocart" visibility="private" type="_grCQo_8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQw_8LEeSpA7YFA9YLWQ" name="orderiditem" visibility="private" type="_grCQov8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQXp8LEeSpA7YFA9YLWQ" name="deleteitem" visibility="private" type="_grCQqv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQXf8LEeSpA7YFA9YLWQ" name="bankgateway" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQXv8LEeSpA7YFA9YLWQ" name="interface1" visibility="private" type="_grCQsP8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQX_8LEeSpA7YFA9YLWQ" name="interface2" visibility="private" type="_grCQs8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQYp8LEeSpA7YFA9YLWQ" name="interface3" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQYf8LEeSpA7YFA9YLWQ" name="interface4" visibility="private" type="_grCQtv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQYv8LEeSpA7YFA9YLWQ" name="interface5" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQY_8LEeSpA7YFA9YLWQ" name="interface6" visibility="private" type="_grCQqv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQZp8LEeSpA7YFA9YLWQ" name="interface7" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQZf8LEeSpA7YFA9YLWQ" name="interface8" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQZv8LEeSpA7YFA9YLWQ" name="interface9" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedAttribute xmi:type="uml:Property" xmi:id="_grCQZ_8LEeSpA7YFA9YLWQ" name="interface10" visibility="private" type="_grCQv8LEeSpA7YFA9YLWQ" aggregation-
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQaP8LEeSpA7YFA9YLWQ" name="interface1_1" kind="assembly">
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQaf8LEeSpA7YFA9YLWQ" partWithPort="_grCQvP8LEeSpA7YFA9YLWQ" role="_grCQiv8LEeSpA7YFA9YLWQ"/>
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQav8LEeSpA7YFA9YLWQ" partWithPort="_grCQv_8LEeSpA7YFA9YLWQ" role="_grCQh_8LEeSpA7YFA9YLWQ"/>
</ownedConnector>
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQa_8LEeSpA7YFA9YLWQ" name="interface2_1" kind="assembly">
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQbP8LEeSpA7YFA9YLWQ" partWithPort="_grCQwF8LEeSpA7YFA9YLWQ" role="_grCQof8LEeSpA7YFA9YLWQ"/>
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQb8LEeSpA7YFA9YLWQ" partWithPort="_grCQv_8LEeSpA7YFA9YLWQ" role="_grCQiP8LEeSpA7YFA9YLWQ"/>
</ownedConnector>
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQbv8LEeSpA7YFA9YLWQ" name="interface3_1" kind="assembly">
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQb_8LEeSpA7YFA9YLWQ" partWithPort="_grCQvF8LEeSpA7YFA9YLWQ" role="_grCQj_8LEeSpA7YFA9YLWQ"/>
