

Test Case Generation
using UML
Behavioral & Structural Models

Pankaj Gupta

(Roll No: 212CS3120)



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India
June 2013

Test Case Generation
using UML
Behavioral & Structural Models

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

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Pankaj Gupta

(Roll No.- 212CS3120)

under the supervision of

Prof. D.P. Mohapatra



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

June 2013



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 Behavioral & Structural Models* by *Pankaj Gupta* 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: June 1, 2014

(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 Kumar 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.

Pankaj Gupta
Roll-212cs3120

Abstract

Quality software can be developed when it is properly tested. Due to increase in the size and complexity of object-oriented software, manual testing has become time, resource and cost consuming. Properly designed test cases discover more errors and bugs present in the software. The test cases can be generated much early in the software development process, during the design phase.

The unified modeling language (UML) is the most widely used language to describe the analysis and designs of object-oriented software. Test cases can be derived from UML models more efficiently. In our work, we propose a novel approach for automatic test case generation from the combination of UML class and activity diagrams. In our approach, we first draw the UML class and activity diagrams using IBM Rational Software Architect (RSA). Then, export the XML metadata interchange (XMI) from IBM Rational Software Architect (RSA). The XMI file is processed to extract variables from the class and predicates from activity diagram using Java code. The predicates are then used to generate the test cases. We have not used any intermediate form which makes the automation difficult. Our approach achieves 100% branch coverage and suitable for mutation testing and unit testing.

In our next work, we focus on UML composite structure diagram to generate test scenarios for integration testing. In our approach, we first draw the UML composite structure diagram using IBM Rational Software Architect (RSA). Then, export the XML metadata interchange (XMI) representation of composite structure diagram from IBM Rational Software Architect. Then, we parse the XMI code and generate the Component Structure Graph (CSG) automatically. Subsequently, we propose two algorithms to generate test scenarios for Top-Down and Bottom-Up integration approach. The generated test scenarios are sufficient enough to find the component in which probability of bug presence is maximum.

Keywords: Unified Modeling Language, test cases, test scenarios, integration testing, mutation testing.

Contents

Certificate	ii
Acknowledgement	iii
Abstract	iv
List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Model Based Testing	1
1.1.1 Benefits of Model Based Testing	2
1.2 Motivation	3
1.3 Problem Statement and Objectives	3
1.4 Thesis Organization	4
2 Basic Definitions and Concepts	6
2.1 Test Case	6
2.2 Testing Techniques	6
2.2.1 Black Box Testing Technique	6
2.2.2 White Box Testing Technique	7
2.2.3 Grey Box Testing Technique	7
2.3 Overview of UML diagrams	7
2.4 Coverage Criteria	8
2.5 Integration Testing	8
2.6 Mutation Testing	10
2.6.1 Types of Mutation Testing	10
2.6.2 Mutation Operator	10

2.6.3	Relational Operator Mutant	11
2.6.4	Mutation Score	12
2.7	Equivalence Class Partitioning	12
2.7.1	Boundary Value Analysis	12
2.8	XMI Metadata Interchange (XMI)	13
3	Review of Related Work	14
3.1	Test case generation using combination of UML diagrams	14
3.2	Test Case Generation for Integrating testing Using UML diagrams .	16
4	Test Case generation from combination of class and activity diagrams	18
4.1	Testing Coverage Criteria	18
4.1.1	Mutation coverage	18
4.1.2	Decision coverage:	19
4.2	Relevant UML Diagrams	20
4.2.1	Activity Diagram	20
4.2.2	Class diagrams	20
4.3	Proposed Approach	21
4.4	Proposed algorithm	23
4.4.1	Description of algorithm	23
4.5	Case Study	25
4.5.1	Working of algorithm	29
4.5.2	Analysis of mutation coverage	32
5	Test Scenario Generation from UML Composite Structure Diagram	34
5.1	Basic Concepts and Definitions	35
5.1.1	Composite structure diagram	35
5.2	Proposed Approach	36
5.2.1	Proposed Algorithm	37
5.2.2	Description of Algorithm	38
5.3	Case Study	39
5.3.1	Working of Algorithm	41

6 Conclusion and Future Work	46
6.1 Test Case generation from combination of class and activity diagrams	46
6.2 Test Scenario Generation from UML Composite Structure Diagram	47
Bibliography	48

List of Figures

1.1	Model Based Testing Process	2
4.1	Java code for CADExtractor	25
4.2	Class Diagram of Railway Reservation System	26
4.3	Activity Diagram of Railway Reservation System	28
4.4	XMI code of Railway Reservation System	30
4.5	Screenshot of generated test cases	33
5.1	Basic Symbols of Composite Structure Diagram	36
5.2	Block diagram of proposed approach	36
5.3	Composite Structure Diagram of Railway Reservation System . . .	40
5.4	XMI code of Railway Reservation System	41
5.5	Java code for CSDExtractor	42
5.6	Composite Structure Graph of Composite Structure Diagram in Figure 5.3.	43
5.7	Test Scenarios generated for Top-Down and Bottom-Up Integration Testing	45

List of Tables

4.1	Outcome of different possible relation	19
4.2	Table showing test input with expected output	31
4.3	Outcome of different possible relational operator mutant	32
4.4	Outcome of different possible mutant	32

Chapter 1

Introduction

Software testing usually involves executing a program on a set of tests and comparing the expected output with the actual output [1]. Testing is done to find the errors, which may later cause system failure. The Testing phase is carried out in three steps: test case generation, test execution and test evolution [2]. Test case generation requires a lot of effort and remaining two steps are relatively easy. Further, due to increase in size and complexity of software, the generation of effective test cases is becoming much more difficult. Manual testing requires a lot of time, cost and most important it is error-prone. So, automated testing is becoming more popular, as it requires less manpower. If the testing process begins before implementation, cost of the software development is reduced. Testing also measures the software quality in terms of its capability for reliability, correctness, maintainability, testability, usability and re-usability. Some of the objectives of testing are as follows:

- A quality test case should have high probability of finding an error.
- It ensures quality of the product.
- Software testing prevents the occurrence of failure.

1.1 Model Based Testing

Model based testing is testing technique in which test cases are derived from a model that describes some (usually functional) aspects of the system under test

(SUT). A model is a depiction of a system’s behavior. Models help us understand and envisage the system behavior. Model based testing involves three steps

1. Creating a model of system requirements for testing.
2. Generating test data from this requirement-model representation.
3. Verifying your design algorithm with generated test cases

A typical deployment of MBT in industry goes through the four stages shown in 1.1 The model is generally created from the requirement specification document.

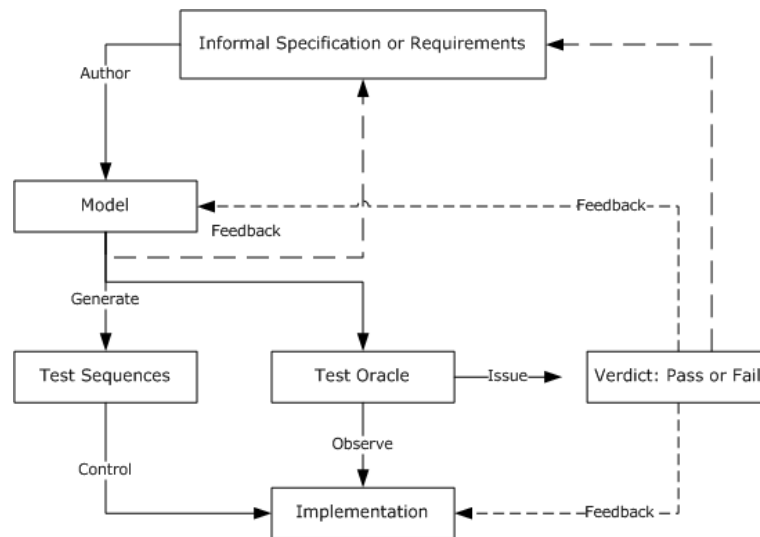


Figure 1.1: Model Based Testing Process

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.

1.1.1 Benefits of Model Based Testing

1. Model-based testing is easily understood by both the business and developer communities.
2. Model-based testing divides business rationale from testing code.

3. Model-based testing is the quickest approach to get utilization of automated testing.
4. Model-based testing empowers us to switch testing instrument if required or help various stages utilizing the same model.
5. Model-based testing focuses on requirement coverage.
6. Design more and code less.

1.2 Motivation

Now-a-days most of the project is developed in object-oriented language. These object-oriented languages are quite large and complex in nature because of its features, like encapsulation, inheritance, polymorphism. Automatically generating test cases from code of object-oriented programs are very much difficult because of its feature like encapsulation, inheritance, polymorphism, etc. The unified modeling language (UML) is most widely used language to model object-oriented designs. UML diagrams an important source of test case generation. In the recent days, researchers have considered different UML diagrams for generating test cases. Identifying error early during the design phase is much more efficient than identifying error after developing code. Integration testing and mutation testing are two important testing techniques in the testing process. In our thesis, we generate test cases from combination of class and activity diagram which is suitable for mutation testing. Next, we generate test scenarios from the UML composite structure diagram which is suitable for integration testing.

1.3 Problem Statement and Objectives

Software testing is a time consuming and expensive process in the software development life cycle. In traditional software testing methodology, if any, error occurs in the coding phase, then we have to go to that part of the code and design document from beginning to sort out the error. This issue can be solved by starting the testing process from the initial stage of the SDLC. In mode-based testing, test

cases can be designed from design phase, which is the second state in SDLC. UML is most widely used to represent models and we can design test cases from UML model very effectively. So based on the above, we have set the following objective.

- To propose a methodology to automatically generate test cases using combination of class and activity diagram.
- To propose a methodology to automatically generate the test scenarios using the UML composite structure diagram.
- Implementation of the proposed approach and evaluate their coverage such as decision coverage and mutation coverage.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter-2, includes basic concepts and different terminologies used in the rest of the thesis. The chapter contains the definitions of test case, test scenario, coverage criteria, integration testing, mutation testing. Then, we present some basic concepts of equivalence class partitioning with an example. Finally, we discuss the concepts of XML Metadata Interchange (XMI).

Chapter-3, provides a brief review of the related work relevant to our contribution. We have discussed various model based testing approaches and test case generation technique, in this chapter.

Chapter-4, present the technique for test case generation using a combination of class and activity diagram. We first discuss a few basic concepts and definitions used in describing our methodology. Next, we describe our proposed methodology to generate test cases. Finally, we illustrate our methodology with an example of the Railway Reservation System.

Chapter-5, present the technique for test scenario generation using Composite Structure diagrams. We first discuss a few basic concepts and definitions used in describing our methodology. Next, we describe our proposed methodology to generate test scenarios. Finally, we illustrate our methodology with an example of the Railway Reservation System.

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

It is essential to discuss some basic concepts and definitions to understand the thesis report. In this chapter, we have discussed some of the basic concepts and terminologies, on which our research is based.

2.1 Test Case

Test cases are built using specifications and requirements document, i.e., what the system needs to perform.

A test case is a triplet (I, S, O) where I is the data input to the system, S is the state of the system to which the data is input, and O is the expected output obtained from the system [4]. Combination of test cases with which a given software product is to be tested is called test suite [5].

2.2 Testing Techniques

Testing techniques are mainly divided into three categories:

2.2.1 Black Box Testing Technique

Black-box testing examines the functionality of an system without having the knowledge of internal logic of code. In a black box testing the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. It also known as functional testing.

2.2.2 White Box Testing Technique

In white box testing test cases are designed based on analysis of some aspect of source code and is based on some heuristic. White box testing is also called glass testing, open box testing and structural testing. To perform white box testing on an application, the analyzer needs to have learning of the inward working of the code.

2.2.3 Grey Box Testing Technique

Grey black box testing is a combo of black box testing and white box testing. The analyzer has the constrained learning of the inside workings of an application. It is focused around the interior information structures and calculations for planning the experiments more than black box testing however short of what white box testing. This system is imperative when directing integration testing between two modules of code composed by two separate developers, where just interfaces are uncovered for test.

