INTERACTION BASED SOFTWARE TESTING

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

Master of Engineering
In
Software Engineering

By:
Rohit Kumar
(800831021)

Under the supervision of:
Dr. Rajesh Kumar Bhatia
Assistant Professor and Head
CSED, TU, Patiala

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 14700

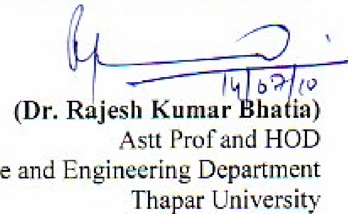
Thapar University, Patiala
JULY 2010

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Interaction Based Test Case Generation"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Rajesh Kumar Bhatia and refers other researcher's works which are duly listed in the reference section. The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

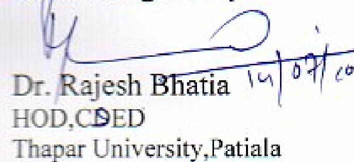
(Rohit Kumar)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

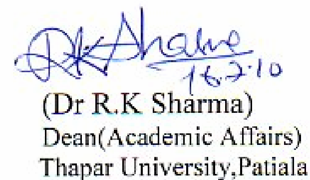


(Dr. Rajesh Kumar Bhatia)
Astt Prof and HOD
Computer Science and Engineering Department
Thapar University

Countersigned by-



Dr. Rajesh Bhatia
HOD,CSED
Thapar University,Patiala



(Dr R.K Sharma)
Dean(Academic Affairs)
Thapar University,Patiala

Acknowledgment

I wish to express my deep gratitude to Dr. Rajesh K. Bhatia, Assistant Professor and Head of CSED, Thapar University Patiala for providing his uncanny guidance and support throughout the thesis.
No words of thanks are enough for my family whose support and care makes me stay on earth. Thanks to be with me every time and everywhere.

I am thankful to Dr. Rajesh Kumar Bhatia, Head, Computer Science & Engineering Department, TU, Patiala for the motivation and inspiration that triggered me for the thesis work. I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.

Last but not the least, I express my heartfelt thanks my friends and well-wishers for co-operation, which they were always ready to extend.

Rohit Kumar
800831021
M.E. (Software Engineering)-2nd year
Computer Science & Engineering Department
Thapar University
Patiala -147004

Abstract

Testing is defined as a process of executing a program or system with the intent of finding errors. Testing consists of validating and verifying the software artifacts so that they meet their business and technical requirements. It is a very important and time-consuming part of software development life cycle. But most of the software fails because of not being tested properly. The reason behind failure mainly includes errors present in analysis and design phase.

In proposed work, design diagrams of the software are used for test cases generation. The test case generation using design helps to plan test case early. Generating test cases from design are more effective and efficient as design is closer to White Box testing. Sometimes during test cases generation, we come to know about any incompleteness and inconsistence in requirement, which can overcome by taking necessary measures taken in time.

The proposed system focus on three UML diagrams that are Class, Statechart and Activity diagram. The petal files also called mdl file of these diagrams are parsed by the tool.. It tokenizes the line into words and then matches the string with the pattern to find the test case information. Then the text files created are entered to the Oracle database using SQL *Loader. Before loading text files to SQL *Loader it was necessary to create control file and a Dat file. The control file tells SQL*Loader where to find the data, how to parse and interpret the data and where to insert the data. Dat file which contains the data i.e. the text file from the tool is saved with extension .dat and it serves the purpose of Dat file in SQL *Loader to load the data. The text files in the form of Dat file are loaded to Oracle database. Further queries are fired from Java to extract the strings from database and use them to generate automatic test cases.

.

Table of Contents

Certificate ....Error! Bookmark not defined.
Acknowledgment...iii
Abstract...iv
Table of Contents v
Chapter 1 1
Introduction.. 1
1.1 Object Oriented System Testing 1
1.2 Model Based Software Testing 2
1.3 Why we need Model Based Testing? [5] 3
Chapter 2 4
Background Information and Literature Review 4
2.1 Type of testing strategies.. 4
2.1.1 White Box Testing 4
2.1.2 Black Box Testing... 6
2.1.3 White Box versus Black Box 9
2.2 Grey Box Testing 9
2.3 Unified Modeling Language 10
2.4 UML Based Software Testing.. 11
2.4.1 Use Case Diagram-Based Software Testing.. 12
2.4.2 Structural Diagram-Based Software Testing 12
2.4.3 Behavior Diagram Based Software Testing... 12
2.4.4 Interaction Diagram Based Testing 12
2.5 Class Diagram Based Testing 13
2.5.1 Process to Test a Class of Software System 13
2.5.3 What to Test in a Class?... 14
2.6 Activity Diagram Based Software Testing.. 15
2.7 State Diagram Based Testing of Software System.. 16
2.7.1 Piecewise Coverage 16
2.7.2 Transition Coverage... 17
2.7.3 Round-trip Path Coverage... 17
2.8 Class Diagram Based Testing Related Work 17
2.9.1 UML Class Diagram Based Testing Using Verification [19]...................................... 17

2.10 State Diagram Based Testing Related Work ... 19
2.10.1 State Diagram Testing using Flow Graph ... 19
2.10.2 State Diagram Testing using Finite State Machine ... 20
2.11 Activity Diagram Based Related Work ... 20
2.11.1 Activity Diagram Testing using Slicing ... 20
3.1 Gaps in Existing Work ... 22
3.1.1 Gap in Class Diagram's Existing Work ... 22
3.1.2 Gap in Statechart Diagram's Existing Work ... 23
3.2.3 Gap in Activity Diagram's Existing Work ... 23
3.2 Problem Formulation ... 23
3.3 Objectives of thesis. ... 25
Chapter 4 ... 26
Proposed System and Implementation ... 26
4.1 Brief Overview about System ... 26
4.2 Methodology ... 27
4.3 Description of Methodology ... 28
Chapter 5 ... 35
Testing & Results (Experimental Results) ... 35
5.1 Case study: Online Hotel Reservation System ... 35
5.2 Combine test cases from Activity and State diagram of Hotel ... 39
Reservation System ... 39
5.3 Snapshot of Tool ... 40
Chapter 6 ... 43
Conclusions and Future Scope ... 43
6.1 Conclusions ... 43
6.2 Future Work ... 44
References ... 45
List of Papers/Publications ... 48

List of Figures and Tables

Figure 2.1: The 4+1 View in UML………………………………………………………...11
Figure 4.1: The Flow Chart of Methodology…………………………………………28
Figure 4.2: The Snapshot of Petal File………………………………………………..29
Figure 4.3: Block Diagram of System………………………………………………..30
Figure 4.4: The Test Case Data from Class in the Database…………………………….31
Figure 4.5: The Test Case Data from Statechart Diagram in the Database…………… 32
Figure 4.6: The Test Case Data from Activity Diagram in the Database.…………..…...33
Figure 5.1 Class Diagram of Hotel Reservation System……………………………… 35
Figure 5.2 State Diagram of Hotel Reservation System…………………………………37
Figure 5.3 Activity Diagram of Hotel Reservation System……………………………...38
Figure 5.4: Snapshot of the Tool……………………………………………………….40
Figure 5.5: User Interface for Output…………………………………………………41
Figure 5.6: Test Cases Generation from Class Diagram………………………………....41
Figure 5.7 Test Cases Generation from Statechart Diagram and Activity Diagram…..…42
Table 2.1: The Different Phases and UML Diagrams…………………………………...13
Table 5.2: Test Cases from Class Diagram………………………………………………36
Table 5.2: Test Cases from State Diagram………………………………………………37
Table 5.3: Test Cases from Activity Diagram…………………………………………...39
Table 5.4: Combined Test Case from Activity and Statechart Diagram……………. ….39

Chapter 1
Introduction

Software testing is the practice of executing a program with the goal of finding errors. Testing is a course of trying to discover the errors in a program and used to compute the quality of developed computer software. Software testing is a series of processes intended to make sure software artifacts does what it was planned to do and that it does not do anything unintentional.

A successful testing is one that steps forward in the above track by causing the program to fail. The testing establishes some measure of confidence that a program does what it is made-up to do and does not do what it is not made-up to do and this purpose is best achieved by a diligent exploration for errors. Software testing is not merely pronouncement defects or bugs in the software, it is the totally dedicated discipline of evaluating the worth of the software. [1] Testing the product means adding value to it, which means raising the quality or reliability of the program. Raising the reliability of the system means finding and eliminating errors. Hence one should not test a the system to show that it works, rather, one should start with the assumption that the system contains errors and then test the system to find as many flaws as possible.[2]

1.1 Object Oriented System Testing

Object-oriented programming features in programming languages obviously affect some part of testing. The characteristics like inheritance and interfaces support polymorphism in which code influences objects without their correct Class being known. Testers should make sure that the code works no matter what the exact Class of objects might be. Language features that maintain and enforce data hiding can make testing difficult because operations must sometimes be added to a Class interface to support testing. The accessibility of these features can contribute to better and reusable testing software.

Not only alterations in programming languages influence testing, but also changes in the development process and changes in analysis and design. Most of object-oriented software-testing activities have counterparts in traditional processes. The differences in old and new methods of developing and testing software are much deeper as there is focus on objects instead of on functions that transform inputs to outputs. The major difference is in the way object-oriented software is designed as a set of objects that essentially model a problem and

then collaborate to affect a solution. Underlying this approach is the idea that as a solution to a problem might need to change eventually, the structure and sections of the problem itself does not change as much or as frequently. Consequently, a program whose design is structured from the problem will be more adaptable to changes later. A programmer known with the problem and its components can identify them in the system, thus making the system more maintainable. Because components are derivative of the problem, they can frequently be reused in the progress of other programs to resolve similar or associated problems, by this means improving the reusability of system components. The main advantage of this approach is that analysis models plot clearly to design models that, in turn, plot to code. One can begin testing through analysis and purify the tests prepared in analysis to tests for design. Tests for design can be refined to tests of execution. This means that a testing process can be interwoven with the development stage.

1.2 Model Based Software Testing

Testing activities which are based on models are fetching popularity. UML models represent specification documents which present the perfect basis for developing tests and developing testing situations. A test requires various specification, or a explanation or documentation of what the tested unit should be and how it must act [3]. Testing that is not based on a specification is totally useless. Even code based white box testing techniques that initially focus on the structure of the code, are based on specification. The code is used only as a basis to describe input factor settings that guide to the coverage of different code artifacts. Models are supplementary important if UML tools that support automatic test case generation are used.

In general, one can discriminate between two ways of how to use the UML in tandem with testing activities [3]

- Model-based testing – this is concerned with deriving test information out of UML models.
- Test modeling – it concentrates on how to model testing structure and test behavior with the UML.