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQcP8LEeSpA7YFA9YLWQ" partWithPort="_grCQvP8LEeSpA7YFA9YLWQ" role="_grCQi_8LEeSpA7YFA9YLWQ"/>
</ownedConnector>
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQc8LEeSpA7YFA9YLWQ" name="interface4_1" kind="assembly">
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQcv8LEeSpA7YFA9YLWQ" partWithPort="_grCQv8LEeSpA7YFA9YLWQ" role="_grCQl8LEeSpA7YFA9YLWQ"/>
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQc_8LEeSpA7YFA9YLWQ" partWithPort="_grCQvF8LEeSpA7YFA9YLWQ" role="_grCQkP8LEeSpA7YFA9YLWQ"/>
</ownedConnector>
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQdP8LEeSpA7YFA9YLWQ" name="interface5_1" kind="assembly">
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQd8LEeSpA7YFA9YLWQ" partWithPort="_grCQv_8LEeSpA7YFA9YLWQ" role="_grCQm8LEeSpA7YFA9YLWQ"/>
  <end xmi:type="uml:ConnectorEnd" xmi:id="_grCQdv8LEeSpA7YFA9YLWQ" partWithPort="_grCQvF8LEeSpA7YFA9YLWQ" role="_grCQk8LEeSpA7YFA9YLWQ"/>
</ownedConnector>
<ownedConnector xmi:type="uml:Connector" xmi:id="_grCQd_8LEeSpA7YFA9YLWQ" name="interface6_1" kind="assembly">

```

Figure 5.4: XMI

step3: giving this XMI as input to the Java Parser

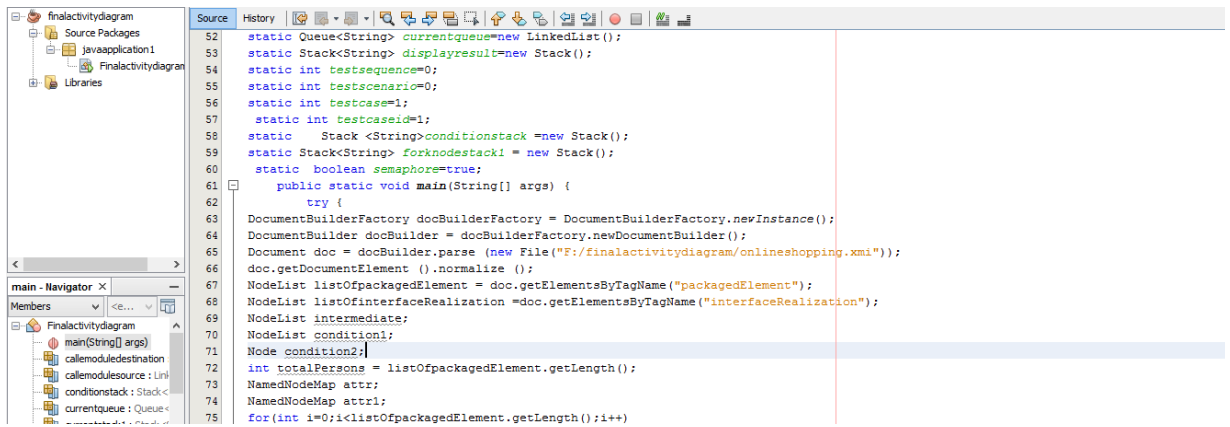


Figure 5.5: java parser

Step4: generatign the intermediate graph

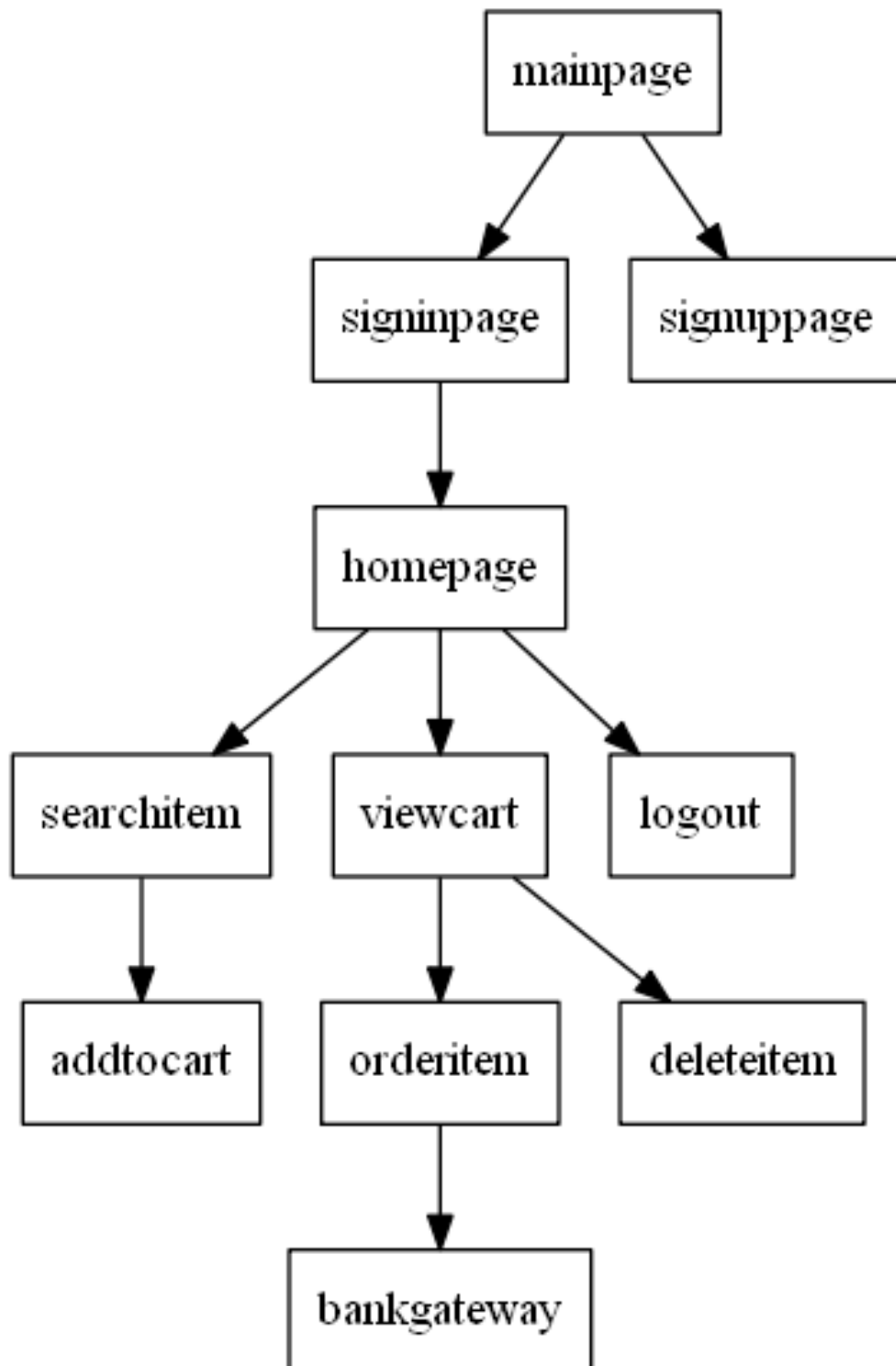


Figure 5.6: intermediate graph

step5: generating the test scenarios

```
testscenario for Top-Down integration testing

testscenario 1[mainpage]stub for[signinpage, signnuppage]
testscenario 2[mainpage, signinpage]stub for[signnuppage, homepage]
testscenario 3[mainpage, signinpage, signnuppage]stub for[homepage]
testscenario 4[mainpage, signinpage, signnuppage, homepage]stub for[searchitem, viewcart, logout]
testscenario 5[mainpage, signinpage, signnuppage, homepage, searchitem]stub for[viewcart, logout, addtocart]
testscenario 6[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart]stub for[logout, addtocart, orderitem, deleteitem]
testscenario 7[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout]stub for[addtocart, orderitem, deleteitem]
testscenario 8[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout, addtocart]stub for[orderitem, deleteitem]
testscenario 9[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout, addtocart, orderitem]stub for[deleteitem, bankgateway]
testscenario 10[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout, addtocart, orderitem, deleteitem]stub for[bankgateway]