2.3 Overview of UML diagrams

Unified Modeling Language (UML) is defined as a graphical idiom for envisioning, identifying, creating and documenting the artifacts of a software system. UML is a blueprint of the actual system and helps in documentation of the system [3]. It makes any complex system easily understandable by the disparate developers who are working on different platforms. Another benefit is that UML model is not a system or platform specific. Modeling is an indispensable part of the huge software project, which as well facilitates in the improvement of Medium and small projects. The UML 2.0 has fourteen diagrams, to model different software artifacts. The increase in its popularity encourages us to use these models as an important source for test case generation.

There are three important types of uml diagrams.

1. **Structure diagram:** Structure diagrams highlight the things that must be available in the system being modelled. Structure modeling captures

static features of a system. Some of structural diagrams are Object Diagram, Component Diagram, Class Diagram, Package Diagram, Composite Structure Diagram, and Deployment Diagram etc.

2. **Behavioral diagram:** behavioral diagrams describe the interaction in the system. It represents the interaction among the structural diagram. Behavioral diagram shows the dynamic nature of the system. Some of behavioral diagrams are Activity Diagram, Use Case Diagram, and State Machine Diagram etc.
3. **Interaction diagrams :** It highlights the flow of data and control among the things present in the system being modeled. Some of interaction diagrams are Interaction Overview Diagram, Sequence Diagram, Communication Diagram, Timing Diagram and etc.

2.4 Coverage Criteria

Code coverage is a measure used to depict the degree to which the source code of a program is tested by a specific test suite. A program with high code coverage has been more comprehensively tested and has a minor chance of containing software bugs.

Some of the coverage criteria are.

1. **Statement coverage:** All the statement in a code is executed at least once.
2. **Branch Coverage:** Every decision in the program has taken all possible outcomes at least once.

2.5 Integration Testing

In integration testing, all the individual components are tested by combining them into a group [7]. Integration testing is done once all the components are tested individually i.e after the unit testing is completed. Integration testing is performed to find faults that occur when two or more components interact with each other [8].

During integration testing some of the components may not be ready for integration, so we require stub and driver to simulate the behavior of actual components. If everything works until we add Component C2 and then Component C1 stops working, then it implies that the error may be likely present in either component C1 or C2.

Stubs: Stubs are dummy code that simulates the functionality of low level components [7].

Driver: Driver is dummy code that simulates the functionality of high level components [7].

There are four major types of integration testing.

1. **Top-Down:** In top-down integration testing first top level components are integrated and tested, then lower level components are tested step by step after that. Stubs are created to simulate lower level components which may not be completed initially [9].
2. **Bottom-Up:** In bottom-up integration testing lower level components are integrated and tested, and then upper level components are tested step by step after that. Drivers are created to simulate the upper level components which may not be completed initially [9].
3. **Sandwich:** Sandwich integration testing is the combination of both bottom-up and top-down integration testing.
4. **Big Bang:** In this testing some or all the component is integrated and tested.

Among the four types of integration testing techniques, top-down and bottom-up techniques are most widely used in industry. The Big Bang approach requires less time and cost to integrate. However, it is difficult to find the component in which error is present. System testing is performed once all the components are integrated and unit tested completely. In this paper, we generate test scenarios using Top-down and Bottom-Up integration approach.

2.6 Mutation Testing

Mutation testing is a method of inserting faults into programs to test whether the tests pick them up, thereby validating or invalidating the tests. Mutant is said to be killed when the test cases are able to detect change in mutant otherwise it is said to be alive. The quality of test cases are measured by percentage of mutant killed.

Mutation testing is one of the error-based testing methods. The goal in mutation testing is to construct a set of test cases T which will distinguish between a given program P and its mutant program P' [10]. The mutant P' is generated by applying mutation transformations to components of P [10]. There is only one change made in each mutant program.

2.6.1 Types of Mutation Testing

There are two types of mutation testing.

- **Weak mutation testing:** In weak mutation coverage [10], we suppose that a program contains a particular simple kind of fault.

Weak mutation testing are of two types. Operator coverage requires test cases that differentiate operators from other operators. Operand coverage requires test cases that differentiate operands from other operands.

- **Strong mutation testing:** In strong mutation, a mutant m of a given program p is said to be killed only if mutant m gives a different output from the original program p .

Both strong and weak mutation testing can be applied to general classes of programs in any language.

2.6.2 Mutation Operator

Mutation Operators are the operators that can be applied to the original program to make it a mutated one. Since we are considering object oriented pro-

programming (OOP) now a days like Java and C++ etc, some of the OOP mutation operators are:

- Access Modifiers.
- Argument order change.
- Arithmetic Operator change.
- Relational operator change.
- Parameter Change.

2.6.3 Relational Operator Mutant

A relational operator compares two values and determines the relationship between them. Most of the programming language has six kinds of relational operators; $>$, $>=$, $<$, $<=$, $==$, $!=$. Because these operators take two operands, only replacement is allowed for the relational operators. In relational operator mutant one relational operator is replaced with other relational operator [11].

Example:

Original LOC:

```
if(x>=y)
printf(correct);
else
printf(wrong);
```

If we change the relational operator then possible mutant are:

Mutant1: $x>y$;

Mutant2: $x<=y$;

Mutant3: $x<y$;

Mutant4: $x=y$;

Mutant5: $x!=y$;

2.6.4 Mutation Score

Mutation score (MS) is the ratio of the number of Dead Mutants over the number of Non Equivalent Mutants. The goal is to have a score of one 100 % , which means that all faults in all mutants have been detected; the more dead mutants the higher the score will be. This technique is used for adequacy testing. The formula for mutation score is given below:

$$MS = (\text{killed Mutants} / \text{Total Mutants}) * 100\%$$

2.7 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. Test cases are designed for each equivalence class. All the test cases belonging to same class behaves similarly.

- Suppose you have a software which accepts values 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 201 and above

2.7.1 Boundary Value Analysis

In Boundary value analysis test cases are designed using the values at the boundaries of different equivalence classes. An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. Test cases designed with boundary input values have a high chance to find errors. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc. The test cases generated for equivalence class partitioning discussed in the above section 2.7.

- **Test Cases 1:** Generate test case precisely at the extremes of input domain i.e. values 100 and 200.

- **Test Cases 2:** Generate test case precisely, just below the limits of input domains i.e. values 99 and 199.
- **Test Cases 3:** Generate test case precisely, just above the limits of input domains i.e. values 101 and 201.

2.8 XMI Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is an interchange format for metadata defined in terms of the Meta-Object Facility (MOF) standard [12]. XMI plays a key role in IBM Rational Software Architecture to represent UML models in XMI metadata format [13].

Chapter 3

Review of Related Work

This chapter presents an overview of the existing method to generate test data using UML diagrams. First, we discuss the previous related work done by researchers on the topic of test scenario generation using UML diagrams for integration testing and then Proceed to discuss the related work done in Test case generation from a combination of UML diagrams.

3.1 Test case generation using combination of UML diagrams

Wang et al. [17] proposed an approach to generate the test case from an interaction and class diagrams. The test adequacy criteria they used is the coverage of the design model elements. 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 method is introduced by Asthana et al. [18] for generating test cases using class and sequence diagrams. First, they get the lower and upper bound of variable from the given class diagram. Then, they have traversed the sequence diagram to obtain all the variable passed. Out of these variables, they found out the variables on which the output will differ and have applied robustness testing on these variables to compare the results. They have automated the process by parsing xml

metadata interchange (XMI) and not used any intermediate representation.

A method is proposed by Swain et al. [19] to generate test case based on use case and sequence diagram. They constructed Concurrent Control Flow Graph (CCFG) from sequence diagrams and Use case Dependency Graph (UDG) from use case diagram to generate test sequence. They have used UML 2.0 sequence diagram for generating test cases. They have developed a semi automated tool (ComTest) which takes XMI representation of sequence diagram as input and generate the test cases. Their testing strategies to derive test cases uses full predicate coverage criteria. The generated test cases are suitable for detecting dependency of use cases and synchronization and messages, object interaction and operational faults.

A methodology is proposed by Swain et al. [6] to prioritize test scenario from UML communication and activity diagrams. They presented an integrated approach 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. [20] 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. [21] have proposed test case generation technique from a ac-

tivity diagram. They proposed an algorithm that automatically creates a table called Activity Dependency Table (ADT). The activity dependency table is then used to create a directed graph called Activity Dependency Graph (ADG). At last ADG with the ADT are used to generate the final test cases. The basic step that was taken to generate test cases are 1) Generation of ADT. 2) Generation of ADG 3) Test case Generation 4) validate generated test cases.

Kansomkeat et al. [22] have proposed a methodology to generate test cases by converting activity diagrams into the Condition-Classification Tree Method (CCTM). They have generated the test cases in three steps 1) Generate condition-classification trees using UML. activity diagram. 2) Create test case table. 3) Generate test cases. They first build an I/O explicit Activity Diagram (IOAD) from an ordinary UML activity diagram and then transforms it into a directed graph, from which test cases for the initial activity diagram are derived.

Kim et al. [24] uses activity diagram to generate test cases. They first convert the activity diagram into I/O explicit Activity Diagram (IOAD). IOAD is then converted to directed graph. Directed graph is then traversed to generate the test cases. The criteria for conversation is based on the single stimulus principle, which avoid the state explosion problem. They have used the all-paths test coverage criterion.

3.2 Test Case Generation for Integrating testing Using UML diagrams

Many researchers have been working in generating test cases from different UML diagrams. In this section, we review the existing work which are nearly related to our approach. Sarma et al. [14] have proposed automatic test case generation from UML sequence diagram. They transformed the sequence diagram into sequence diagram graph. The sequence diagram is then traversed to generate test cases. The generated test cases are used for system testing and to detect interaction and scenario fault.

Traon et al. [7] proposed a methodology for planning integration and regression

testing from an object-oriented (OO) model. They discussed the refinement process of the OO design to produce a model of structural system test dependencies graph (TDG). The generated graph TDG is used for ordering classes and methods to be tested for integration and regression testing. They minimized the number of stubs required during testing process.

Hartmann et al. [15] proposed an approach for modeling components and interaction among them. They used UML statecharts to model the dynamic behavior of the components as well as the communication between them. The interaction between the components is done via message exchange containing no parameters and values. In our approach there is no constraint on message. Components can interact via message containing parameters and values.

Hanh et al. [8] proposed two integration testing strategies. First one is based on deterministic approach and is called Triskell. The second approach is based on genetic algorithm and is called Genetic. They first build the UML class diagram and Package diagram to find the dependency between components. The gathered information is converted into test dependency graph (TDG). Their objective was stub minimization and testing resource allocation. This approach is not suitable for finding erroneous components. In our approach, the probability to find the component in which bug is present is more.

Wu et al. [16] proposed test criteria to test component-based software using UML diagrams. They used UML statechart diagram to characterize the internal behavior of objects in a component. They have used interaction diagrams to evaluate the control flows of components. In most of the related work researchers consider the combination of different UML diagram, but this is not suitable to represent the components.

Chapter 4

Test Case generation from combination of class and activity diagrams

In this chapter, we use UML class and activity diagrams as design specification. We present a testing methodology to test Object Oriented software based on combination of class and activity diagram. Our approach achieves much important coverage like branch coverage and mutation coverage. This chapter describes our proposed methodology in detail.

4.1 Testing Coverage Criteria

In this section, we define testing coverage criteria based on activity diagram. Now, we discuss some of the relevant coverage criteria which are achieved in our approach.

4.1.1 Mutation coverage

The effectiveness of test cases can be evaluated using a fault injection technique called mutational analysis. Mutation testing is a process by which faults are injected into the system to verify the efficiency of the test cases. The product of mutation analysis is a measure called Mutation Score, which indicates the percentage of mutants killed by a test set.

Relational mutation: Let R be an arithmetic relation, and suppose that R' is a wrong relation mutation of R [8]. Suppose R is of the form $x r y$ and R' is $x r'$

y. If R is executed over data for which $x < y$, $x = y$, and $x > y$, then in at least one case it will give a different output from R'. This can be seen by examining Table 4.1. For each possible kind of relation, the outcome of R and R' will differ in at least one case.