The UML symbolizes a specification notation, and testing is the using or the applying of the concepts of a specification. A test case for a tested component, for example, comprises one or more process that will be called on the tested object, a precondition that defines restraints on the input factors for the test and determines the first state of the object, and a post condition that restrains the output of the tested operation, and defines the ending state of the object after test completion [3]. A validation act can then be done to resolve whether the test has not passed, and this is called the result of a test. The ideas of a test case are therefore a bit more composite, but the UML Testing Profile defines them sufficiently [4]. Hence, the UML voluntarily supplies everything that is necessary to develop tests and test cases for a component, and it even presents sufficient information to identify entire application test suites.

1.3 Why we need Model Based Testing? [5]

There are three important benefits of testing analysis and design models: Test cases can be known earlier in the course, even as requirements are being identified. Early testing assists analysts and designers to better understand and express requirements and to make sure that specified requirements are testable.

1. Defects can be identified near the beginning in the development course, saving cost and effort. It is extensively recognized that the more rapidly problems are detected, the easier and cheaper they are to repair.

2. Test cases can be evaluated for exactness early on a project. The exactness of test cases in system is always a subject. If test cases are recognized early and applied to models near the beginning in a project, then any misinterpretations of prerequisites on the part of the testers can be rectified early. The model testing assists to make certain that testers and developers have a unfailing realization of system requirements.

3. Even though testing models is very useful, it is significant to not let testing them become the main center of testing efforts. Code testing is still a significant part of the process. Another distinction between conventional projects and projects using object-oriented technologies concerns purposes for software. Think that a new goal in many companies is to create reusable software, extensible designs, or even object-oriented frameworks that symbolize reusable designs. Testing can be done to discover breakdowns in meeting these objectives. Conventional testing approaches and techniques do not tackle such goals [5]

Chapter 2
Background Information and Literature Review

Software testing is the process of executing a program with the intention of locating errors and bugs. Testing is a destructive process of trying to discover the errors in a system and is an assessment of the quality of developed computer artifact. Software testing is a process, or a series of processes, designed to make sure software system does what it was intended to do and that it does not do anything unintended.

2.1 Type of testing strategies

2.1.1 White Box Testing

The function of any security testing process is to make sure the toughness of a system in facing of malicious attacks or usual software failures. White box testing is performed on the basis of understanding of how the system is implemented. White box testing comprises of monitoring of data flow, control flow, coding methods and exception and error management within the system, to test the anticipated and inadvertent software behavior. White box testing can be done to authenticate whether code implementation go behinds intended design, to authenticate implemented security functionality and to reveal exploitable vulnerabilities. [6] White box testing needs right to use to the source code. Though white box testing can be done any point in the life phase after the code is build up, it is a superior practice to do white box testing throughout the unit testing phase.

White box testing needs knowing what builds software safe or unsafe, how to feel like an invader, and how to use diverse testing tools and techniques.[6] The first step in white box testing is to understand and analyze existing design documentation, code and other important development artifacts, so knowing what formulates software secure is a fundamental requirement. Second, to create tests that exploit software, a tester must think like an assailant. Third, to do testing efficiently, testers require knowing the diverse tools and techniques accessible for white box testing. The three requirements do not work in separation.

The general outline of the white box testing process is as follows [6]:

1) To do risk analysis to direct the entire testing process
2) Develop a test approach that identifies what testing steps are needed to accomplish testing.
3) Develop a featured test plan that systematizes the following testing process.
4) Organize the test environment for test execution.
5) Verify test cases and communicate results.

Benefits of White Box Testing Techniques [7]

Introspection: The capacity to come across inside the application denotes that testers can recognize objects programmatically. This is helpful when the GUI is changing frequently or the GUI is yet unknown as it allows testing to proceed. It also can, in some situations, decrease the fragility of test scripts provided the name of an object does not change.
Constancy: A result of introspection, white-box testing can deliver greater strength and reusability of test cases if the objects that include an application by no means alter.
Thoroughness: In circumstances wherever it is necessary to be acquainted with that all paths has been thoroughly tested, that every potential internal interface has been examined, white-box testing is the merely feasible technique. As such, white-box testing presents testers the capacity to be extra thorough in terms of how a great deal of an application they are capable of testing.

Limitation of White Box Testing Techniques

Tricky situation: Being capable to observe every part of an application denotes that a tester have to have featured programmatic information of the application in organize to effort with it properly. This soaring-level of complexity needs a a great deal of more highly skilled individual to build up test case.
Feebleness: The introspection is thought to overcome the issue of application modify shattering test scripts the reality is that frequently the names of objects alter during product development or new paths through the application are added. The fact that white-box testing requires test scripts to be tightly tied to the underlying code of an application

means that changes to the code will often cause white-box test scripts to break. This introduces a high amount of script maintenance into the testing course.[7]

Integrity Problem: For white-box testing to attain the degree of introspection necessary it must be securely integrated with the application being tested. This creates a few problems. To be firmly incorporated with the code you should install the white-box tool on the system on which the application is in succession. But when one wishes to get rid of the possibility that the testing tool is causing either a performance or operational problem, this becomes impracticable to resolve. One more subject that arises is platform support. Due to the highly incorporated nature of white-box testing tools mostly do not offer support for more than one platform. Unluckily in White Box Testing, exhaustive testing of a code presents some logistical troubles. Even for small programs, the number of possible logical paths can be very large. For example a 100 line C Language program that encloses two nested loops running 1 to 20 times depending upon some initial input after some basic data declaration. [7] Inside the inner loop four if-then-else constructs are necessary. Then there are roughly 10 to the power of 14 logical paths that are to be implemented to test the program fully, which denotes that a magic test processor developing a single test case implements it and calculate results in one millisecond would need 3170 years working endlessly for this exhaustive testing which is certainly unrealistic. [7]

2.1.2 Black Box Testing

This testing methodology looks at what are the available inputs for an application and what the expected outputs are that should result from each input. It is not concerned with the inner workings of the application, the process that the application undertakes to achieve a particular output or any other internal aspect of the application that may be involved in the transformation of an input into an output. Most black-box testing tools employ either coordinate based interaction with the applications graphical user interface (GUI) or image recognition.[9]

Most functional test-case generation techniques are based on domain analysis and partitioning. Domain analysis replaces or supplements the common heuristic method for checking extreme values and limit values of inputs [8]. A domain is defined as a subset of the input space that somehow affects the processing of the tested component. Domains

are determined through boundary inequalities, algebraic expressions that define which locations of the input space belong to the domain of interest [8]. A domain may map to equivalent functionality or behavior, for instance. Domain analysis is used for and sometimes also referred to as partitioning testing, and most functional test case generation techniques are based on that. Equivalence partitioning, for example, is one technique out of this group that divides the set of all possible inputs into equivalence Classes. Each equivalence relation defines the properties for which input sets belong to the same partition. Traditionally, this technique has been concerned only with input value domains, but with the advent of object technology it can be extended to the behavioral equivalence Classes that we find in behavioral models. UML behavioral models such as Statecharts, for example, provide a good basis for such a behavioral equivalence analysis, i.e., the test case design concentrates on differences or similarities in externally visible behavior that is defined through the state model. Functional testing completely ignores the internal mechanism, of a system or a component (its internal implementation) and focuses solely on the outcome generated in response to selected inputs and execution conditions [9]. It is also referred to as black box testing, or specification-based testing, a term which is more meaningful and unambiguous. Binder [10] calls these techniques responsibility-based testing. This comes from the notion of a contract [11] between two entities that determine their mutual responsibilities. For example, meeting the contracted precondition assertion is the client's responsibility and meeting the postcondition is the server's responsibility, because this is what it promises to provide after completing a request [10].

Benefits of Black Box Testing

Ease of use: Because testers do not have to concern themselves with the inner workings of an application, it is easier to create test cases by simply working through the application, as would an end user.[7]

Faster test case development: Because testers only concern themselves with the GUI, they do not need to spend time identifying all of the internal paths that may be involved in a specific process, they need only concern themselves with the various paths through the GUI that a user may take.

Simplicity: Where large, highly complex applications or systems exist black-box testing offers a means of simplifying the testing process by focusing on valid and invalid inputs and ensuring the correct outputs are received[7]

In nutshell we can say black box testing is:-

- More effective on larger units of code than glass box testing.
- Tester needs no knowledge of implementation, including specific programming languages.
- Tester and programmer are independent of each other.
- Tests are done from a user's point of view.
- Will help to expose any ambiguities or inconsistencies in the specifications.

Disadvantages of Black Box Testing

Script maintenance: While an image-based approach to testing is useful, if the user interface is constantly changing the input may also be changing. This makes script maintenance very difficult because black-box tools are reliant on the method of input being known.[7]

Feebleness: Interacting with the GUI can also make test scripts fragile. This is because the GUI may not be rendered consistently from time-to-time on different platforms or machines. Unless the tool is capable of dealing with differences in GUI rendering, it is likely that test scripts will fail to execute properly on a consistent basis.

Lack of introspection: one of then greatest limitation of black-box testing is that it isn't more like white-box testing; that it doesn't know how to look inside an application and therefore can never fully test an application or system. The reasons cited for needing this capability are often to overcome the first two issues mentioned. The reality is quite different. [7]

In nutshell we can say black box testing has following limitations:-

- Only a small number of possible inputs can actually be tested to test every possible input stream would take nearly forever.
- Without clear and concise specifications, test cases are hard to design.

- There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
- May leave many program paths untested.
- Cannot be directed toward specific segments of code which may be very complex
- Most testing related research has been directed toward glass box testing

2.1.3 White Box versus Black Box

White-Box testing methodology seems under the covers and into the subsystem of an application. While black-box testing concerns itself exclusively with the put in and outputs of an application, white-box testing allows you to see what is occurring inside the application [7]. White box testing offers a degree of complexity that is not available with black-box testing as the tester is capable to refer to and cooperate with the objects that contain an application slightly than only having admission to the user interface The main disparity between black-box and white-box testing is the parts on which they desire to focus. The black-box testing is focused on results. If an act is taken and it generates the desired result then the procedure that was actually used to achieve that result is inappropriate. White-box testing, on the other hand, is concerned with the features. It focuses on the inner workings of a system and only when all avenues have been tested and the sum of an application's parts can be shown to be contributing to the entire is testing complete.[7]

2.2 Grey Box Testing

Grey box testing is the mixture of black box and white box testing. Purpose of this testing is to locate out defects linked to awful design or bad implementation of the system. [12]. In Gray box testing, test engineer is equipped with the acquaintance of system and designs test cases or test data based on system knowledge.

Grey box testing involves having knowledge of inner data structures and algorithms for need of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the SUT. This difference is

particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages [12]

2.3 Unified Modeling Language

The OMG's Unified Modeling Language helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. [13]

Considering that the UML diagrams can be used in different stages in the life cycle of a system. The 4+1 view offers a different perspective to Classify and apply UML diagrams. The 4+1 view is essentially how a system can be viewed from a software life cycle perspective. Each of these views represents how a system can be modeled. [14]

The different types of views in UML2 [15]:-

The use-case view portrays the functionality of the software from the system users' viewpoint. In this situation, the user could be a different system or a real person and they are called players. The view is made of use-case diagrams and Activity diagrams. It is vital to understanding the entire structure.