testscenario 11[mainpage, signinpage, signnuppage, homepage, searchitem, viewcart, logout, addtocart, orderitem, deleteitem, bankgateway]:
```

Figure 5.7: generated test scenarios

Chapter 6

Conclusion and Future Work

We mainly focused on the how to develop the test cases using UML diagrams activity diagram and composite structure diagram. Here we are giving some short notes on our work. Finally we given some ideas for future work.

6.1 Test Scenario Generation from UML Activity Diagram

In this paper , we explained a technique to develop the test cases automatically using UML Activity diagram. The discussed technique is totally model based. We developed extractor which gives the intermediate graph automatically by taking the input as XMI of the Activity diagram which is exported from the RSA. By using the our proposed algorithm we produced the test scenarios from the intermediate graph by taking it as the input. The produced test scenarios are enough to find out in which the maximum error chances are present. In future we plan to generate test cases using UML composite structure diagram.

6.2 Test Case generation from Composite Structure diagrams

In this paper , we explained a technique to develop the test cases automatically using UML composite structure diagram. The discussed technique is totally model based. We developed extractor which gives the intermediate graph automatically by taking the input as XMI of the composite structure diagram which is exported

from the RSA. By using the our proposed algorithm we produced the test scenarios from the intermediate graph by taking it as the input. The produced test scenarios are enough to find out in which the maximum error chances are present. In future we plan to generate test cases using combination of UML activity diagram and composite structure diagram.

Bibliography

- [1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.
- [2] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, “A proposed test case generation technique based on activity diagrams.,” *International Journal of Engineering & Technology*, vol. 11, no. 3, 2011.
- [3] J. Arlow and I. Neustadt, *UML 2 and the unified process: practical object-oriented analysis and design*. Pearson Education, 2005.
- [4] S. K. Swain, S. K. Pani, and D. P. Mohapatra, “Model based object-oriented software testing.,” *Journal of Theoretical & Applied Information Technology*, vol. 14, 2010.
- [5] M. Aggarwal and S. Sabharwal, “Test case generation from uml state machine diagram: A survey,” in *Computer and Communication Technology (ICCCT), 2012 Third International Conference on*, pp. 133–140, IEEE, 2012.