Table 4.1: Outcome of different possible relation

$x < y$	$x \leq y$	$x > y$	$x \geq y$	$x == y$	$x != y$
T	F	F	T	T	F
F	T	F	F	T	T
F	F	T	T	F	T

4.1.2 Decision coverage:

Every decision in the program has taken all possible outcomes at least once. decision coverage is also called as branch coverage. Formula for branch coverage is:

Decision Coverage = (Number of decision outcomes executed / Total number of decision outcomes) * 100%

Eg:

Original LOC:

```
if(a >= b)
printf(correct);
else
printf(wrong);
```

To achieve 100 % decision coverage both the FALSE and TRUE condition of the IF statement should be covered. To achieve this we need to consider the following test cases.

- **Test Cases 1:** a=10, b=5 makes the if condition TRUE.
- **Test Cases 2:** a=5, a=10 makes the if condition FALSE.

So, to achieve the 100 % decision coverage minimum two number of test cases are required.

4.2 Relevant UML Diagrams

In this section first we describe activity diagram in detail. Next, we describe the class diagram.

4.2.1 Activity Diagram

Activity diagram describes the workflow behavior of the system [33]. Activity diagram shows the flow of activities through the system. Activity diagram is used to model the dynamic behavior of the system. Some of common symbol used in activity diagram is; activity, transitions, swimlanes, initial node, final node, fork, join, decision, merge and swimlanes. Activity represent a business process. An initial node is a control node at which flow starts. Final node indicates that an activity is completed. A typical activity diagram has one initial node and one or more final node. Transition show the control flow among the activities. A branch node has one incoming transition and multiple outgoing transitions. Each output transition is labeled which represent the Boolean expression to be satisfied to choose the branch. A merge node has multiple incoming transitions and one outgoing transition. In activity diagram, concurrent execution behavior is represented by fork-join structure. Fork represents the start and join, represent the end of concurrent execution. Fork node splits the flow into multiple concurrent flows. A fork node has one incoming flow and multiple outgoing flow. Join node combines multiple concurrent flow into a single flow. Join node has multiple incoming flow and a single outgoing flow. Swimlanes are used to arrange the actions of an activity into areas corresponding to different object that perform the action.

4.2.2 Class diagrams

A class diagram describes the structure of a system by showing the system's classes, their attributes, operations, and the relationships among classes. It represents the static view of the system. The class diagram can be used for constructing executable code of the software application. The class diagrams can be directly

mapped with object oriented languages and thus widely used at the time of construction. Some of the basic symbols used in class diagram are; class, relationship, multiplicity etc. In class diagram classes are represented with boxes which contains three parts; the top part contains the name of class, the middle part contains the attribute of the class, the bottom part contains the methods which the class can perform. There are three types of relationships: association, generalization/specialization and aggregation. Classes represent the problem concepts; associations model the semantic relationships between problem concepts [17]. In Generalization relationship one of the two related classes is considered to be a specialized form of the other class. The aggregation is one kind of association. A multiplicity indicates how many instances of one class are related to another class.

4.3 Proposed Approach

In this section, we discuss our proposed approach to generate test scenarios from class and activity diagrams. The four basic steps in our approach are

1. Construct activity and class diagram.
2. Generate the XMI code.
3. Parse the XMI code.
4. Apply our proposed algorithm.
5. Calculate the decision coverage.

Construct activity and class diagram: Initially the activity diagram and class diagram are constructed using IBM Rational Software Architect (RSA). RSA is most widely used tool to construct the UML diagrams.

generate the XMI code: Next, we export the XMI code of the relevant diagrams draw in step 1 using RSA. XMI format is the standard format used for UML diagrams portability across different object-oriented software. A single combined XMI file for the whole project is generated using RSA tool. The XMI file

contains all the necessary information needed to inter-operate between different tools and transfer UML diagrams.

Parse the XMI code:

We develop a parser called *CADExtractor* shown in Figure 4.1. The *CADExtractor* extract the necessary information such as name of attribute present in the class diagram, predicates present in activity diagram from the XMI code. *CADExtractor* traverse the XMI file in sequential order. For each class, obtain the attributes present in the class.

```
<packagedElement xmi:type="uml:Class" xmi:id="V1FPg6-zEeOOwoWso8qpsg"
name="bankaccount">
<ownedAttribute xmi:type="uml:Property" xmi:id="V1FPhq-zEeOOwoWso8qpsg"
name="numberofattempt" visibility="private">
```

As we can see in the sample XMI file, there exists a class bankaccount with one of the attribute numberofattempt. XMI-IDs are used as reference for later processing. For each activity, obtain the predicate.

```
<edge xmi:type="uml:ControlFlow" xmi:id="V1FPOK-zEeOOwoWso8qpsg"
name="numberofattempt>3" source="V1FO3a-zEeOOwoWso8qpsg"
target="V1FO3q-zEeOOwoWso8qpsg">
```

As we can see in the sample XMI file, there exist a decision numberofattempt>3. All these class attribute and predicate is stored in attributearray and decisionarray respectively. These two arrays are given as input to our proposed algorithm. The total number of decision node in stored in variable totalnumberofbranch and the output is stored in outputarray.

proposed algorithm: To implement our proposed work, we developed algorithm Integrated test case generation algorithm (ITCGA). Detail of ITCGA is discussed in section 4.4.

Calculate the decision coverage: The formula for calculating the decision coverage is:

decision coverage=(number of decision outcome exercised)/(number of decision

outcome))*100%

Each decision node in activity diagram takes two possible outcome. So, total number of decision outcome=number of decision node*2.

Total number of decision outcome exercised is obtained by Algorithm 1.

4.4 Proposed algorithm

In this section, we present our ITCGA algorithm 1 to generate test cases, in pseudo code form.

4.4.1 Description of algorithm

Input to our Algorithm 1 is attributearray, decisionarray. Output of the Algorithm 1 is set of test cases and total number of branch covered. At the beginning of algorithm 1, the testcaseid and branchcovered variable are initialized to null. The Algorithm traverse the decisionarray in sequential order. We have considered the binary relational operator such as ==, !=, >, <, >=, <= i.e., each operator has two operands. The left operand is extracted and it is checked whether it is present in decisionarray or not. If it is not present the algorithm display the message that operand is not present in any class, otherwise test case is generated. The left operand of an operator is variable and the right operand is an integer value. Right operand is assigned to testinput variable and the value of testcaseid is incremented by 1. Left operand is assigned to testcondition variable. Next, algorithm check for the type of operator. If the operator is >= the algorithm print the two test case as (testcaseid, testcondtion, testinput, expected output). Other test case is (testcaseid, testcondtion, testinput+1, expected output). If the operator is > the algorithm print the one test case as (testcaseid, testcondtion, testinput+1, expected output). If the operator is <= the algorithm print the two test case as (testcaseid, testcondtion, testinput, expected output). Other test case is (testcaseid, testcondtion, testinput-1, expected output). If the operator is < the algorithm print the one test case as (testcaseid, testcondtion, testinput-1, expected output). If the operator is != the algorithm print the one test case

Algorithm 1 Integrated Test Case Generation Algorithm (ITCGA)

Input: attributearray, decisionarray, outputarray

Output: set of test cases, number of branch covered

```

1: testcaseid  $\leftarrow \phi$ 
2: branchcovered  $\leftarrow \phi$ 
3: testinput  $\leftarrow \phi$ 
4: testcondition  $\leftarrow \phi$ 
5: operatorsymbol  $\leftarrow \phi$ 
6: for each elementj in decisionarray do
7:   testcondition  $\leftarrow$  leftoperand
8:   if testcondition present in attributearray then
9:     operatorsymbol  $\leftarrow$  operator
10:    testinput  $\leftarrow$  rightoperand
11:    if operatorsymbol== '  $\geq$  ' then
12:      branchcovered  $\leftarrow$  branchcovered + 1
13:      print (testcaseidi, testcondition, testinput, outputarrayj)
14:      testcaseid  $\leftarrow$  testcaseid + 1
15:      print (testcaseidi, testcondition, testinput+1, outputarrayj)
16:    else
17:      if operatorsymbol== '  $>$  ' then
18:        branchcovered  $\leftarrow$  branchcovered + 1
19:        testcaseid  $\leftarrow$  testcaseid + 1
20:        print (testcaseidi, testcondition, testinput+1, outputarrayj)
21:      end if
22:    else
23:      if operatorsymbol== '  $<$  ' then
24:        branchcovered  $\leftarrow$  branchcovered + 1
25:        testcaseid  $\leftarrow$  testcaseid + 1
26:        print (testcaseidi, testcondition, testinput-1, outputarrayj)
27:      end if
28:    else
29:      if operatorsymbol== '  $\leq$  ' then
30:        branchcovered  $\leftarrow$  branchcovered + 1
31:        testcaseid  $\leftarrow$  testcaseid + 1
32:        print (testcaseidi, testcondition, testinput, outputarrayj)
33:        testcaseid  $\leftarrow$  testcaseid + 1
34:        print (testcaseidi, testcondition, testinput-1, outputarrayj)
35:      end if
36:    else
37:      if operatorsymbol== ' = ' then
38:        branchcovered  $\leftarrow$  branchcovered + 1
39:        testcaseid  $\leftarrow$  testcaseid + 1
40:        print (testcaseidi, testcondition, testinput, outputarrayj)
41:        testcaseid  $\leftarrow$  testcaseid + 1
42:      end if
43:    else
44:      if operatorsymbol== '  $\neq$  ' then
45:        branchcovered  $\leftarrow$  branchcovered + 1
46:        testcaseid  $\leftarrow$  testcaseid + 1
47:        print (testcaseidi, testcondition, testinput-1, outputarrayj)
48:        print (testcaseidi, testcondition, testinput+1, outputarrayj)
49:      end if
50:    end if
51:  else
52:    Print testcondition not present in any class
53:  end if
54: end for

```

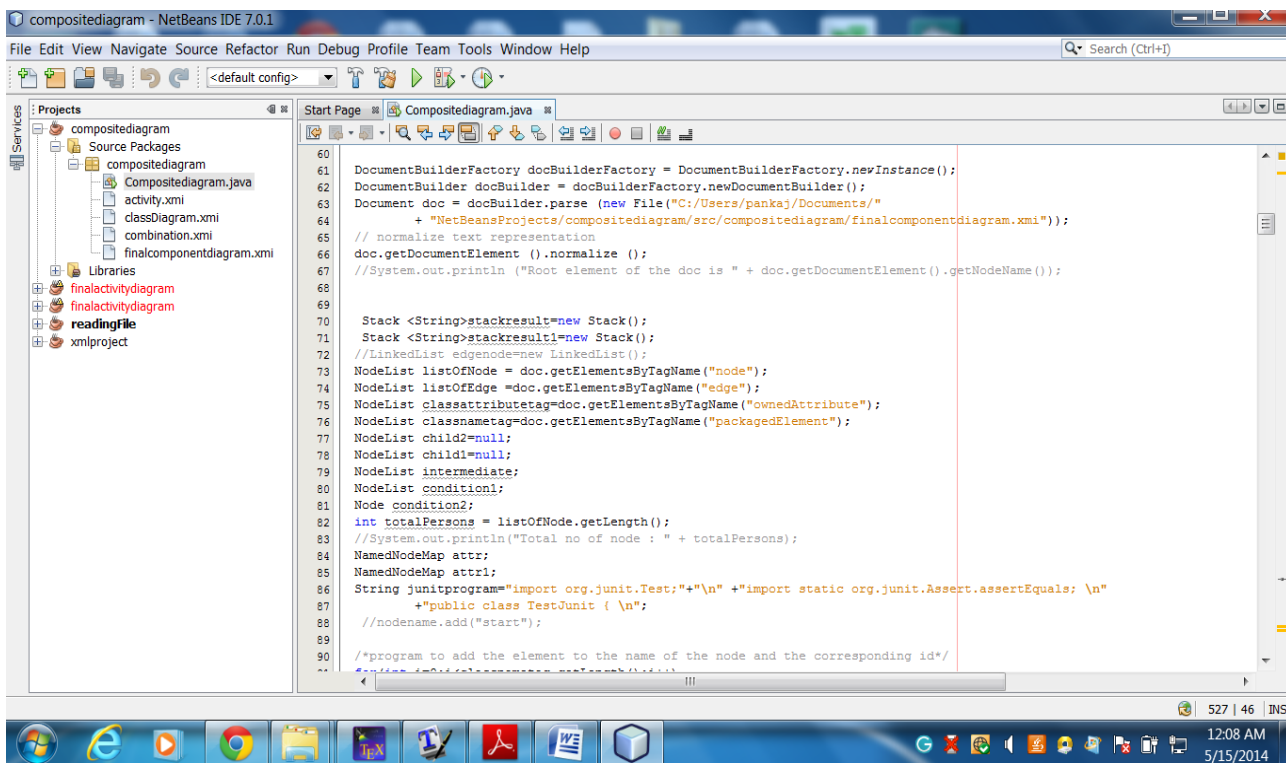


Figure 4.1: Java code for CADEXtractor

as (testcaseid, testcondition, testinput-1, expected output). Other test case is (testcaseid, testcondition, testinput+1, expected output). If the operator is == the algorithm print the two test case as (testcaseid, testcondition, testinput, expected output). The process is repeated until decisionarray is empty.