The logical view gaze inside the system describing how the system functionality is presented. The static parts of the system are imprisoned in Class and object diagrams; the dynamic part of system is captured in state, sequence, collaboration and Activity diagrams. This is also a very vital view as it exposes most of the system's vocabulary.

The process view focus on the dynamic part of a system and is composed of state, sequence, collaboration and Activity diagrams with a focus on the active Classes.

The implementation view is a explanation of the implementation modules and their reliance. It comprises of component diagrams and is of extra need to programmers.

The deployment view shows the physical topology of system which the hardware system executes. Physical constituents are called nodes. The view is made of deployment diagrams

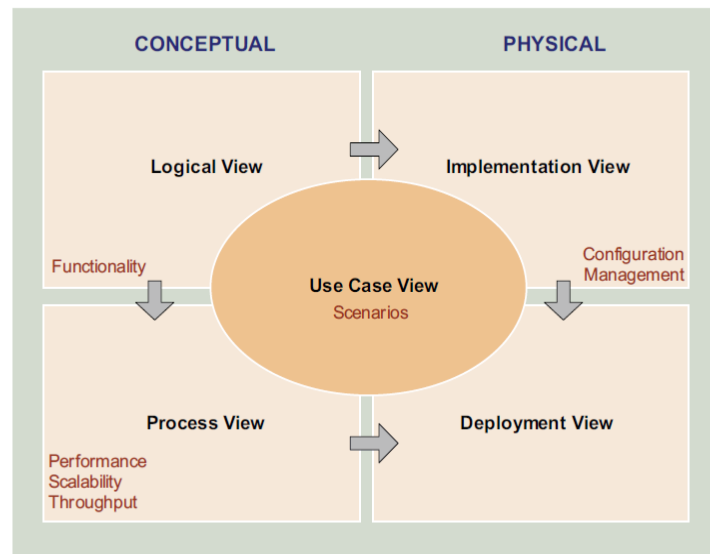


Figure 2.1: The 4+1 View in UML [14]

2.4 UML Based Software Testing

UML based testing is the formation of testing artifacts on the basis of UML models. The models offer the important idea for generating the test cases and test suites, and for validating the absolute implementation of software. [3]

The UML contains diagrams according to the diverse views that we can have on a system. These views can be separated into user view and architectural view, which may be further subdivided into structural view and behavioral view, implementation view, and environmental view. These views can be associated with the different diagram types of the UML. The user view is typically represented by the use case diagram, and the structural view by Class and object diagrams. Sequence, collaboration, Statechart, and Activity diagrams can be associated with the functional and behavioral views on a system, and component and deployment diagrams specify the coarse-grained structure

and organization of the system in the environment in which it will be deployed. So testing can based on diverse types of views we can have on the system. [3]

2.4.1 Use Case Diagram-Based Software Testing

Use cases are helpful for requirements based testing and high-level test suit design of the system. Testing done with the help of use cases can be divided into testing that is based on the use case diagram and testing that is based on use case template, that is similar to distinctive black box testing that is at a higher level of abstraction.

2.4.2 Structural Diagram-Based Software Testing

We can recognize from structural models that what can be tested in a system that comprises of interacting entities. Since Class, objects, packages are the ingredients of structure view, so dependencies, association, cardinality; hierarchies are the test case issues that can be extracted from it.

2.4.3 Behavior Diagram Based Software Testing

The UML describes Interaction modeling through sequence diagrams and Statechart diagrams. A sequence diagram illustrates interactions in terms of temporally arranged method invocations with input and return parameters. Statechart diagrams represent the behavior of an object by notifying its reactions to the receipt of events and sequence diagram. The vertical axis shows the passage of time, and the horizontal axis the objects that participate in an interaction sequence

2.4.4 Interaction Diagram Based Testing

The UML supports Interaction modeling through Statechart diagrams and Activity diagrams. Statechart diagrams represent the behavior of an object by specifying its responses to the receipt of events and Activity diagrams that concentrate on internal behavior of an instance or the control flow within its operations.

There are many phases in the testing process, including unit, function, system, regression, and solution testing. The following table illustrates the differences between these phases, as well as the potential UML diagram for use in the phase. [17]

Table 2.1: The Different Phases and UML Diagrams [17]

Test type	Coverage criteria	Fault Model	UML diagram
Unit	Code	Correctness, error	Class and state diagram
Function	Functional	Functional and API, behavior issues	Interaction and Class diagram
System	Operational scenario	Workload, contention	Use case, Activity, interaction diagram
Regression	Functional	Unexpected behavior from new changed function	Same as function
Solution	Inter system communication	Inter-op problems	Use case and deployment diagram

2.5 Class Diagram Based Testing

Class testing includes activities those are associated with verifying that the implementation of a Class keep up a correspondences exactly with the specification for that Class. If an implementation is right, then each of the Class's instances should act properly. Class testing is same to unit testing in custom and old testing processes. Class testing address few aspects of integration testing as every object defines a level of scope in which many methods cooperate around a set of instance attributes. It is assumed that a Class to be tested has a complete and correct specification, and that it has been tested within the context of the models. If more than one form of specification is used for a Class, then it is assumed all forms are consistent and that information may be taken from whichever form is most useful as the basis for developing test cases for the Class.[5]

2.5.1 Process to Test a Class of Software System

The code for a Class can be tested by evaluated or by executing test cases. Review is a practical option to execution based testing, but has two limitations over execution [5]:

1 Reviews are subject to human fault.

2 Reviews require considerably more effort with respect to regression testing, often requiring almost as many resources as the original testing.

While execution-based testing conquers these drawbacks, a huge amount of is required for the identification of test cases and the development of test drivers. The effort required to build a test driver for a Class can surpass the effort of developing that Class by quite a lot of orders of magnitude. In this particular case, the efforts and advantages of testing the Class outside the system in which it will be used must be calculated. The condition is not unusual to object oriented programming. The similar condition invokes in old procedural development with respect to many of the subprograms invoked at higher levels in a structure chart. [5]

2.5.2 Who Test the Class in Software System?

Classes are mostly tested by developer, similarly as sub programs are unit tested by the developer. A developer also plays the function of a Class tester and minimizes the number of people that have to understand a Class's specification. It also facilitates implementation based testing as the tester is intimately well-known with the code. Finally, the test driver can be used by the developer to correct the code as it is written. A goal of testing is to uncover errors, not to fix bugs. However, a helpful component of Class testing is in helping to separate errors in the code. The main drawback of test drivers and code being developed by the same person is that any misunderstandings of the specifications by the developer will be propagated to the test suite and test drivers. These problems are headed off by formal reviews of the code, and by requiring a test plan to be written by another Class developer, and by allowing the code to be reviewed independently. It is not abnormal for independent testers to find problems with the specifications for a Class, so time should be allowed during testing to resolve them. [5]

2.5.3 What to Test in a Class?

It is strictly required to ensure that the code for a Class precisely meets the requirements set onward in its specification. Incomplete coverage of code after a wide range of test cases have been run against the Class could be an indication that the Class contains extra, undocumented behaviors. Or it could merely suggest that the

implementation must be tested using more test cases. The amount of consideration given to testing a Class to make sure it does nothing more than what it is specified for depends on the risk associated with the Class supplying extra behaviors. [5]

2.6 Activity Diagram Based Software Testing

UML Activity diagrams are mostly helpful for structural testing activities; it signifies that they provide same information as source code or control flow graphs in custom white box testing, although at a upper level of abstraction if required. Developing Activity diagrams can be seen in most cases as programming without a specific programming language. [3]

Control flow testing in a single instance maps to a white box unit test, however it is only important in component development and testing. Component based testing deals more with the integration of objects and mutual interactions relatively than with their individual inner functioning. For unit testing, Activity diagrams provide custom code coverage measures, although at a higher level of abstraction. [3] An Activity may be a single statement, a cluster of statements, or a full procedure with looping and decisions. Code coverage criteria can be tailored without difficulty to cope with Activity diagram concepts. Traditional control flow graphs and UML Activity diagrams are basically the same. Bezier treats control flow-based testing thoroughly [18].

One can identify flows of control within an instance of an Activity diagram that we can map to traditional coverage criteria:

- Testing the control flow graph with the concept of conventional coverage criteria.
- Control flow coverage of every Activity in the Activity diagram of the system.

The component based testing with the UML is action that is spread over a number of different objects. This reflects the collaborations of objects, their mutual effort toward a single goal. In this case it is the procedure of the Activity. Such higher-level procedures cross component or object boundaries. At a boundary between two objects any flow of control is translated into some operation invocation or some signal invocation between the objects. The client object calls the methods of the server object. Here we have a typical contract at the particular connection between the two objects, so for testing one

have to go back to the structural model and the behavioral model of each entity and derive appropriate test cases for assessing this interaction point. [3]

We can identify flows of control between instances of an Activity diagram that we can map to traditional coverage criteria:

• Message flow coverage in the Activity diagram.

• Signal flow coverage in the Activity diagram.

The upper level transactions are made of lower level transactions of loads of different objects. The decomposition Activity is accountable for generating these calling hierarchies between transactions. .

2.7 State Diagram Based Testing of Software System

State based testing focus on ensuring the correct implementation of the component state model of the system. The test case design is based on the individual states and the transitions among these states. In object oriented or component based testing, any type of testing is effectively state-based as soon as the object or component exhibits states, even if the tests are not obtained from the state model. [3] In that point, there is no test case without the notion of a state or state transition and pre and post conditions of every single test case must consider states and behavior. Binder [10] presents a thorough overview of state-based test case generation, and he also proposes to use so-called state reporter methods that effectively access and report internal state information whenever invoked.

The main test case design strategies or testing criteria for state-based testing: [3]

2.7.1 Piecewise Coverage

Piecewise coverage focus on exercising different specification parts, like coverage of every state, every event, or every action. These techniques are not expressly related to the structure of the every state machine that executes the behavior, so it is only incidentally efficient at finding behavior error and faults. It is possible to visit every state and miss some events or actions, or produce every action without visiting all states or accepting all events.

2.7.2 Transition Coverage

Full transition coverage is achieved through a test suite if every specified transition in the state model is exercised at least once. As a consequence, it covers all states, all events, and all actions. Transition coverage may be improved if every specified transition sequence is exercised at least once; this is referred to as n-transition coverage [16].