- [6] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel, “Efficient object-oriented integration and regression testing,” *Reliability, IEEE Transactions on*, vol. 49, no. 1, pp. 12–25, 2000.
- [7] V. Le Hanh, K. Akif, Y. Le Traon, and J.-M. Jezeque, “Selecting an efficient oo integration testing strategy: an experimental comparison of actual strategies,” in *ECOOP Object-Oriented Programming*, pp. 381–401, Springer, 2001.

-
- [8] P. C. Jorgensen and C. Erickson, "Object-oriented integration testing," *Communications of the ACM*, vol. 37, no. 9, pp. 30–38, 1994.
- [9] W. E. Howden, "Weak mutation testing and completeness of test sets," *Software Engineering, IEEE Transactions on*, no. 4, pp. 371–379, 1982.
- [10] B. Demuth, H. Hussmann, and S. Obermaier, "Experiments with xmi based transformations of software models," in *Workshop on Transformations in UML*, 2001.
- [11] Y. Wang and M. Zheng, "Test case generation from uml models," in *45th Annual Midwest Instruction and Computing Symposium, Cedar Falls, Iowa*, vol. 4, 2012.
- [12] S. Asthana, S. Tripathi, and S. K. Singh, "A novel approach to generate test cases using class and sequence diagrams," in *Contemporary Computing*, pp. 155–167, Springer, 2010.
- [13] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test case generation based on use case and sequence diagram," *International Journal of Software Engineering, IJSE*, vol. 3, no. 2, pp. 21–52, 2010.
- [14] R. K. Swain, V. Panthi, D. P. Mohapatra, and P. K. Behera, "Prioritizing test scenarios from uml communication and activity diagrams," *Innovations in Systems and Software Engineering*, pp. 1–16, 2013.
- [15] O. Pilskalns, A. Andrews, S. Ghosh, and R. France, "Rigorous testing by merging structural and behavioral uml representations," in *UML 2003-The Unified Modeling Language. Modeling Languages and Applications*, pp. 234–248, Springer, 2003.
- [16] O. Pilskalns, A. Andrews, S. Ghosh, and R. France, "Rigorous testing by merging structural and behavioral uml representations," in *UML The Unified Modeling Language. Modeling Languages and Applications*, pp. 234–248, Springer, 2003.

-
- [17] S. Kansomkeat, P. Thiket, and J. Offutt, “Generating test cases from uml activity diagrams using the condition-classification tree method,” in *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, vol. 1, pp. V1–62, IEEE, 2010.
- [18] D. Kundu and D. Samanta, “A novel approach to generate test cases from uml activity diagrams,” *Journal of Object Technology*, vol. 8, no. 3, pp. 65–83, 2009.
- [19] H. Kim, S. Kang, J. Baik, and I. Ko, “Test cases generation from uml activity diagrams,” in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, vol. 3, pp. 556–561, IEEE, 2007.
- [20] M. Sarma, D. Kundu, and R. Mall, “Automatic test case generation from uml sequence diagram,” in *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*, pp. 60–67, IEEE, 2007.
- [21] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel, “Efficient object-oriented integration and regression testing,” *Reliability, IEEE Transactions on*, vol. 49, no. 1, pp. 12–25, 2000.
- [22] M. Sarma, D. Kundu, and R. Mall, “Automatic test case generation from uml sequence diagram,” in *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*, pp. 60–67, IEEE, 2007.
- [23] Y. Wu, M.-H. Chen, and J. Offutt, “Uml-based integration testing for component-based software,” in *COTS-Based Software Systems*, pp. 251–260, Springer, 2003.
- [24] Y. Wu, M.-H. Chen, and J. Offutt, “Uml-based integration testing for component-based software,” in *COTS-Based Software Systems*, pp. 251–260, Springer, 2003.

-
- [25] S. H. Edwards, “A framework for practical, automated black-box testing of component-based software,” *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 97–111, 2001.
- [26] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, “Automated whitebox fuzz testing,” in *NDSS*, vol. 8, pp. 151–166, 2008.
- [27] S. C. Lee and J. Offutt, “Generating test cases for xml-based web component interactions using mutation analysis,” in *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pp. 200–209, IEEE, 2001.
- [28] T. Sekulin, *Implementing a business process into an ERP solution*. PhD thesis, uniwiien, 2008.