4.5 Case Study

We consider the example of railway reservation system to discuss our proposed approach. We model the class diagram and activity diagram of railway reservation system in RSA which is shown in Figure 4.2 and Figure 4.3 respectively.

Figure 4.2 represents the class diagram of railway reservation system. There are five classes named as; train, bankaccount, passenger, ticket, railwaysystem. Each class has three parts. There are five attributes in train class; trainno, trainname, numberofseatleft, source and destination. The train class does not perform any operation so, it does not contain any method. There are four attribute in the passenger class; name, age, gender, contact number. The passenger can book and

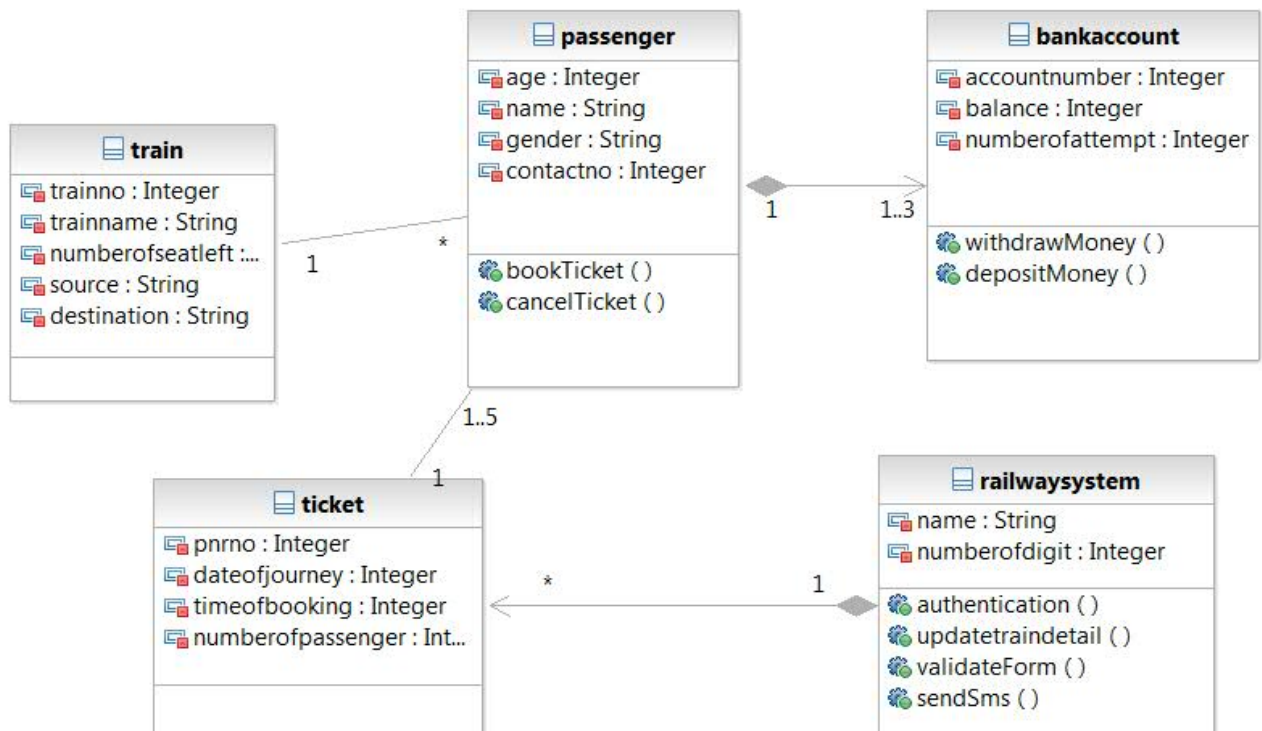


Figure 4.2: Class Diagram of Railway Reservation System

cancel ticket via bookTicket and the cancelTicket method in passenger class. The ticket class has four attributes; pnno, dateofjourney, timeofbooking, numberofpassenger. Similarly the bankaccount and railwaysystem have their corresponding attribute and method as given in figure 1. The link between train and passenger class is 1 to * association. A train can have any number of passengers. The link between ticket and passenger is 1 to 1..5 association, i.e. With a single ticket maximum five people can travel. The link between passenger and bank account is 1 to 1..3 composition. A passenger can have three bank account and composition relationship indicates that when a passenger does not exist his bank account will also not exist. Similarly, link between ticket and the railwaysystem is * to 1 composition.

Figure 4.3 shows the activity diagram for railway reservation system. First user visits the website. Then the user enters form and to station code and enter the book ticket button. Now the system check in time of booking, in case it is greater than or equal to 8, the system will ask for entering the date of journey;

otherwise the system displays an error message to the user that Inter incoming day booking not allowed before 8 AM. The user then enters the date of journey. Now the system check for date of journey, in case it is below or equal to 60 days the system shows the list of trains; otherwise it shows an error message that date of journey is beyond the advance reservation period. The user then selects one of the trains. Next, the system check for whether the seat is available or not; in case seat is available, the system display an information detail form; otherwise the system display a message to the user that seat not available. The user then enters the number of passengers. The system checks for number of people travelling if it is greater than or equal to 5; the system displays the error message that number of passengers per ticket cannot exceed 5; otherwise the system will ask for entering the mobile number. The user then enters the mobile number. The system check for number of digits in mobile number; in case number of digits is equal to 10 it send the validation code; otherwise the system displays an error message that please enter the 10 digit number. The system checks for senior citizens; if the age is greater than 60 then it will show to be eligible for senior citizen concession, the passenger should be 60 years or more error message; otherwise the system will display the amount.

The user chooses to pay amount and the system will display the list of banks. The user chooses one of the bank and enter username name and password. User proceed further and enter the profile password. If the user enters the wrong password an error message will be displayed that profile password is wrong, please try again. The number of attempts should not be greater than 3, if it exceeds 3 the system will display number of attempt exceeds 3 your account is blocked for 1 day. In order to generate the ticket total balance should then be greater than the amount; otherwise insufficient amount message is displayed. The transfer of control is represented by swimlanes. Three swimlane are required to represent the control flow, first for passenger, second for railway authority, third for the bank. The flow between the activity are represented by arrows.

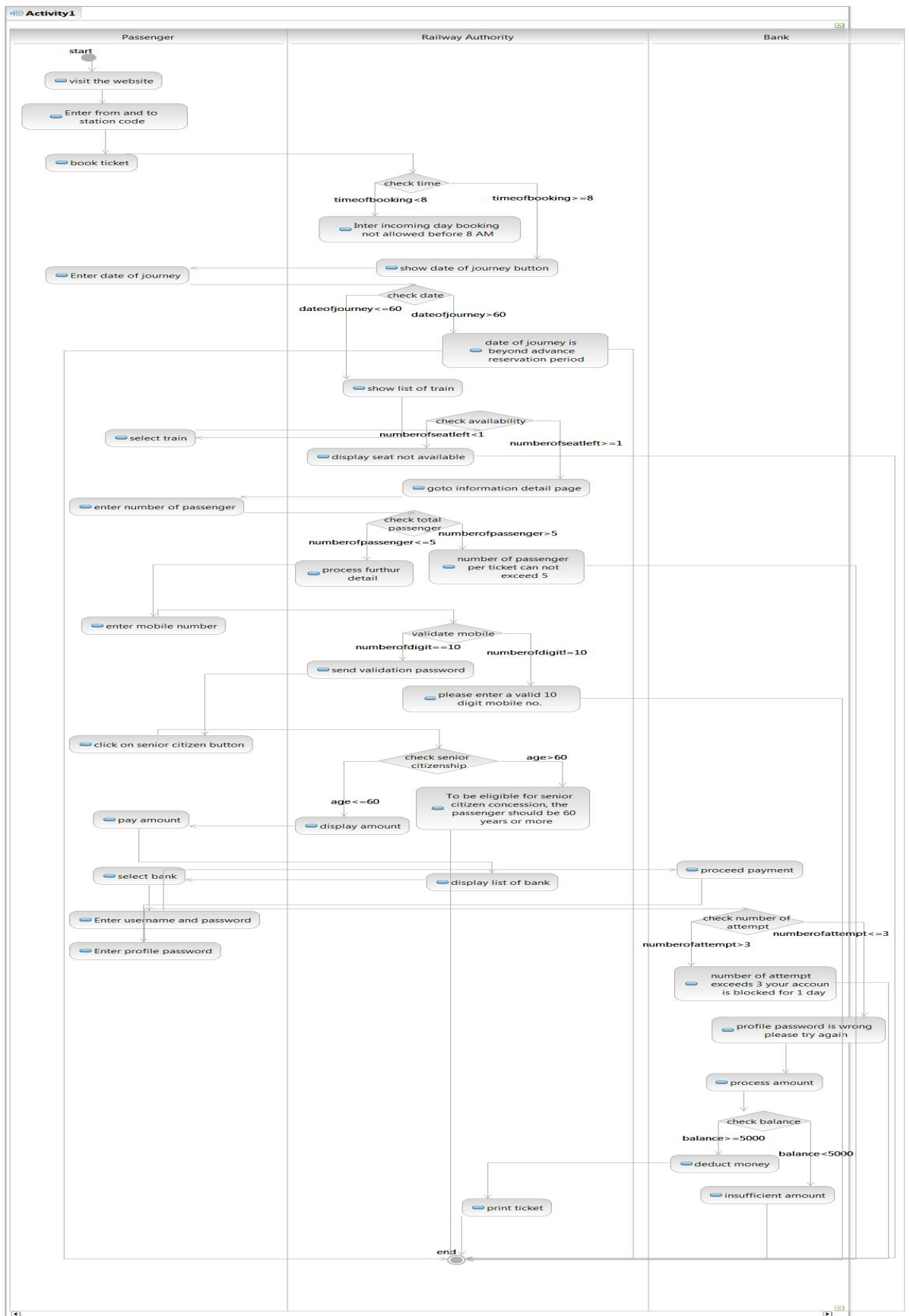


Figure 4.3: Activity Diagram of Railway Reservation System

The exported XMI code of constructed diagrams is given as input to our parser *CADExtractor* show in Figure 5.5, that extract the necessary information and given as input to Algorithm 1.

4.5.1 Working of algorithm

The Algorithm 1 begin by reading the element of decisionarray in sequential order. First element is the predicate `timeofbooking<8`. leftoperand `timeofbooking` is assigned to variable `testcondition`, rightoperand `8` is assigned to variable `testinput` and the operator `<` is assigned to `operatorsymbol`. Now algorithm check whether `timeofbooking` is present in `attributearray` or not. since it is present the if condition is satisfied and it check for type of operator. since the operator symbol is `<` it matches the appropriate `if...elseif...else` statement and output is printed as (1, `timeofbooking`, 7, Inter incoming day booking not allowed befor 8 AM). The value of variable `branchcovered` and `testcaseid` is increment by 1.