2.7.3 Round-trip Path Coverage

Round-trip path coverage is defined as the coverage of at least every defined sequence of transitions that begin and end in the same state. The shortest round trip path is a transition that loops back on the same state. A test suite that achieves full round-trip path coverage will reveal all incorrect or missing event/action pairs. A test case comprises a precondition, a conditional expression that should be true before the test event is executed, an event, or a sequence of events, and a post condition that should be true after the events have been executed. The pre and post conditions are made up of a number of conditional expressions that refer to the input parameter values of the event as well as to the internal attributes of an object before and after event execution. Sometimes they also include the actions that are performed on other associated objects, that is, if they effectively change the state of these other objects. We may also call this the outcome of an event. The combination of the object's internal attributes is its internal state. An internal state is any arbitrary combination of an object's attributes. Sometimes this is also referred to as state machine or physical state. In system development we are usually only concerned with value domains of attribute combinations that cause an object to exhibit different behavior. Such a value domain represents an externally visible or logical state. [3]

2.8 Class Diagram Based Testing Related Work

2.9.1 UML Class Diagram Based Testing Using Verification [19]

Lee Copeland gives this method. When performing syntax testing, this approach verifies that the Class diagram contains correct and proper information. There are three kinds of questions: Is it complete? Is it correct? Is it consistent?

Complete:

1. Does each Class define attributes, methods, relationships, and cardinality?
2. Is each association well named?
3. Is each association and aggregation's cardinality correct?

Correct:

1. Are all attributes private?
2. Are all parameters explicit rather than being embedded in method names?
3. Do all sub Classes implement the "is-a-kind-of" relationship properly?
4. Are all object states represented explicitly using states and transitions rather than as sub Classes?
5. In inheritance structures, are all attributes and methods pushed as high in the inheritance structure as is proper?
6. Are all polymorphic methods within related subClasses identically named?
7. Does each association reflect a relationship that exists over the lives of the related objects?

Consistent:

1. Are each 0...* and 1...* Relationships implemented with containers/collectors?
2. Are each association's cardinalities consistent (instantaneous vs. over-time)?

Domain expert testing- after checking the syntax of the Class diagrams, the second type of testing—domain expert testing comes. There are two options: either find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) In this process two kinds of questions arise: Is it correct? Is it consistent?

Correct:

1. Is each Class named with a strong noun?
2. Have all redundant, irrelevant, or vague Classes been removed from the diagram?
3. Is each attribute defined within the proper Class? Is it of the correct type?
4. Is the visibility of each attribute correct?
5. Are the default values of each attribute specified correctly?

6. Is each attribute essential rather than computable from others?
7. Is each method in the correct Class?
8. Are all method names strong verbs?
9. Does each method take the correct input parameters and return the correct output parameter?
10. Is the visibility of each method correct?
11. Does each method implement one and only one behavior?
12. Is the public interface free from unnecessary methods?

Consistent:
Is the Class diagram drawn at the appropriate level: conceptual, specification, or implementation?
Trace ability testing- Finally, after having our domain expert scours the Class diagrams, the third type of testing—tractability testing arises. It is to ensure to trace from the use cases and the sequence diagrams to the Class diagrams and from the Class diagrams back to the use cases and sequence diagrams. Again, one question arises: Is it consistent?
Consistent:
1. Is each object on the sequence diagram represented by a Class on the Class diagram?
2. Is every message on the sequence diagram reflected as a method in the appropriate Class?

2.10 State Diagram Based Testing Related Work

2.10.1 State Diagram Testing using Flow Graph

Kansomkeat and Rivepiboon [21] have proposed a method for generating test sequences using UML Statechart diagrams. They transform the Statechart diagram into a flattened structure of states called testing flow graph (TFG). From the TFG, they list possible event sequences which they consider as test sequences. The testing criterion they used to guide the generation of test sequences is the coverage of the states and transitions of TFG. It automatically generates test cases from UML specification with the aid of the Rational software Corporation's Rational Rose tool. The first step, we create Statechart diagrams from this tool and use it to transform into intermediate diagrams, called Testing

Flow Graph (TFG). It reduces the complexity of UML Statechart diagrams. The TFG is a clearly flow and simple structure diagram. Then, the TFG is used to generate test sequences by parsing to coverage the state and transition

2.10.2 State Diagram Testing using Finite State Machine

Kim et al. [22] proposed a method for generating test cases for Class testing using UML Statechart diagram. They transformed Statecharts to extended FSMs (EFSMs) to derive test cases. In the resulting EFSMs, the hierarchical and concurrent structure of states is flattened and broadcast communications are eliminated. Then data flow is identified by transforming the EFSMs into flow graphs, to which conventional data flow analysis techniques are applied. The method transforms state diagrams into extended finite state machines (EFSMs). Then data flow is identified by transforming EFSMs into flow graphs to which convention data flow analysis techniques can be applied

2.11 Activity Diagram Based Related Work

2.11.1 Activity Diagram Testing using Slicing

Generally test cases from Activity diagram are derived by using dynamic slicing. Originally, dynamic slicing was defined and used with respect to program code. Dynamic slices and slicing criteria are defined in terms of flow dependency graph to make it applicable to Activity diagram. A dynamic slice helps to consider the dependencies among activities that arise during run time. Dynamic slices are based on the FDG of an Activity diagram using an edge marking method. Construction of FDG is the only static part in our approach. Slices are created from FDG corresponding to conditional predicates at each Activity edge in the respective Activity diagram. It is needed to apply function minimization methods to generate test data based on each slice. This approach automatically generates test data to achieve high path coverage. The test cases generated through this approach can be used for testing cluster level behaviors. [23] In this approach, we have not addressed specific issues like concurrency and polymorphism.

In an Activity diagram, all Activity edges are labeled with guard conditional predicates. Of course the conditional predicate might trivially be an empty predicate, which is always true. For each Activity edge, there will be a corresponding node in FDG. From FDG, we create the dynamic slice for the slicing criteria (m) for the given set of inputs for each e

node. In our case, the slicing criterion (m) specifies the location (i.e. identity) m of an e node in FDG. [23]For the given set of inputs, those nodes of FDG that do not affect the predicate at m for a given execution are removed to form the dynamic slice of an Activity diagram, for the slicing criterion (m). With respect to each slice, we generate test data. The generated test data for the predicate associated with m corresponds to the true or false values of the predicate and also, these test data values are generated subject to the slice condition.[23]

Chapter 3
Problem Statement

Problem statement describes the gap in the existing work and problem formulation. The gap in existing work shows the limitation and flaws in the existing work and which technique they have used. In problem formulation, it has been given appropriate solution to solve the existing problem and suggested the novel work.

3.1 Gaps in Existing Work

3.1.1 Gap in Class Diagram's Existing Work

In **"UML Class Diagram Based Testing Using Verification"** testing performed with the help of design review and it tells about how to verify design. It has syntax testing, domain expert testing and trace ability testing. It does not deal with automated testing. It does not have any test case generation technique

In "**A Practical Guide to Testing Object-Oriented Software**" [5] test cases has been generated on the basis of Class diagram using OCL [24], but the problem is that these test cases have not been generated automatically. Also test cases depend on Class diagram and OCL, thus test cases generated did not tell about initial and final state of the system. One more problem is that it is assumed that models are consistent. If not, the test case generation will result in error in most cases.

In **"Software Engineering: A Practitioner's Approach" [25]** test cases have been generated on the basis of Class in a code. These approaches have not been used in UML Class diagram. Also this approach generates test cases manually not automatically.
In **"inter Class test case generation**" [26] test cases has been generated on the basis of Class collaboration diagram, but the problem is that these test cases have not been generated automatically. Another problem is that it is assumed that models are consistent. If not, the test case generation will result in error in most cases. As test cases depends only on Class diagram.

3.1.2 Gap in Statechart Diagram's Existing Work

In **"Automatic test case generation using Unified Modeling Language (UML) State diagrams"** test cases has been generated on the basis of transition path coverage. But the test cases generated deals with initial state, final state and test data only. These test cases are not very efficient and effective as lot of information like Initial state, transition, final state and Class related data is not present in these test cases.

In **"*Testing Object-Oriented Systems: Models, Patterns and Tools*"** test cases generated depends on only transition path coverage criteria. Test cases generated are efficient and effective, as this approach does not miss any transition. But this work is not automated and also efficiency of test case depends on only one UML diagram Statechart diagram.

3.2.3 Gap in Activity Diagram's Existing Work

In **"Automatic test case generation using unified modeling language"** test cases generated by transition path coverage criteria. Test cases generated are efficient and effective, as this approach does not miss any transition. But this work is not automated and also efficiency of test case depends on only one UML diagram Statechart diagram. Moreover information like pre condition, post condition, guard conditions and Class related data is not present in these test cases.

3.2 Problem Formulation

Generally testing is performed on coding part, which contains variables, loops, executable statements, Classes, functions etc. This type of testing is to tedious to fix problems because if executable statements or functions or number of loops or variables in a code have been changed then number of test cases will also change. Let a code has two for statements in it thus program has two predicate nodes in a control flow graph. According to white box testing methods basis path testing has two predicates, thus its cyclomatic complexity will be two and it has two independent paths. According to Basis Path Testing, test cases derived to exercise the basis set are guaranteed to execute every

statement in the program at least once during testing. If a new for statement is added to the code then number of predicate nodes in the control flow graph will be three. Now this code has cyclomatic complexity three and has three independent paths. Its test cases will also increase. That means increase in statements leads to rise in complexity, which will affect the test plan and test case structures.

Similar problem occurs in object oriented programming methodology, it contains Classes and functions. If Class scope has been changed that means public, private, protected or if a new Class has been added that is inherited from or dependent upon other Classes then it is very difficult task to find which module is affected by this change and what the affect on other modules is. It is very difficult to identify test cases required to fix this problem and also to know whether these test cases are optimal or not. There are two solutions for this problem first one is coding review and second one is model based testing of UML diagrams of system. Individual performs coding review manually, but this is not very efficient method as each programmer has his own style of coding. Another alternative is to perform dynamic testing that means run software and check behavior of software. But these procedures again indirectly focus on coding review for finding errors or change effect. Coding review is not much efficient method, which is used in structured and object oriented approach. So a novel solution has been proposed that is model based testing. In software development life cycle design part comes first than coding or implementation part. Test case should be applied on design rather than coding because a software design is converted into code. So if problem occur in design, then it can be corrected before coding. This results in error correction before coding and errors do not propagate to next phase. This leads to reduction of cost of correcting errors after software is ready.

To perform model based testing, Unified Modeling Language (UML) is used. UML has different design diagrams that are used to specify the software behavior i.e. static and dynamic. UML diagram methodology is applied on object-oriented approach, not on structured approach. Many researchers and practitioners are working on UML based testing. UML based testing is independent of code; it specifies general behavior of software. This type of design can be easily deployed on any languages like C++, Java etc. UML design can be used to generate test cases. UML deals with different

diagrams like Class, sequence, Activity, Statechart, component diagram, object, collaboration diagram. All these diagrams can be used for testing.

Each diagram in UML describes different view, so test cases generated from different diagram presents different test cases view. Different diagrams can be combined and testing can be done on system. But there are some limitations like:

- UML is a graphical view of software. If an automated testing is performed on UML design, it is difficult. Automated testing means automatically generate test cases from deign.
- How to combine three UML diagrams and perform testing i.e. how information from one UML diagram can be passed to another UML diagram for efficient test case generation.
- Finding static test cases and dynamic test cases from UML diagrams. UML diagram has two-type of behavior i.e. static and dynamic. So finding static test cases and dynamic test cases that are optimal solution.

Above said problem can be resolved with help of proposed system. In the proposed system, test case generation is performed on three diagrams i.e. Class diagram, Activity diagram and Statechart diagram. Proposed system focus on these three diagrams and with the help of these diagrams an optimal solution can be found and better test case generation can be performed than previous approaches, which are based on either single UML diagram or two UML diagrams.

.

3.3 Objectives of thesis.

In present work UML design based automatic testing is proposed with following objectives:-

- To explore existing UML based test case generation techniques.
- To analyze various UML diagrams for their suitability to generate test cases.
- To design & develop a technique to generate test cases from UML Class, Activity and Statechart diagram.
- To develop a tool that generates test case automatically from petal files of UML Class diagram, Activity diagram, and Statechart diagram.

Chapter 4
Proposed System and Implementation

The proposed system focuses on the three UML diagrams that are Class diagram, Activity diagram and state diagram. Various types of testing information is extracted from there diagrams to generate test cases from these diagrams. For static test case information Class diagram is used and for dynamic type of test case information Activity and state diagrams are used. With the help of Class diagram the information that is extracted out include Classes, relationship between Classes, dependency, parent/child relationship, associations. With the help of Statechart diagram information that is extracted include Initial states, Transitions, Final states, Pre conditions, Post conditions. With the help of Activity diagram information is extracted include Initials states, Final states, Guard condition between activities.

4.1 Brief Overview about System

In UML Class diagram testing, the information like association, cardinality, inheritance, are found. This information is used to generate static test cases automatically. UML Activity diagram and Statechart diagram aid to generate dynamic test cases. The information like source state, transition, target state from Statechart is clubbed with information like pre-condition, post-condition, guard conditions from Activity diagram. This information is extracted with the help of programming language and further been stored in database. A guard condition emerges at the beginning of the interaction and encloses all the information that is needed to make the decision about whether to execute the interaction. If the guard condition tests true, the traces execute. Because a guard condition is non-compulsory and the interaction can also takes place if no guard condition is specified in the interaction. The pre condition of activity or state defines constraints which are needed for successful execution. It constitutes the activity caller's portion of the contract. The post condition is a condition that must always be true just after the completion of activity. The Post conditions are sometimes tested using assertions within the activity itself.

4.2 Methodology

Following are the major steps in proposed techniques: -

1. **Input Petal File into the system**: The Petal file of Class diagram is given as input to the developed tool. Petal files [27] nothing but files which one gets by opening a Rational Rose file in word pad or text file.

2. **Read mdl file**: The mdl file is read by the tool and with the concept of tokenizer patterns like Class name, Class attributes, Class cardinality, Class operations, inheritance, dependency etc are found.

3. **Is pattern found**: If pattern is found then it is entered to the queue else next pattern is found.

4. **Store the patterns in a queue:** All pattern found is stored in a queue. There are different queues for every pattern like Class name queue, Class attributes queue, Class cardinality queue, Class operations queue, inheritance queue, dependency queue etc.

5. **Search for another pattern**: Tool searches for various pattern in petal for Class name if it is found then Class name is entered into Class name queue Else if it found Class attributes it enters it to the Class attributes queue similarly so on.

6. **Is EOF (End of File) is reached**: Tool keeps on searching the patterns until EOF is reached.

7. **Create text file from queue and store text file in database**: When end of file is reached, the tool generates the text file which contains all information about Class diagram in form of tuples which can be entered to the database easily using sql *loader. Sql *loader loads all the data from the text file to the Oracle database.

8. **Are all Petal files input**: After Class diagram, Petal files of Activity diagram and Statechart diagram are entered to the tool and loop back to second step in flow chart.

9. **Retrieving strings to generate test cases**: There are tables present in the Oracle database which contains the information from Class diagram, sequence diagram, Statechart diagram. From Net beans IDE tool fires queries to generate test cases.

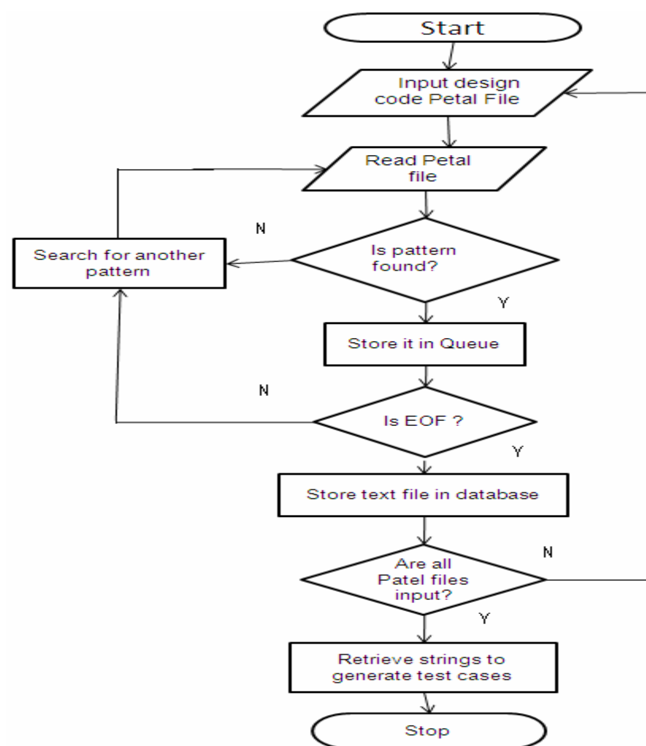


Figure 4.1: The Flow Chart of Methodology

4.3 Description of Methodology

Firstly Class diagram, sequence diagram and Statechart diagram are drawn for a particular system using Rational Rose and corresponding mdl file of Class diagram, Statechart diagram, Activity diagram are saved. These diagram's petal files (mdl file) [27] are parsed by the tool. Firstly Class diagram petal file is parsed by the tool. The tool reads the petal file line by line. It tokenized the line into words and then matched the

string with the pattern to find the Class name. When Class name is found it entered it to the queue and searched for another pattern, its attributes, its operations, its inheritance Classes, its dependency, its cardinality and all these are written on a text file. Similarly petal file of Activity diagram and Statechart diagram are entered in the tool. Then the text files created are entered to the Oracle database using SQL *Loader [28]. Before loading text files to SQL *Loader it was necessary to create control file and a Dat file .The control file is a text file written in a language that SQL*Loader understands. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, where to insert the data. Dat file was the file which contains the data i.e. the text file from the tool is saved with extension .dat and it served the purpose of dat file in SQL *Loader to load the data. Then is required to create a table in database and all the text files in the form of Dat file were loaded to Oracle database. Now Java and Oracle are connected and queries are fired from Java to extract the strings from database and use them to generate test cases.

```
  notation      "Unified")
root_usecase_package      (object Class_Category "Use Case View"
  quid          "39C9260C00D6"
  exportControl      "Public"
  global        TRUE
  logical_models     (list unit_reference_list
     (object Class "Person"                          Class name
        quid          "4BF37FE80000"
        used_nodes    (list uses_relationship_list
           (object Uses_Relationship                 Dependency
              quid          "4BF3830001C5"            Relationship
              supplier      "Use Case View::Addresses"
              quidu         "4BF381C600DA"))
        class_attributes   (list class_attribute_list
           (object ClassAttribute "name"              Class Attribute
              quid          "4BF3800C0290")
           (object ClassAttribute "phoneNumber"
              quid          "4BF3800F033C")
           (object ClassAttribute "emailAddress"
              quid          "4BF3801800BB")))
     (object Class "Student"
        quid          "4BF3808B0261"
        superclasses       (list inheritance_relationship_list
           (object Inheritance_Relationship          Inheritance Relationship
              quid          "4BF3819D004E"
              supplier      "Use Case View::Person"
              quidu         "4BF37FE80000"))
        operations    (list Operations
           (object Operation "isEligibleToEnroll"    Class Operation
              quid          "4BF381130196"
              concurrency        "Sequential"
              opExportControl    "Public"
              uid           0)
           (object Operation "getRegistered"
              quid          "4BF3969902CE"
```

Figure 4.2: Petal file of Class Diagram

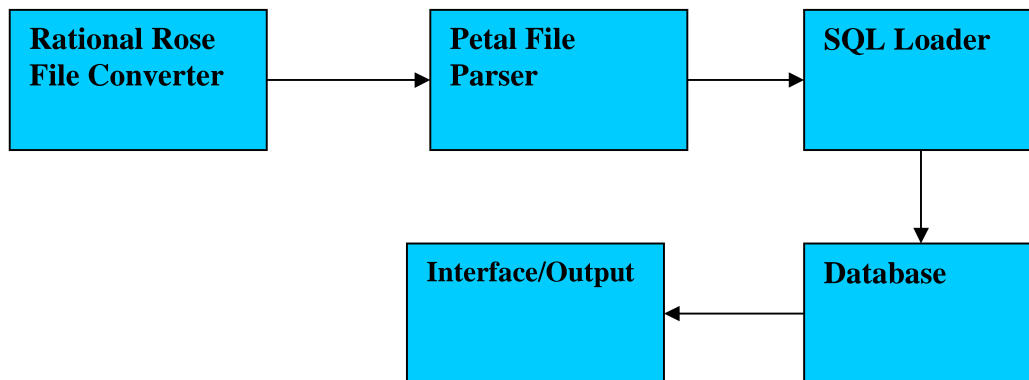


Figure 4.3: Block Diagram of System

The SQL command to create table for Class data is given below .The table contains Class Classname VARCHAR(20), Operation VARCHAR(20), Attribute VARCHAR(20), Dependency VARCHAR(20), Inheritance VARCHAR(20).

create table Class1(

Classname VARCHAR(20),

Operation VARCHAR(20),

Attribute VARCHAR(20),

Dependency VARCHAR(20),

Inheritance VARCHAR(20));

Data from text file is entered to the tool with the help of SQL *Loader [28]. In order to enter the data from text file to Oracle database Dat and control files are created. Dat file is nothing but text file generated from tool with the extension of dat and control file is a text file written in a language that SQL*Loader understands. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, and where to insert the data.