The algorithm then proceed to next element which is found to be `timeofbooking>=8`. leftoperand `timeofbooking` is assigned to variable `testcondition`, rightoperand `8` is assigned to variable `testinput` and the operator `>=` is assigned to `operatorsymbol`. Now algorithm check whether `timeofbooking` is present in `attributearray` or not. since it is present the if condition is satisfied and it check for type of operator. since the operator symbol is `>=` it matches the appropriate `if...elseif...else` statement. Output printed is (2, `timeofbooking`, 8, show date of journey button) and (2, `timeofbooking`, 9, show date of journey button). The value of variable `branchcovered` is increment by 1 and `testcaseid` is incremented by 2.

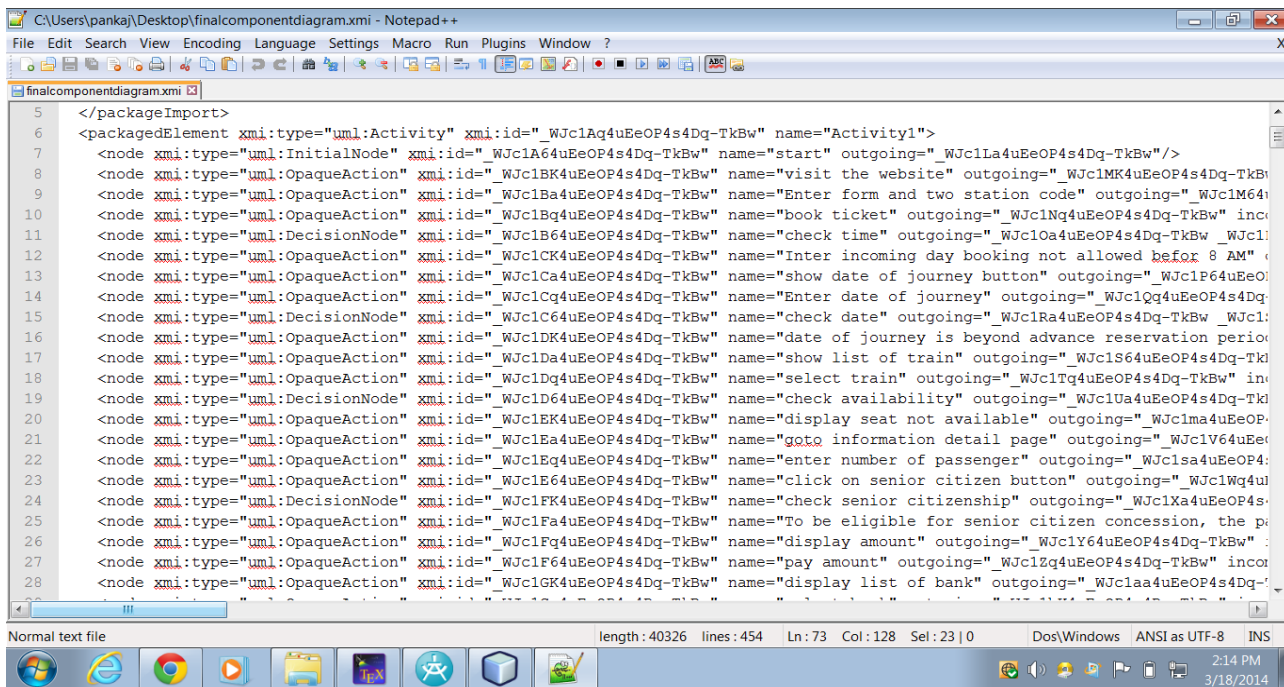
Above process is repeated until the `decisionarray` becomes empty. Overall test cases generated is shown in table 4.2 and the value of `branchcovered` becomes 16. Next, decision coverage is calculated in following way: since there is only one condition in decision node, there is only two possible outcome one is true and other is false.

number of decision node are: 8

number of decision outcome: $8*2$

number of decision outcome excersized: 16

Chapter 4 Test Case generation from combination of class and activity diagrams



```
5 </packageImport>
6 <packageElement xmi:type="uml:Activity" xmi:id="_WJc1Aq4uEeOP4s4Dq-TkBw" name="Activity1">
7 <node xmi:type="uml:InitialNode" xmi:id="_WJc1A64uEeOP4s4Dq-TkBw" name="start" outgoing="_WJc1La4uEeOP4s4Dq-TkBw"/>
8 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1BK4uEeOP4s4Dq-TkBw" name="visit the website" outgoing="_WJc1MK4uEeOP4s4Dq-TkBw"/>
9 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Ba4uEeOP4s4Dq-TkBw" name="Enter form and two station code" outgoing="_WJc1M64uEeOP4s4Dq-TkBw"/>
10 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Bq4uEeOP4s4Dq-TkBw" name="book ticket" outgoing="_WJc1Nq4uEeOP4s4Dq-TkBw" incoming="_WJc1M64uEeOP4s4Dq-TkBw"/>
11 <node xmi:type="uml:DecisionNode" xmi:id="_WJc1B64uEeOP4s4Dq-TkBw" name="check time" outgoing="_WJc1Oa4uEeOP4s4Dq-TkBw" outgoing_label=">
12 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1CK4uEeOP4s4Dq-TkBw" name="Inter incoming day booking not allowed before 8 AM" outgoing="_WJc1P64uEeOP4s4Dq-TkBw"/>
13 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Ca4uEeOP4s4Dq-TkBw" name="show date of journey button" outgoing="_WJc1P64uEeOP4s4Dq-TkBw"/>
14 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Cq4uEeOP4s4Dq-TkBw" name="Enter date of journey" outgoing="_WJc1Qq4uEeOP4s4Dq-TkBw"/>
15 <node xmi:type="uml:DecisionNode" xmi:id="_WJc1C64uEeOP4s4Dq-TkBw" name="check date" outgoing="_WJc1Ra4uEeOP4s4Dq-TkBw" outgoing_label=">
16 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1DK4uEeOP4s4Dq-TkBw" name="date of journey is beyond advance reservation period" outgoing="_WJc1S64uEeOP4s4Dq-TkBw"/>
17 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Da4uEeOP4s4Dq-TkBw" name="show list of train" outgoing="_WJc1S64uEeOP4s4Dq-TkBw"/>
18 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Dq4uEeOP4s4Dq-TkBw" name="select train" outgoing="_WJc1Tq4uEeOP4s4Dq-TkBw" incoming="_WJc1S64uEeOP4s4Dq-TkBw"/>
19 <node xmi:type="uml:DecisionNode" xmi:id="_WJc1D64uEeOP4s4Dq-TkBw" name="check availability" outgoing="_WJc1Ua4uEeOP4s4Dq-TkBw" outgoing_label=">
20 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1EK4uEeOP4s4Dq-TkBw" name="display seat not available" outgoing="_WJc1ma4uEeOP4s4Dq-TkBw"/>
21 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Ea4uEeOP4s4Dq-TkBw" name="goto information detail page" outgoing="_WJc1V64uEeOP4s4Dq-TkBw"/>
22 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Eq4uEeOP4s4Dq-TkBw" name="enter number of passenger" outgoing="_WJc1sa4uEeOP4s4Dq-TkBw"/>
23 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1E64uEeOP4s4Dq-TkBw" name="click on senior citizen button" outgoing="_WJc1Wq4uEeOP4s4Dq-TkBw"/>
24 <node xmi:type="uml:DecisionNode" xmi:id="_WJc1FK4uEeOP4s4Dq-TkBw" name="check senior citizenship" outgoing="_WJc1Xa4uEeOP4s4Dq-TkBw"/>
25 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Fa4uEeOP4s4Dq-TkBw" name="To be eligible for senior citizen concession, the passenger must be 60 years of age or above" outgoing="_WJc1Y64uEeOP4s4Dq-TkBw"/>
26 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1Fq4uEeOP4s4Dq-TkBw" name="display amount" outgoing="_WJc1Y64uEeOP4s4Dq-TkBw" outgoing_label=">
27 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1F64uEeOP4s4Dq-TkBw" name="pay amount" outgoing="_WJc1Zq4uEeOP4s4Dq-TkBw" incoming="_WJc1Y64uEeOP4s4Dq-TkBw"/>
28 <node xmi:type="uml:OpaqueAction" xmi:id="_WJc1GK4uEeOP4s4Dq-TkBw" name="display list of bank" outgoing="_WJc1aa4uEeOP4s4Dq-TkBw" incoming="_WJc1Zq4uEeOP4s4Dq-TkBw"/>
```

Figure 4.4: XMI code of Railway Reservation System

branch coverage = $\frac{\text{number of decision outcome exercised}}{\text{number of decision outcomes}} \times 100$

branch coverage = 100%

Table 4.2: Table showing test input with expected output

testcaseid	testcondition	input	expected output
1	timeofbooking	7	Inter incoming day booking not allowed before 8 AM
2	timeofbooking	8	show date of journey button
3	timeofbooking	9	show date of journey button
4	dateofjourney	61	date of journey is beyond advance reservation period
5	dateofjourney	60	show list of train
6	dateofjourney	59	show list of train
7	numberofseatleft	0	display seat not available
8	numberofseatleft	1	goto information detail page
9	numberofseatleft	2	goto information detail page
10	age	61	To be eligible for senior citizen concession, the passenger should be 60 years or more
11	age	60	display amount
12	age	59	display amount
13	numberofattempt	4	number of attempt exceeds 3 your account is blocked for 1 day
14	numberofattempt	3	profile password is wrong please try again
15	numberofattempt	2	profile password is wrong please try again
16	balance	5000	deduct money
17	balance	5001	deduct money
18	balance	4999	insufficient amount
19	numberofdigit	9	please enter a valid 10 digit mobile no.
20	numberofdigit	11	please enter a valid 10 digit mobile no.
21	numberofdigit	10	send validation password
22	numberofpassenger	6	number of passenger per ticket can not exceed 5
23	numberofpassenger	5	process furthur detail
24	numberofpassenger	4	process furthur detail

4.5.2 Analysis of mutation coverage

Table 4.3 show the result of actual operator and its mutants. The actual operator is `timeofbooking<8` and its mutants are `timeofbooking>8`, `timeofbooking<=8`, `timeofbooking>=8`, `timeofbooking==8`, `timeofbooking!=8`. Similarly, for predicate `numberofdigit==8` the outcome of different possible mutant is shown in Table 4.4.

From the table 4.3 and 4.4 we can conclude that mutants are giving different output in at least one case. Hence, the test cases obtained is able to detect all the possible mutant.