The syntax of control file is given below:

LOAD DATA
INFILE 'C:\Documents and Settings\Software\Desktop\Classresult.dat'
APPEND INTO TABLE Class1
FIELDS TERMINATED BY","(Classname, Operation, Attribute, Dependency, Inheritance)

The Class test case information is entered into database. The below figure shows the data present in a Class, which may extracted at later stage to generate the test case with the help of interface (Netbeans)

CLASSNAME	OPERATION	ATTRIBUTE	DEPENDENCY	INHERITANCE
Admin	allow	ID	Room	null
Admin	deny	Passwrd	null	null
Admin	Validating_customer	null	null	null
Admin	Room_checking	null	null	null
Admin	Report_generation	null	null	null
Customer	Billing	Name	null	null
Customer	Signin	Contact	null	null
Customer	RoomCancel	Age	null	null
Customer	RoomSelectiobn	Gender	null	null
Customer	null	Address	null	null
More than 10 rows available. Increase rows selector to view more rows.				

Figure 4.4: The Test Case Data from Class in the Database

The SQL command to create table for Statechart diagram is shown below and the table contains Transition VARCHAR(200), Initial State VARCHAR(50), Final State VARCHAR(50));

create table Statechart1(
Transition VARCHAR(200),
InitialState VARCHAR(50),
FinalState VARCHAR(50));

Data from text file is entered to the tool with the help of SQL *Loader [28]. In order to enter the data from text file to Oracle database Dat and control files are created. Dat file is nothing but text file generated from tool with the extension of dat and control file is a text file written in a language that SQL*Loader understands. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, and where to insert the data.

The syntax of control file is given below:

LOAD DATA
INFILE 'C:\Documents and Settings\Software\Desktop\stateresult1.dat'
APPEND INTO TABLE Statechart1
FIELDS TERMINATED BY","(Transition, InitialState, FinalState)

The Statechart test case information is entered into database. The below figure shows the data present in a Statechart, which may extracted at later stage to generate the test case with the help of interface(Netbeans)

TRANSITION	INITIAL STATE	FINAL STATE
[bill_generation]	"RoomSelected"	"Credit_Card_Swapped"
[room_vacant>0]	"RoomSearched"	"RoomSelected"
[bal>=rent +1000]	"Credit_Card_Swapped"	"RoomConfirmed"
[bal<=1000+rent]	"Credit_Card_Swapped"	"quit"
[!valid_pin]	"Credit_Card_Swapped"	"Credit_Card_Swapped"
[report_ready]	"RoomConfirmed"	"ReportEmailedTocustomer"

Figure 4.5: The Test Case Data from Statechart Diagram in the Database

SQL command to create table for Activity diagram is given below and it contains Guard VARCHAR(20), Precondition VARCHAR(20), Postcondition VARCHAR(20));

create table Activity(
Guard VARCHAR(20),
PreCondition VARCHAR(20),
PostCondition VARCHAR(20));

Data from text file is entered to the tool with the help of SQL *Loader [28]. In order to enter the data from text file to Oracle database Dat and control files are created. Dat file is nothing but text file generated from tool with the extension of dat and control file is a text file written in a language that SQL*Loader understands. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, and where to insert the data.

The syntax of control file is given below:

LOAD DATA
INFILE 'C:\Documents and Settings\Software\Desktop\Activity.dat'
APPEND INTO TABLE Activity
FIELDS TERMINATED BY",''(Guard, PreCondition , PostCondition)

The Statechart test case information is entered into database. The below figure shows the data present in a Statechart, which may extracted at later stage to generate the test case with the help of interface(Netbeans)

GUARDCONDITION	PRECONDITION	POSTCONDITION
[room_vacant==0]	pre_user_search_room	post_user_quit
[room_vacant>0]	pre_user_search_room	post_room_available
[bal<1000+rent]	pre_payments_details_provided	post_card_rejected
[!valid_pin]	pre_pin_required	post_swap_card_again
[bal>rent+1000]	pre_payment_details_ready	post_credit_card_accepted
[!ready_report]	pre_report_in_progress	post_user_wait
[report_ready]	pre_report_generated	post_report_sent_to_user

Figure 4.6: The Test Case Data from Activity Diagram in the Database

All the data from Class diagram, Activity diagram, Statechart diagram is entered to the Oracle database using dat files and control files. With the help of JDBC Java is connected to Oracle database and queries are fired from Java on different tables to extract the data and to generate the test cases

To sum up the methodology that has been used in proposed system: -

1. Draw the Class, Activity, state diagram with the help of Rational Rose.
2. Save the corresponding petal file of the above diagrams.
3. Parse the petal file of all the diagrams with the help of the tool.
4. With the help of tokenizers, extract the desired information for test case generation.
5. Store the test case information in a database.
6. Extract the test case information in the form of test suit

Chapter 5
Testing & Results (Experimental Results)

5.1 Case study: Online Hotel Reservation System

This case study deals with online reservation of rooms of hotel and allow users to look for and book hotels online A user can search by day, hotel, accommodation type, rates, place, and number of people .The proposed system checks and daily update inventory as rooms are occupied The proposed system has a payment gateway to process their bookings in real-time. Hotels can be notified of bookings by email or fax as required. Users can signing and check the details of the booking or cancel booking. Hotels can sign in and revise: - tariff - Room accessibility - Room types and description.
In the proposed system the testing is performed on three diagrams namely Class diagram Activity diagram and Statechart diagram.

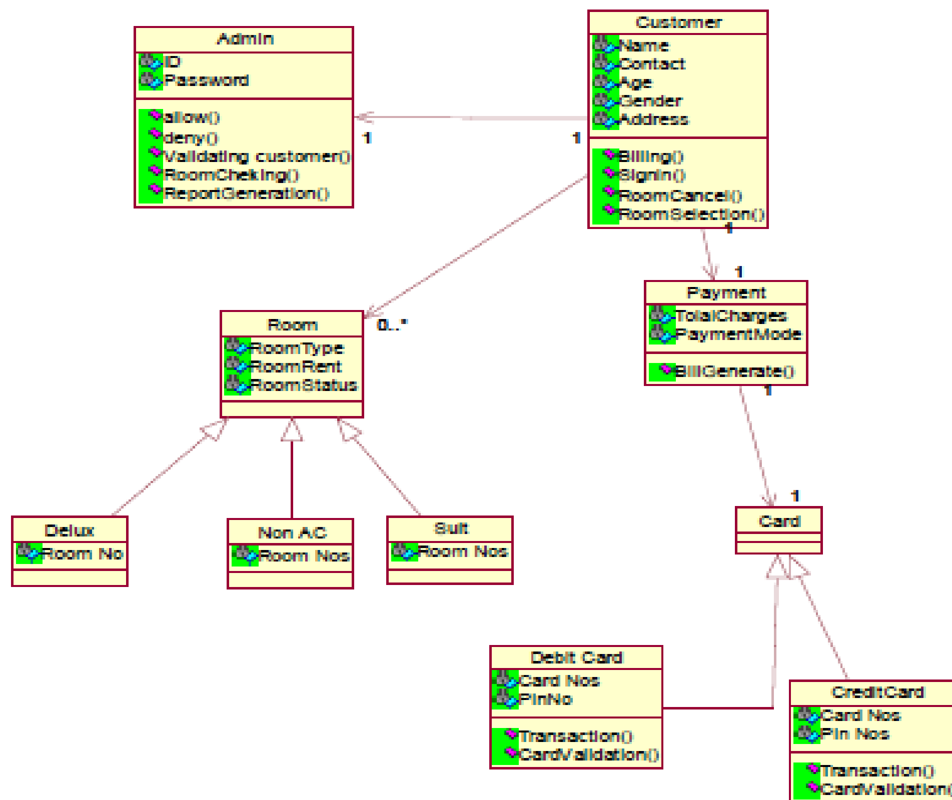


Figure 5.1: Class Diagram of Hotel Reservation System

With the help of Class diagram the following information is extracted which will serve the purpose of generating static test cases:-

- Classes
- Relationship between Classes
- Dependency
- Parent/child relationship
- Associations

The manual test case test case generated in the proposed system from Class diagram:-

Table 5.1: Test Cases from Class diagram

CLASSNAME	OPREATION	ATTRIBUTE	DEPENDENCY	INHERITANCE
Admin	**Deny**	**Passwrd**	**Null**	**Null**
Admin	**Validating_customer**	**Null**	**Null**	**Null**
Admin	**Room_checking**	**Null**	**Null**	**Null**
Admin	**Report_generation**	**Null**	**Null**	**Null**
Customer	**Billing**	**Name**	**Null**	**Null**
Customer	**Signin**	**Contact**	**Null**	**Null**
Customer	**RoomCancel**	**Age**	**Null**	**Null**
Customer	**RoomSelection**	**Gender**	**Null**	**Null**
Customer	**Null**	**Address**	**Null**	**Null**

With the help of Statechart following information will be extracted

- Initial states
- Transitions
- Final states

Initial state is that state from where an event starts. Final state is a state is a state at which event ends and transition is stimulus that triggers an event. All three are the necessary constituents of state diagram.

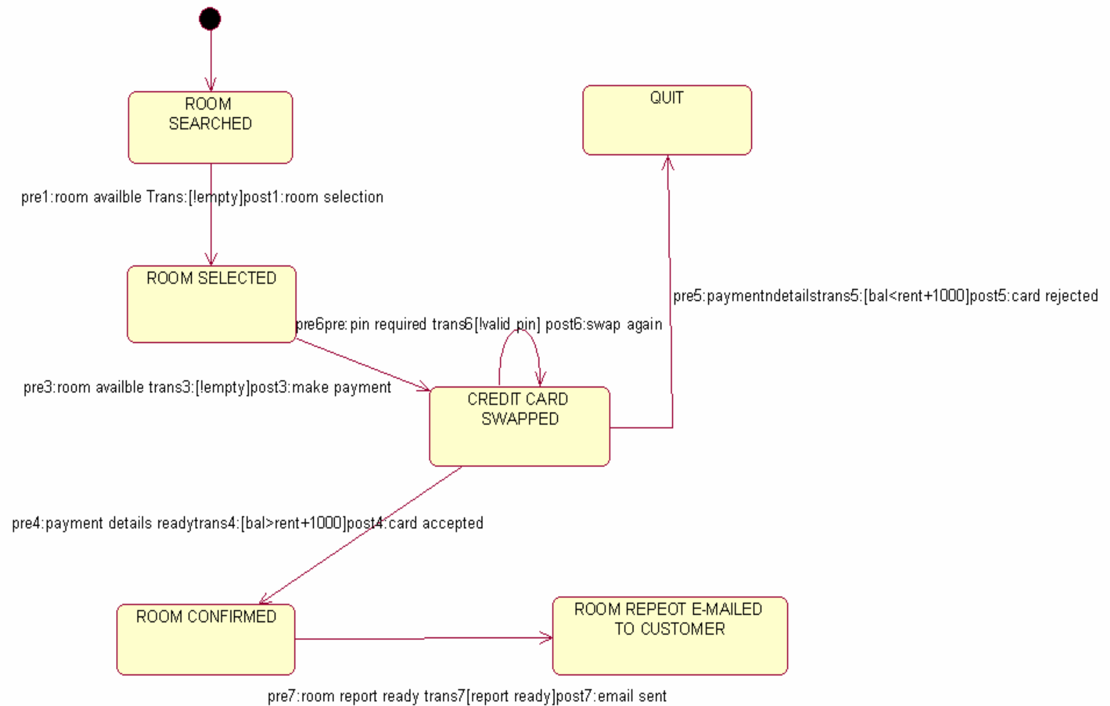


Figure 5.2: State Diagram of Hotel Reservation System

The test cases generated from the state diagram are shown in the table 5.2. It contains Three fields namely Transition, InitialState, FinalState. Initial state contains information of initial state and final state of a transition.