Table 4.3: Outcome of different possible relational operator mutant

testcaseid	actual operator	mutant	mutant	mutant	mutant	mutant
	<i>timeofbooking</i> < 8	<i>timeofbooking</i> > 8	<i>timeofbooking</i> ≤ 8	<i>timeofbooking</i> ≥ 8	<i>timeofbooking</i> = 8	<i>timeofbooking</i> ≠ 8
1	T	F	T	F	F	T
2	F	F	T	T	T	F
3	F	T	F	T	F	T

Table 4.4: Outcome of different possible mutant

testcaseid	actual operator	mutant	mutant	mutant	mutant	mutant
	<i>numberofdigit</i> = 10	<i>numberofdigit</i> > 10	<i>numberofdigit</i> ≤ 10	<i>numberofdigit</i> ≥ 10	<i>numberofdigit</i> < 10	<i>numberofdigit</i> ≠ 10
19	F	F	T	F	T	T
20	F	T	F	T	F	T
21	T	F	T	T	F	F

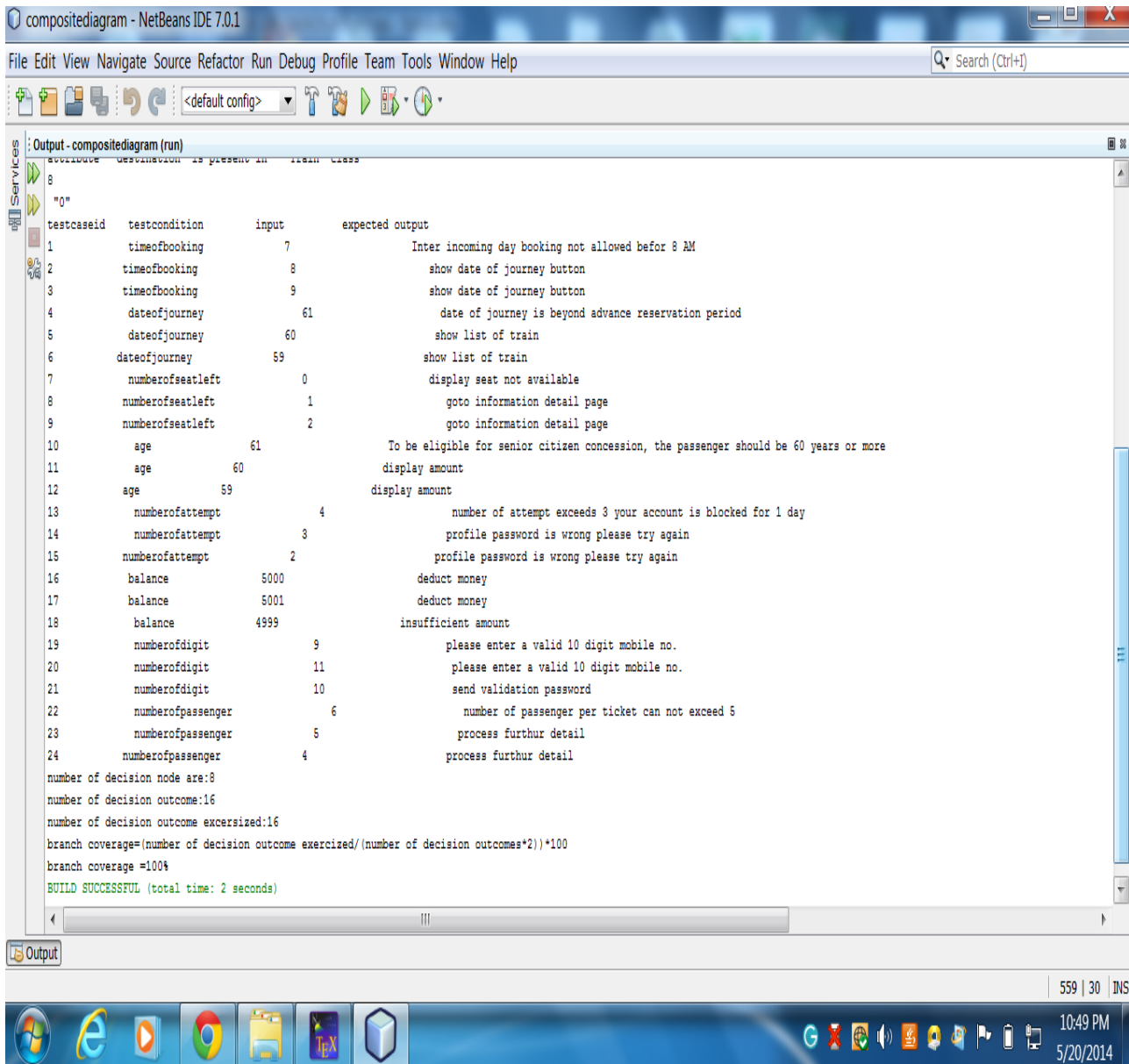


Figure 4.5: Screenshot of generated test cases

Chapter 5

Test Scenario Generation from UML Composite Structure Diagram

Unit testing is the first step in the software testing process. Unit testing ensures that each component is developed correctly [15]. In software testing, most of the bugs go unidentified even after unit testing is performed successfully. When the components are integrated, they may not perform as per the requirement, due to badly designed interface. So efficient test scenario needs to be generated in order to identify the bugs when two or more components are combined together. Once unit testing is done, integration testing is performed to find errors in component interface when they interact with each other. The Unified Modeling Language (UML) composite structure diagram is a suitable diagram for describing the interactions between system components.

The components are written in different programming languages, and executes on various platforms [25]. Components interact with each other by passing messages [26]. The component performs correctly when tested individually. But when integrated with new component unexpected result may occur [25]. Testing methodology is categorized into two types: black box and white box. Black box testing is testing the software based on user requirements, without any knowledge of the internal structure of program [27]. White box testing requires the knowledge of the internal structure of the component [28]. In component-based software implementation of the component is not available [25]. Due to this it is difficult

to apply white-box techniques to test component-based software.

5.1 Basic Concepts and Definitions

In this section, we discuss some of basic terminology which are required to understand our work.

5.1.1 Composite structure diagram

Composite structure diagram visualizes the internal structure of a component, including the interaction between component [29]. Composite structure diagram is a kind of the component diagram used in modeling a system. Some of the symbols present in composite structure diagram is show in Figure 5.1.

1. **Component:** A Component is an independent piece of code that provides access to the services through some dedicated interfaces [30].
2. **Port:** Port represents a group of messages or operation calls that pass either into or out of a component.
3. **Interface:** It provides a medium through which components interact with each other.
4. **Provided interface:** A component with a provided interface port supplies services that it implements to other components requiring these services.
5. **Required interface:** A component with a required interface port receive services that are implemented by other components.

Test Criteria: In Top-Down integration testing, at-most $(n-1)$ stubs are created, where n is the number of nodes [31]. In Bottom-Up integration testing, at-most $(n-l)$ drivers are created, where n is the total number of nodes and l is the number of leaf nodes [31].


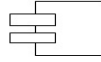



Symbols	Names
	Port
	Component
	Interface
	Provided Interface
	Required Interface

Figure 5.1: Basic Symbols of Composite Structure Diagram

Composite Structure Graph (CSG): Composite structure graph $G (V,E)$ is a set of nodes connected by edges. V is the set of nodes representing components and E is the set of directed edges joining the calling and called vertices.

5.2 Proposed Approach

In this section, we discuss our proposed approach to generate test scenarios from composite structure diagram. The block diagram of our proposed approach is shown in Figure 5.2.

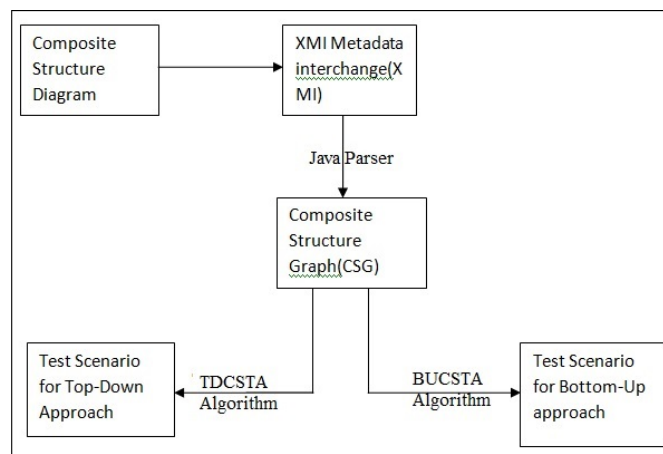


Figure 5.2: Block diagram of proposed approach

Initially the Composite Structure diagram is constructed using IBM Rational Software Architect (RSA). Diagram is then exported to XMI code. We develop a parser called *CSDExtractor* shown in Figure 5.5. The *CSDExtractor* extract the necessary information such as component name, dependency between components

etc from the XMI code. Based on the extracted information composite structure graph (CSG) is generated. Gvedit [32] tool is used to visualize the graph. Each component in composite structure diagram is represented as a node in the CSG. The component name in composite structure diagram corresponds to node name in CSG. The required and provided interface are represented as directed edges from one node to another. In each pair of node there is one *callee* node and another *called* node. *Callee* node represents the component which requires services and the *called* node represents the component which provides services. Root node represents the component which does not provide services and leaf nodes represents these components which does not require services in CSG. To implement our proposed work, we developed two algorithms named Top-Down test scenario generation algorithm (TDTSGA) and Bottom-Up test scenario generation algorithm (BUTSGA) given in Algorithm 2 and Algorithm 3, respectively. We have modified breath first search to TDTSGA, as we can not directly apply breath first search in Top-Down approach. We discuss the details of implementation of our proposed approach in Section 5.2.1.

5.2.1 Proposed Algorithm

This section, we present our proposed algorithm to generate test scenarios, in pseudo code form.

Algorithm 2 Top-Down Test Scenario Generation Algorithm (TDTSGA)

Input: Composite Structure Graph(CSG)

Output: Set of test scenario.

```
1: Stack S= $\phi$ 
2: Queue Q = $\phi$ 
3: x= $\phi$ 
4: testscenario= $\phi$ 
5: Enque(Q,root)
6: repeat
7:   x=Deque(Q)
8:   push(S,x)
9:   for i= 1 to n do
10:     Enque(Q, $y_i$ )
11:   end for
12:   testscenarioj = elements of Stack S  $\cup$  elements of queue Q
13: until Q is not empty
14: MakeEmpty( S )
15: Exit
```

5.2.2 Description of Algorithm

TDTSGA takes CSG as its input and generates a set of test scenarios as its output. Algorithm 2 maintains a queue Q, a stack S and a variable *testscenario*. At the beginning of algorithm 2, the stack, queue and *testscenario* variable are initialized to null. First, the root node of CSG is inserted into Q. Next, the element is deleted from Q and is pushed into S. All the neighboring nodes of the deleted node are inserted into Q. Now the content of S and Q are assigned to variable *testscenario*. The *testscenario* indicates that the components present in S are to be tested by integrating it with stub components present in Q. For the termination of the algorithm, it checks whether Q is empty or not. If it is not empty, an element is deleted from the queue and the process is repeated otherwise S is emptied.

Algorithm 3 Bottom-Up Test Scenario Generation Algorithm(BUTSGA)

Input: Composite Structure Graph(CSG)

Output: Set of test scenario.

```

1: Stack S1 =  $\phi$ 
2: Stack S2 =  $\phi$ 
3: Stack S3 =  $\phi$ 
4: testscenario =  $\phi$ 
5: x =  $\phi$ 
6: y =  $\phi$ 
7: for i= 1 to n do
8:   Push(S1, xi ) ;
9: end for
10: repeat
11:   y = Pop(S1)
12:   for For i= 1 to m do
13:     Push(S2, yi)
14:     Push(S3, yi)
15:   end for
16:   repeat
17:     z = Pop(S3)
18:
19:     for For i= 1 to p do
20:       Push(S2, zi)
21:     end for
22:   until S3 is not empty
23:   if S1 is empty then
24:     testscenarioj = elements of Stack S2  $\cup$  Y
25:   else
26:     testscenarioj = elements of Stack S2  $\cup$  Y take as driver
27:   end if
28:   MakeEmpty(S2)
29: until S1 is not empty
30: Exit

```

BUTSGA algorithm is applied to generate test scenarios for Bottom-Up integration testing. BUTSGA takes CSG as its input and generates a set of test scenarios as its output. Algorithm 3 maintains three stacks: S1, S2 and S3 and

three variable x , y and $testscenario$. At the beginning of the algorithm 3, all the stacks and variables are initialized to null. This algorithm first traverses all the non-leaf nodes using Breath First Search (BFS) and pushes all the visited nodes into $S1$. An element is popped from $S1$ and is assigned to variable y . All the neighboring nodes of the deleted node are pushed into $S2$, and stack $S3$. An element is popped from $S3$ and is assigned to variable z . All the neighboring nodes of deleted node are pushed into $S2$. All the neighboring nodes of the element present in the $S3$ are then pushed into stack $S2$ until $S3$ is empty. Now the content of $S2$ and y are assigned to *testscenario*. The *testscenario* indicates that the component present in $S2$ are to be tested by integrating it with driver component present in y . The contents of stack $S2$ are deleted and are checked whether the stack $S1$ is empty or not. If it is empty, then algorithm 3 terminates else the above procedure is repeated again.