Table 5.2: Test Cases from State Diagram

Transition	InitiaState	FinalState
[bill_generation]	"RoomSelected"	"Credit_Card_Swapped"
Room_vacant>0	"RoomSearched"	"RoomSelected"
[bal>=rent +1000]	"Credit_Card_Swapped"	"RoomConfirmed"
[bal<=1000+rent]	"Credit_Card_Swapped"	"quit"
[!valid_pin]	"Credit_Card_Swapped"	"Credit_Card_Swapped"
[report_ready]	"RoomConfirmed"	"ReportEmailedTocustomer"

The information extracted from the Activity diagram includes, pre condition and post condition and guard condition between activities..

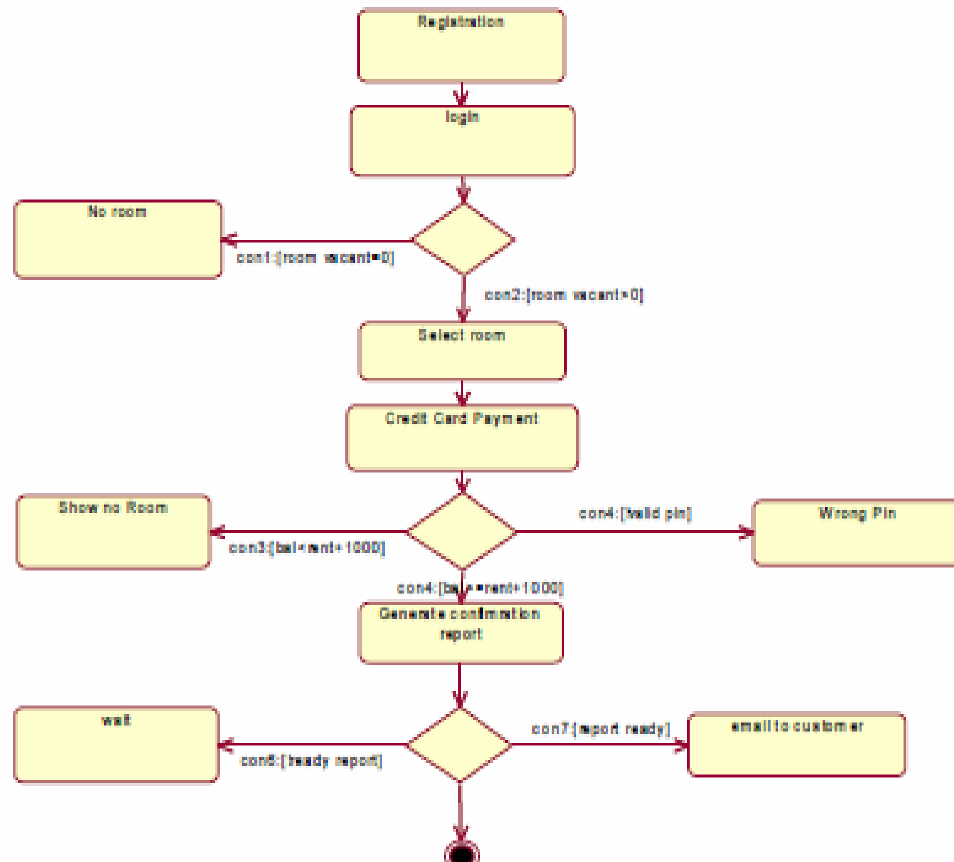


Figure 5.3: Activity Diagram of Hotel Reservation System

A guard condition emerges at the beginning of the interaction and encloses all the information that is needed to make the decision about whether to execute the interaction. If the guard condition tests true, the traces execute. Because a guard condition is non-compulsory and the interaction can also takes place if no guard condition is specified in the interaction. The pre condition of activity or state defines constraints which are needed for successful execution. It constitutes the activity caller's portion of the contract. The post condition is a condition that must always be true just after the completion of activity. The Post conditions are sometimes tested using assertions within the activity itself

Table 5.3 Test Cases from Activity Diagram

Guard	PreCondition	PostCondition
[room_vacant==0]	pre_user_search_room	post_user_quit
[room_vacant>0]	pre_user_search_room	post_room_available
[bal<1000+rent]	pre_payments_details_provided	post_card_rejected
[!valid_pin]	pre_pin_required	post_swap_card_again
[bal>rent+1000]	pre_payment_details_ready	Post_credit_card_accepted
[!ready_report]	pre_report_in_progress	post_user_wait
[report_ready]	pre_report_generated	post_report_sent_to_user

5.2 Combine test cases from Activity and State diagram of Hotel Reservation System

From Statechart diagram it is known that which the source state is and after a specific transition what is the final state. From the Activity diagram important information retrieved for test case generation that is path coverage and guard conditions. Both the information from Activity diagram and state diagram are clubbed together to form dynamic test cases. These test cases are the powerful form of test cases that are generated separately from individual state and Activity diagram and are more accurate. These combined test cases cover broader aspect of testing.

Table 5.4: Combined Test Cases from Activity and State diagram

Transition	InitiaState	FinalState	PreCondition	PostCondition
[bill_generation	RoomSelected	Credit_Card_Swapped	_________	___________
room_vacant>0	RoomSearched	RoomSelected	pre_user_search_Room	post_room_available
[bal>=rent +1000]	Credit_Card_Swapped	RoomConfirmed	pre_payment_details_ready	post_credit_card_Accepted
[bal<=1000 +rent]	Credit_Card_Swapped	Quit	pre_payments_details_provided	post_card_rejected
[!valid_pin]	Credit_Card_Swapped"	Credit_Card_Swapped	pre_pin_required	post_swap_card_again
[report_ready]	RoomConfirmed	ReportEmailed Tocustomer	pre_report_generated	post_report_sent_to_user

5.3 Snapshot of Tool

The following figure shows the snapshot of the tool that is developed for automatic static test case generation from Class diagram and dynamic test case generation from the combination of Activity and state diagram of Online Hotel Reservation System. The system has been developed with help of Java and Net beans.

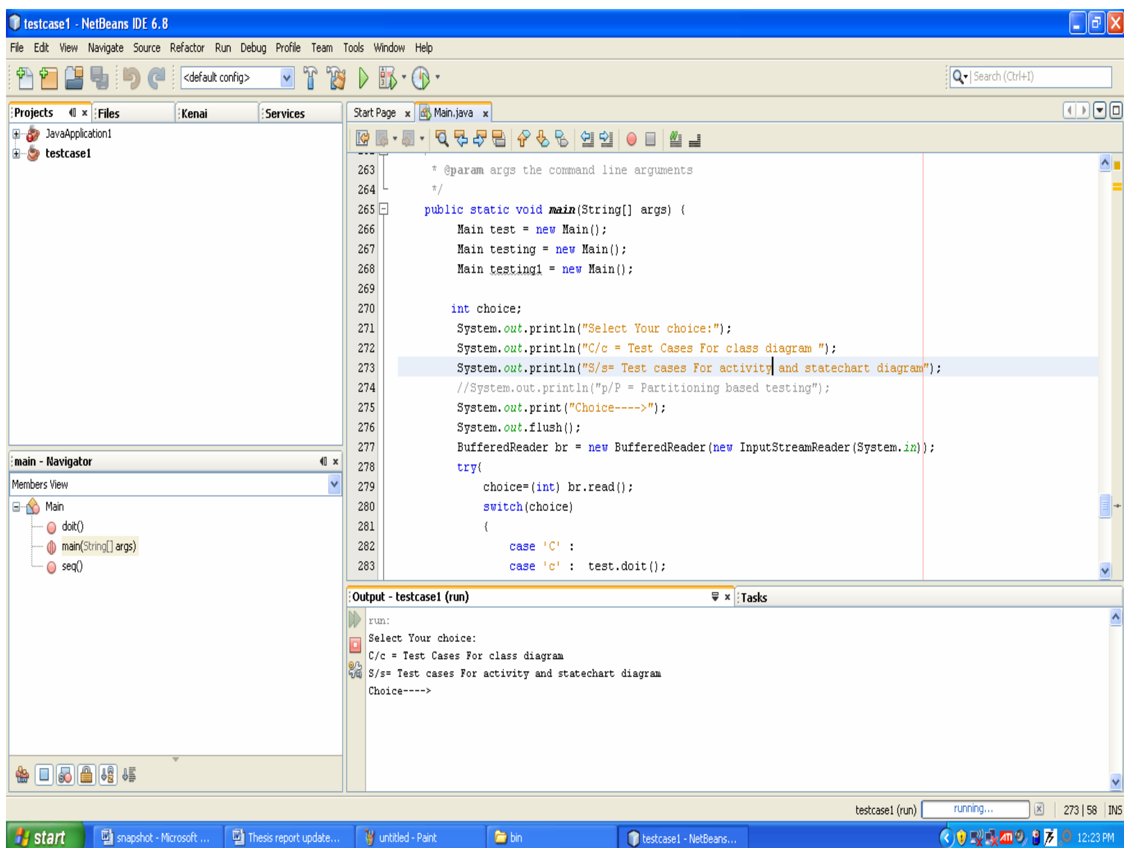


Figure 5.4: Snapshot of the Tool

The following figure shows options provided to us after running the tool developed with Netbeans. Option C/c generates test cases of Class diagram which are static test cases and option S/s generates the dynamic test cases from Activity and State diagram of system.

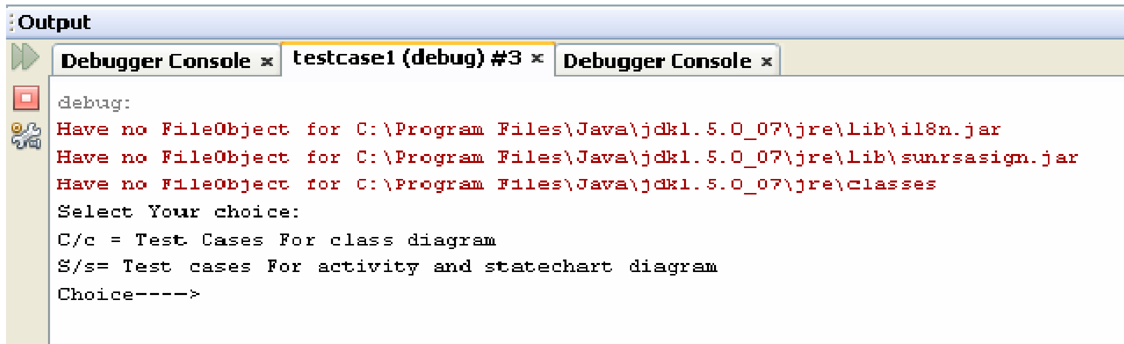


Figure 5.5: User Interface for Output

The following figure shows the exclusive test cases generated from Class diagram of Hotel Reservation System. The test cases are based on the operation, attributes, dependency, inheritance relationship among the different Classes in the Class diagram of Hotel Reservation System.