5.3 Case Study

We consider the example of *railway reservation system* to discuss our proposed approach. We model the composite structure diagram of railway reservation system in RSA which is shown in Figure 5.3. We consider 12 components named as *humaninterface*, *loginpage*, *signuppage*, *homepage*, *planmyticket*, *bookedhistory*, *cancellation*, *bankgateways*, *getpnrstatus*, *goforcancellation*, *getsms* and *printticket*. We have considered the 11 interfaces named as *interface1*, *interface2*, *interface3*, *interface4*, *interface5*, *interface6*, *interface7*, *interface8*, *interface9*, *interface10*, *interface11*. we consid 22 ports named as *port1*, *port2*, *port3*, *port4*, *port5*, *port6*, *port7*, *port8*, *port9*, *port10*, *port11*, *port12*, *port13*, *port14*, *port15*, *port16*, *port17*, *port18*, *port19*, *port20*, *port21*, *port22*. The ports are used to connect the interfaces of two components. The two components are connected via interface. There are two type of interfaces: required interface and provider interface. The component $c1$ which require services is connected through the required interface to the provided interface of the components that provides these required services. In Figure 5.3, *humaninterface* and *loginpage* interact with each other via *interface1*. When

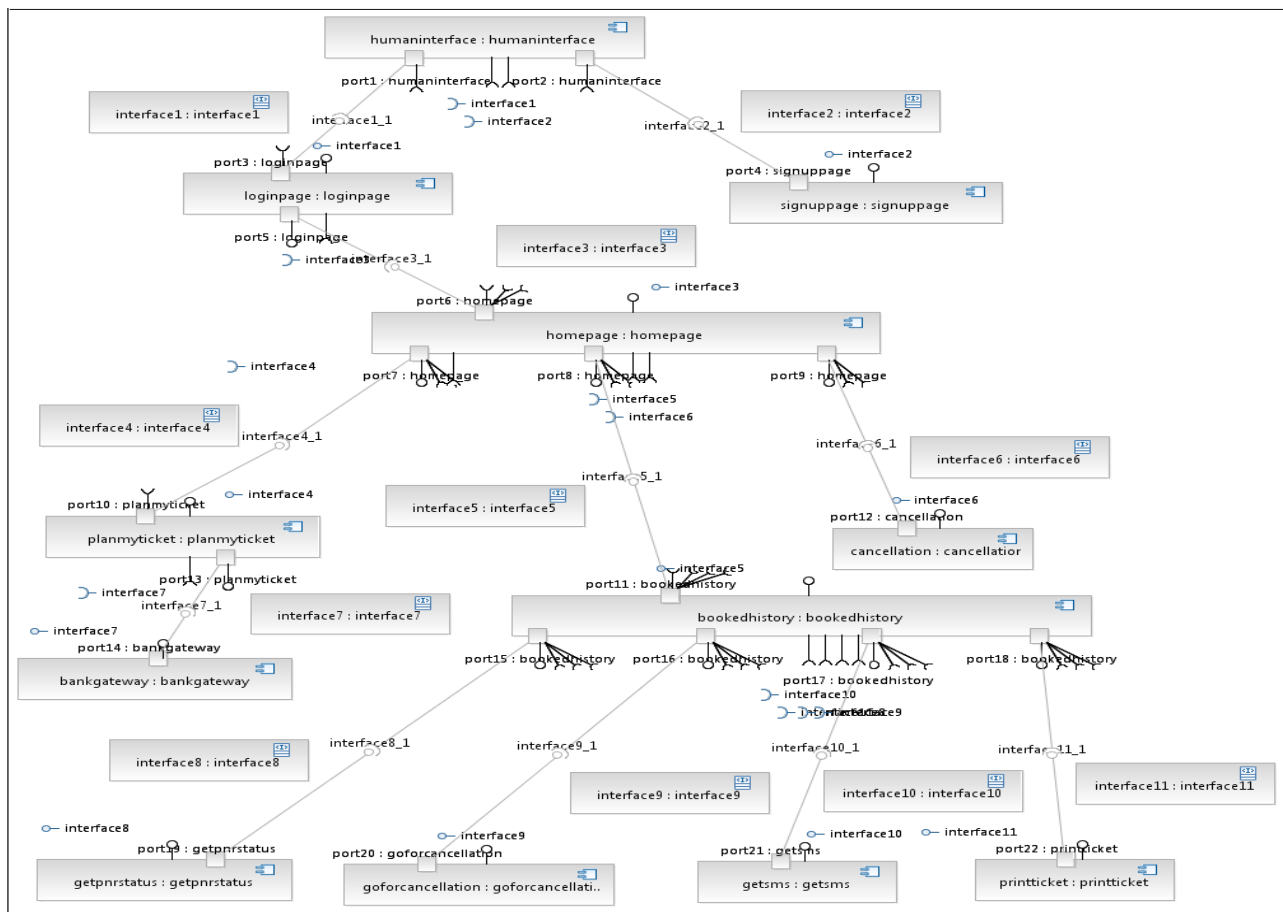


Figure 5.3: Composite Structure Diagram of Railway Reservation System

we connect the required and provider interface of *humaninterface* and *loginpage*, then *interface1_1* is created automatically. The interaction between *humaninterface* and *signuppage* occurs via *interface2*. Similarly, *interface2_1* is created, when we connect the required and provider interface of *humaninterface* and *signuppage*, respectively. Similarly, all the components are connected to each other via required and provided interface.

The exported XMI code of composite structure diagram is given as input to our parser *CSDExtractor* show in Figure 5.5, that results in composite structure graph (CSG) as shown in Figure 5.6. The CSG has 12 nodes and 7 leaf nodes. As discussed in Section 2.5, testing criteria requires 11 stubs to be created for Top-down integration approach and 5 drivers need for Bottom-Up integration approach.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <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"
3 <packageImport xmi:type="uml:PackageImport" xmi:id="_Nu48wX0IEeOdk6Zgeo0KmQ">
4 <importedPackage xmi:type="uml:Model" href="http://schema.omg.org/spec/UML/2.1/uml.xml#_0"/>
5 </packageImport>
6 <packagedElement xmi:type="uml:Class" xmi:id="_Nu48wn0IEeOdk6Zgeo0KmQ" name="railwayreservation">
7 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48w30IEeOdk6Zgeo0KmQ" name="interface1" visibility="private" type="_Nu5j-
8 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48xH0IEeOdk6Zgeo0KmQ" name="humaninterface" visibility="private" type="_N
9 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48xX0IEeOdk6Zgeo0KmQ" name="loginpage" visibility="private" type="_Nu5j_3
10 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48xn0IEeOdk6Zgeo0KmQ" name="signuppage" visibility="private" type="_Nu5kB
11 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48x30IEeOdk6Zgeo0KmQ" name="interface2" visibility="private" type="_Nu5k
12 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48yH0IEeOdk6Zgeo0KmQ" name="homepage" visibility="private" type="_Nu5kCX0
13 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48yX0IEeOdk6Zgeo0KmQ" name="interface3" visibility="private" type="_Nu5kD
14 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48yn0IEeOdk6Zgeo0KmQ" name="planmyticket" visibility="private" type="_Nu5
15 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48y30IEeOdk6Zgeo0KmQ" name="bookedhistory" visibility="private" type="_Nu
16 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48zH0IEeOdk6Zgeo0KmQ" name="cancellation" visibility="private" type="_Nu5
17 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48zX0IEeOdk6Zgeo0KmQ" name="bankgateway" visibility="private" type="_Nu5k
18 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48zn0IEeOdk6Zgeo0KmQ" name="getpnrstatus" visibility="private" type="_Nu5
19 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu48z30IEeOdk6Zgeo0KmQ" name="goforcancellation" visibility="private" type=
20 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu5j0H0IEeOdk6Zgeo0KmQ" name="getsms" visibility="private" type="_Nu5kKH0IE
21 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu5j0X0IEeOdk6Zgeo0KmQ" name="printticket" visibility="private" type="_Nu5k
22 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu5j0n0IEeOdk6Zgeo0KmQ" name="interface4" visibility="private" type="_Nu5kL
23 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu5j030IEeOdk6Zgeo0KmQ" name="interface5" visibility="private" type="_Nu5kM
24 <ownedAttribute xmi:type="uml:Property" xmi:id="_Nu5j1H0IEeOdk6Zgeo0KmQ" name="interface6" visibility="private" type="_Nu5kM

```

Figure 5.4: XMI code of Railway Reservation System

5.3.1 Working of Algorithm

Test scenario generation process starts with the root node of CSG. Stubs are created for each component which are called by root component. Thus, the generated test scenarios at the first level of integration are: $testscenario_1 = \{humaninterface\}$ stub for $\{loginpage, signuppage\}$. In the second level, replace one of the stub with actual component and integrate it with stubs of the next level. Second test scenario, $testscenario_2 = \{humaninterface, loginpage\}$ stub for $\{signuppage, homepage\}$. Suppose *humaninteface* component worked correctly during the generation of first test scenario, but on integrated with *signuppage* it does not work correctly. Then, we can ensure that error may be present either in *interface* of *humaninteface* component or in *interface* of *signuppage* component. This process of test scenario generation continues till all the components are integrated. In all total 11 numbers of stubs are created during the process: *loginpage*, *signuppage*, *homepage*, *planmyticket*, *bookedhistory*, *cancellation*, *bankgateway*, *printticket*, *getsms*, *goforcancellation*, *getpnrstatus*. After executing the TDTSGA algorithm we get

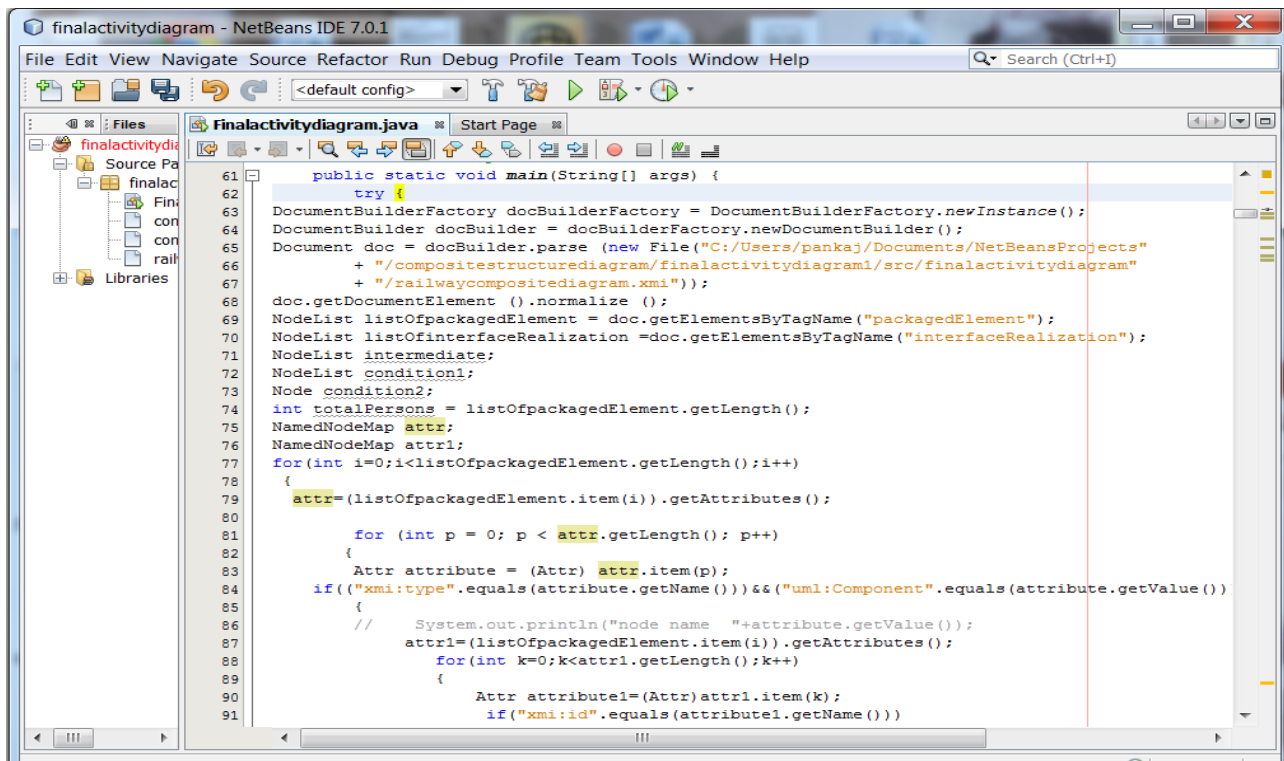


Figure 5.5: Java code for CSDExtractor

the following results for Top-Down integration:

1. testscenario₁ = {humaninterface} stub for {loginpage, signuppagement}
2. testscenario₂ = {humaninterface, loginpage} stub for {signuppagement, homepage}
3. testscenario₃ = {humaninterface, loginpage, signuppagement} stub for {homepage}
4. testscenario₄ = {humaninterface, loginpage, signuppagement, homepage} stub for {planmyticket, bookedhistory, cancellation}
5. testscenario₅ = {humaninterface, loginpage, signuppagement, homepage, planmyticket} stub for {bookedhistory, cancellation, bankgateway}
6. testscenario₆ = {humaninterface, loginpage, signuppagement, homepage, planmyticket, bookedhistory} stub for {cancellation, bankgateway, printticket, getsms, goforcancellation, getpnrstatus}

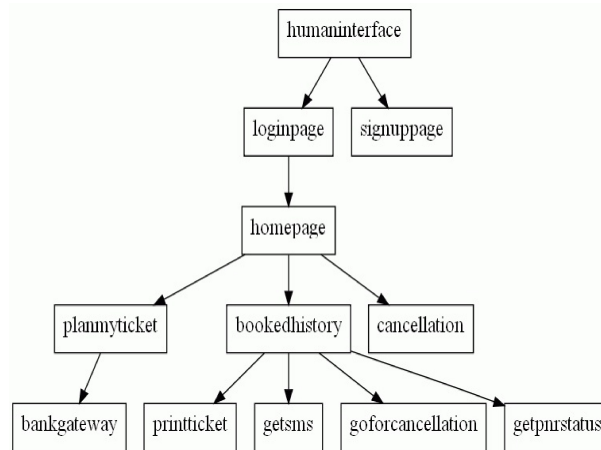


Figure 5.6: Composite Structure Graph of Composite Structure Diagram in Figure 5.3.

7. $\text{testscenario}_7 = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}\}$ stub for $\{\text{bankgateway}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}\}$
8. $\text{testscenario}_8 = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{bankgateway}\}$ stub for $\{\text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}\}$
9. $\text{testscenario}_9 = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{bankgateway}, \text{printticket}\}$ stub for $\{\text{getsms}, \text{goforcancellation}, \text{getpnrstatus}\}$
10. $\text{testscenario}_{10} = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{bankgateway}, \text{printticket}, \text{getsms}\}$ stub for $\{\text{goforcancellation}, \text{getpnrstatus}\}$
11. $\text{testscenario}_{11} = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{bankgateway}, \text{printticket}, \text{getsms}, \text{goforcancellation}\}$ stub for $\{\text{getpnrstatus}\}$
12. $\text{testscenario}_{12} = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{bankgateway}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}\}$

This result contains all the possible test scenarios that are generated by Algorithm 2 based on the given CSG as shown in Figure 5.6.

Bottom-Up integration starts with the leaf node components of the composite structure graph shown in Figure 5.6. For each set of leaf nodes, a driver is created as the parent node of these leaf nodes. Thus, the generated test scenario at first iteration of integration are $\text{testscenario}_1 = \{\text{bankgateway}, \text{driver-planmyticket}\}$. When *bankgateway* is integrated with *planmyticket*, if it does not perform correctly as from first test scenario, then we can assume that, error may be likely to be present in either *bankgateway* or *planmyticket* interface. In the second iteration, replace the driver with actual component and integrate it with driver of next level. This process of test scenarios generation continues till all the components are integrated. In all total 5 drivers are created: *bookedhistory*, *planmyticket*, *homepage*, *loginpage*, *humaninterface*.

After executing the algorithm BUTSGA, we get the following results for Bottom-Up integration:

1. $\text{testscenario}_1 = \{\text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}\}$ driver for $\{\text{bookedhistory}\}$
2. $\text{testscenario}_2 = \{\text{bankgateway}\}$ driver for $\{\text{planmyticket}\}$
3. $\text{testscenario}_3 = \{\text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}, \text{bankgateway}\}$ driver for $\{\text{homepage}\}$
4. $\text{testscenario}_4 = \{\text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}, \text{bankgateway}\}$ driver for $\{\text{loginpage}\}$
5. $\text{testscenario}_5 = \{\text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}, \text{bankgateway}\}$ driver for $\{\text{humaninterface}\}$
6. $\text{testscenario}_6 = \{\text{humaninterface}, \text{loginpage}, \text{signuppage}, \text{homepage}, \text{planmyticket}, \text{bookedhistory}, \text{cancellation}, \text{printticket}, \text{getsms}, \text{goforcancellation}, \text{getpnrstatus}, \text{bankgateway}\}$

Chapter 5 Test Scenario Generation from UML Composite Structure Diagram

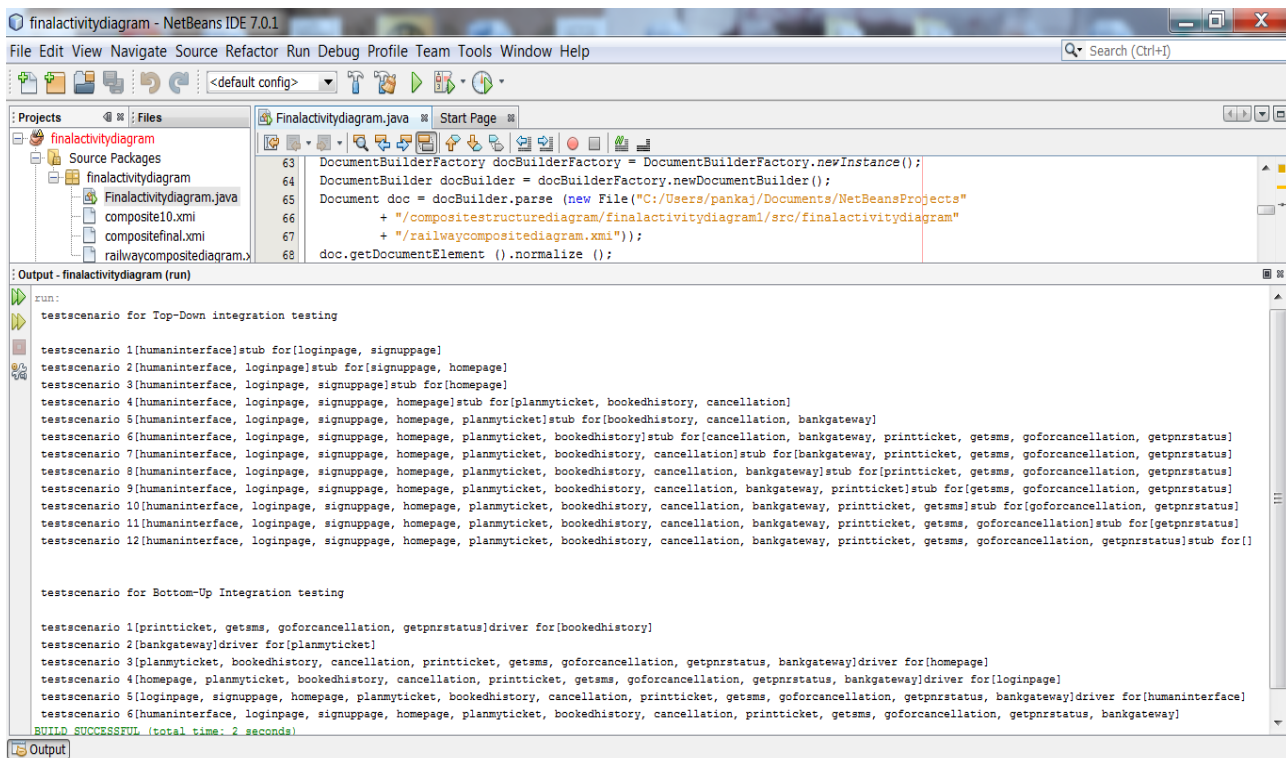


Figure 5.7: Test Scenarios generated for Top-Down and Bottom-Up Integration Testing

Figure 5.7 shows the result obtained in both Top-Down and Bottom-Up Integration Testing.

Chapter 6

Conclusion and Future Work

The major aim of our research was to automate the test cases generation from UML diagrams. Below, we summarize the important contributions of our work. At the end, some suggestions for future work are given.

6.1 Test Case generation from combination of class and activity diagrams

In this paper, we discussed a methodology to automatically generate test scenarios from the combination of class diagram and activity diagram. The proposed methodology is completely model-based and suitable for mutation testing. We developed a parser (CADExtractor) to generate the test cases automatically. We implemented the proposed algorithm Integrated test case generation algorithm (ITCGA). We have achieved the 100% branch coverage and the generated test cases are suitable for unit testing and mutation testing. The test cases are able to detect all the relational operators mutant. The generated test cases are not redundant. In the future, we further will generalize the approach by considering float, string data type.

6.2 Test Scenario Generation from UML Composite Structure Diagram

In this paper, we discussed a methodology to automatically generate test scenarios from UML composite structure diagram. The proposed methodology is completely model-based and suitable for integration testing. We developed a parser (CSDExtractor) to generate the composite structure graph automatically from input composite structure diagram. We implemented two proposed algorithm, Top-Down Test Scenario Generation Algorithm (TDTSGA) and Bottom-Up Test Scenario generation algorithm (BUTSGA) to generate test scenarios for Top-Down and Bottom-Up Integration. The generated test scenarios are sufficient enough to find the component in which probability of bug presence is maximum. In future, we plan to use coupling measure to detect the fault prone components.

In most of the work generated test cases cannot be directly fed into the system under test (SUT). So, our next work is to propose a methodology to feed the test cases directly into the system under test (SUT).

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] 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.
- [7] 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.
- [8] 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.
- [9] P. C. Jorgensen and C. Erickson, “Object-oriented integration testing,” *Communications of the ACM*, vol. 37, no. 9, pp. 30–38, 1994.
- [10] W. E. Howden, “Weak mutation testing and completeness of test sets,” *Software Engineering, IEEE Transactions on*, no. 4, pp. 371–379, 1982.
- [11] Y.-S. Ma and J. Offutt, “Description of class mutation operators for java,” 2005.
- [12] J. Kovse and T. Harder, “Generic xmi-based uml model transformations,” in *Object-Oriented Information Systems*, pp. 192–198, Springer, 2002.
- [13] B. Demuth, H. Hussmann, and S. Obermaier, “Experiments with xmi based transformations of software models,” in *Workshop on Transformations in UML*, 2001.
- [14] 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.
- [15] J. Hartmann, C. Imoberdorf, and M. Meisinger, “Uml-based integration testing,” in *ACM SIGSOFT Software Engineering Notes*, vol. 25, pp. 60–70, ACM, 2000.
- [16] 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.
- [17] 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.

- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.

-
- [26] 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.
- [27] 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.
- [28] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, “Automated whitebox fuzz testing,” in *NDSS*, vol. 8, pp. 151–166, 2008.
- [29] T. Sekulin, *Implementing a business process into an ERP solution*. PhD thesis, uniwien, 2008.
- [30] Y. Wu and J. Offutt, “Maintaining evolving component-based software with uml,” in *CSMR*, pp. 133–142, 2003.
- [31] N. Chauhan, *Software Testing: Principles and Practices*. Oxford University Press, 2010.
- [32] <http://www.graphviz.org/Download..php>.
- [33] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide*. Pearson Education India, 1999.