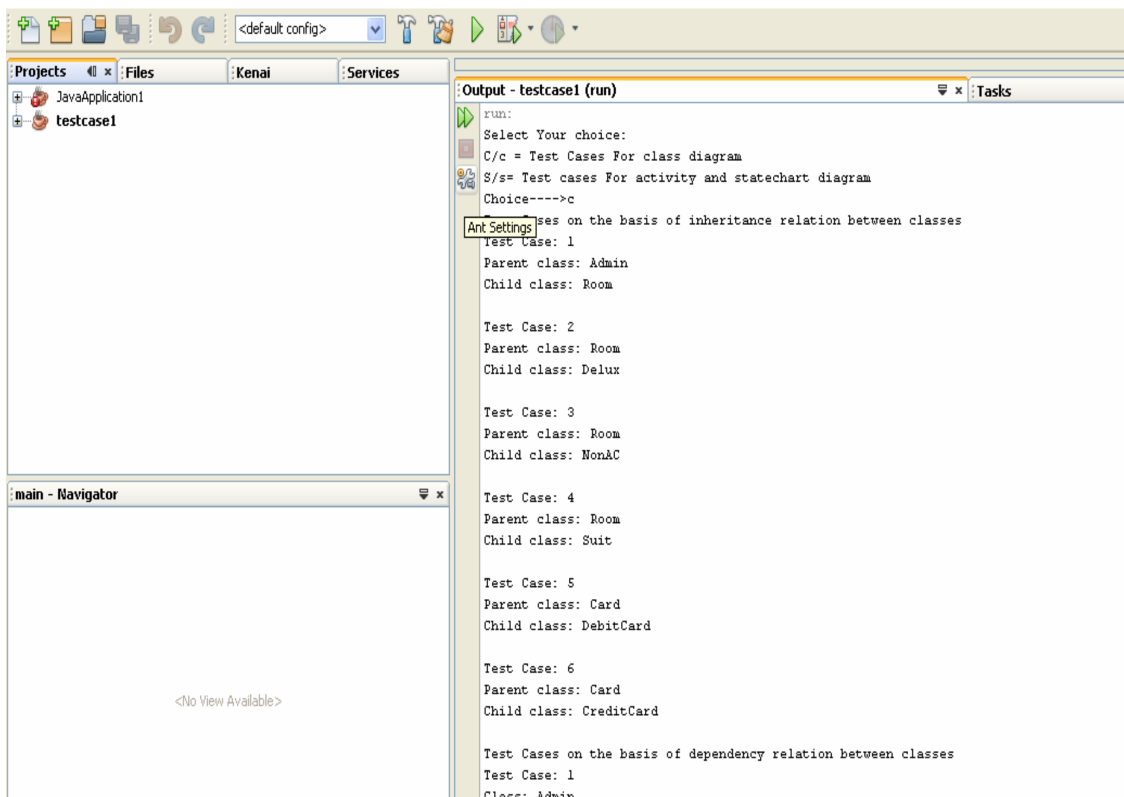


Figure 5.6: Test Cases Generation from Class Diagram

The following figure shows the exclusive test cases generation from Statechart diagram and Activity diagram. The case information from state diagram includes Initial states, Transitions, Final states, Pre conditions, Post conditions. case information from Activity diagram information are find out include Initials states, Final states, Guard conditions between activities

.

testcase1 - NetBeans IDE 6.8

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Debugger Console × | Debugger Console × | testcase1 (run) #3 ×

```
run:
Select Your choice:
C/c = Test Cases For class diagram
S/s= Test cases For activity and statechart diagram
Choice---->s
*** Populate
*** Populate
*** Close connection
*** Create an empty JoinRowset
*** Adding
*** Adding
*** Where clause: WHERE
(null.TRANSITION = null.GUARDCONDITION);
Type of join result:true


TEST CASE:1
PRECONDITION: [room_vacant>0]
INITIAL STATE: "RoomSearched"
TEST SEQUENCE:pre_user_search_room
POST CONDITION: post_room_available
FINAL STATE: "RoomSelected"

TEST CASE:2
PRECONDITION: [!valid_pin]
INITIAL STATE: "Credit_Card_Swapped"
TEST SEQUENCE:pre_pin_required
POST CONDITION: post_swap_card_again
FINAL STATE: "Credit_Card_Swapped"

TEST CASE:3
```

Figure 5.7: Test cases Generation from Statechart Diagram and Activity Diagram

Chapter 6
Conclusions and Future Scope

To perform model based testing, a language is needed that can deal with the design efficiently i.e. Unified Modeling Language. UML consists of different type of diagrams that are used to specify the static and dynamic aspects of software. UML diagram are used in object oriented approach, not in structural approach. In present work a tool is developed that extracts information from three UML diagrams and can generate test cases automatically. Class diagram based testing solve the problem of static test cases where Activity and Statechart diagram based testing solves the problem of dynamic test cases.

The proposed work is divided into several modules that perform automate test generation on UML model. The modules are petal file reader, SQL Loader and tests case generator.

6.1 Conclusions

- A new algorithm has been proposed to extract information from petal file of Class, Statechart and Activity diagram.
- Proposed techniques generates test case information from combination of various UML diagrams.
- Generated test case are more effective due to
    - Combination of three diagrams covers both static as well as dynamic aspect of software system
    - These test cases shows initial state, final state, guard condition, pre conditions, post conditions, test sequences, parent child relationships, dependencies.
    - Proposed techniques are much simpler as compare to graph based test case generation techniques.

6.2 Future Work

- Further work can be explored to use formal methods with UML diagrams.
- This technique further can be extended to include all types of software artifacts to predict changes for regression testing.
- Concept of overloading**,** polymorphism etc can be further used to generate automatic test cases.

References

1) Herzl, C. William, “The Complete Guide to Software Testing”, 2nd ed. Publication info: Wellesley, Mass.: QED Information Sciences, 1988. ISBN: 0894352423.Physical description: ix, pp. 280
2) Myers, J. Greenford, “The art of software testing”, Publication info: New York: Wiley, 1979. ISBN: 0471043281 Physical descriptions: xi, pp. 177.
3) Hans-Gerhard Gross “Component-Based Software Testing with UML” Springer-link Velar Berlin Heidelberg 2005 page 80-112
4) ***Grady Booch***, James Rumbaugh, Ivar Jacobson, The ***Unified Modeling Language User Guide***, Addison Wesley, Reading, Massachusetts, May 2005
5) John D. McGregor, David A. Sykes “A Practical Guide to Testing Object-Oriented Software”, Addison Wesley, March 05, 2001, pp. 167
6) The ***Build Security In*** (***BSI***) ***U.S.*** Department of Homeland Security https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html (2010).
7) TestPlan Group, www.testplant.com/download_files/BB_vs_WB_Testing.pdf 2010
8) B. Bezier. Black-Box Testing, Techniques for Functional Testing of Software and Systems. Wiley, New York, 1995
9) IEEE. Standard Glossary of Software Engineering Terminology, Volume IEEE Std. 610.12-1990. IEEE, 1999
10) R. Binder. “Testing Object-Oriented Systems: Models, Patterns and Tools”. Addison-Wesley, 2000.
11) B. Meyer. “Object-oriented Software Construction”. Prentice Hall, 1997.

12) The Testing Geek Group http://www.testinggeek.com/index.php/testing-types/system-knowledge/51-grey-box-testing 2010

13) The Object Management Group,www.omg.org/gettingstarted/what_is_uml.htm 2010**.**

14) The Developer Group www.developer.com/design/article.php/.../UML-Overview.htm

15) The Sparx System Group, www.sparxsystems.com/.../FCGSS_US_WP-

-Applying_4+1_w_**UML**2.pdf (2010)

16) Alejandra Cechich, Mario Piattini, Antonio Component-based software quality: methods and techniques Springerlink 2003 - Computers - 402 pages

17) Clay E. Williams **"**Software Testing and the UML**"** Center for Software Engineering, IBM T. J. Watson Research Center

18) B. Bezier. "Software Testing Techniques", Van Nostrand Reinhold, 2nd edition, 1990.

19) Gurpreet Singh and Rajesh Kumar Bhatia "UML Design Based Testing"ACM/IEEE 11th International Conference on Model-Driven Engg. Languages and Systems (Formerly UML)", Toulouse, France, 28 Sep- Oct-2010

20) P.Samuel, R. Mall, A.K.Bothra, "Automatic test case generation using unified modeling language (UML) state diagrams", IET Software, 2008, Vol. 2, No. 2, pp. 79–93/doi: 10.1049/iet-sen:20060061

21) KANSOMKEAT S., RIVEPIBOON W.: "Automated-generating test case using UML Statechart diagram". Proc. SAICSIT 2003, ACM 2003, pp. 296–300

22) KIM Y.G., HONG H.S., BAE D.H., ET AL.: "Test cases generation from UML state diagram", Proc. Software, 1999, 146, (4),pp. 187–192

23) P.Samuel, R.Mall "Slicing-Based Test Case Generation from UML Activity Diagrams" ACM SIGSOFT Software Engineering Notes Page 1 November 2009 Volume 34 Number 6

24) Yoonsik Cheon, Carmen Avila, "Automating Java Program Testing Using OCL and AspectJ," , pp.1020-1025, 2010 Seventh International Conference on Information Technology, 2010

25) R. S. Pressman. “Software Engineering: A Practitioner’s Approach”, 6th Edition, McGraw Hill, New York, 2005, pp. 424, 434, 449

26) V. Martena, A.Orso, M.Pezzμe “Interclass Testing of Object Oriented Software” Technical Report GIT-CC-02-28, May 2002

27) M. Dahm ,Grammar and API for Rational Rose Petal files, July 19,2001

28) Jonathan Gennick and Sanjay Mishra, “ Oracle SQL*Loader The Definitive” Guide, O’Reilly & Associates, Inc., April 2001.

List of Papers/Publications

1) Rohit Kumar and Rajesh Bhatia, "Class Diagram and Interaction Diagram Based Software Testing" ISSRE 2010 21st annual International Symposium on Software Reliability Engineering, November 1-4-2010 San Jose, CA USA(Communicated).

2) Rohit Kumar and Rajesh Bhatia, "Interaction Based Software Testing" SCAM 2010, 10th International Working Conference on Source Code Analysis and Manipulation, 12 September, Timisoara, Romania(Communicated).