BEHAVIOR BASED AUTOMATED TEST CASE GENERATION

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

Master of Engineering
in
Software Engineering

By:
Rohin Verma
(800831011)

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

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2010

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Software Engineering"**, 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.

Rohin Verma

(Rohin Verma)

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

10/07/10

(Dr. Rajesh Kumar Bhatia)
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(RAJESH BHATIA)
Head
Computer Science & Engineering, Department
Thapar University
Patiala

R.K.Sharma 16.7.10

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

Acknowledgment

I wish to express my deep gratitude to Dr. Rajesh K. Bhatia, Assistant Professor & Head, Computer Science & engineering department, TU, Patiala for providing his uncanny guidance and support throughout the thesis.

I would also like to thank all my colleagues, who have given me moral support and their relentless advice throughout the completion of this work. I also thank all the staff members of software lab and Computer Science and Engineering Department, Thapar University, Patiala, for their continuous assistance.

I wish to express indebtedness to my parents who have been constant source of love and encouragement.

Finally, I would like to thank God for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Rohin Verma
800831011
M.E.(Software Engineering)-2nd year
Computer Science & Engineering Department
Thapar University
Patiala –147004

Abstract

Test case generation is the most important part of the testing process. Most of the software fails because the test cases written are not able to detect errors. Generally testing is performed on requirements or the code, while the design is seldom concerned but in present work design based test case generation is proposed.

Present work uses petal files, also called mdl files of Class diagram, Sequence diagram & State chart diagram to generate test cases. Class diagram can give information of static view of system. Sequence diagram can give information of time ordering of messages. Statechart diagram can give information of different states of system. Tool reads petal files of Class diagram, Sequence diagram & State chart diagram one by one and creates text files. Text files contain information extracted from Class diagram, Sequence diagram & State chart diagram. Then tables are created in database and using SQL *Loader all data from text files are entered into database. Finally using Netbeans IDE SQL queries are written in java to generate test cases from the data stored in database.

In present work design is used for test case generation. Using design for test case generation helps to plan test cases early. Due to this most of the errors can be stopped and eliminated before going to next phase. This also leads to reduction of the effort required to remove errors at later stages. Generating test cases automatically from design is more realistic as compared to requirements or any other artifacts. Design based test cases cover both static as well as dynamic aspect of the system.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

IEEE definition of Software Testing is the process of executing a program or system with the intent of finding errors. [1] Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. [2]

Software is like other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed and reasonably small set of ways. By contrast, software can fail in many bizarre ways. Almost 50%-70% of the software projects fail due to the improper or inefficient testing. So it is very important to test the software. Testing software is more complex than exercising a program to see if it works. Software testing is an empirical technical investigation conducted to provide stakeholders with the information about the quality of the product or service under test, with respect to the context in which it is intended to operate. Quality is not an absolute term; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behavior of the product against a specification.

Testing is performed by independent group of testers after the functionality is developed and before it is shipped to the customer. This practice often results in the testing phase being used as buffer to compensate for delays in the projects, thereby compromising the time devoted to the testing. Another approach is to start software testing at the same moment the project starts and it is a continuous process till project ends.

1.1 Testing Objectives

The main objective of testing is to prove that the software product as a minimum meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances. There are two components to this objective. The first component is to prove that the requirements specification from which the software was designed is correct. The second component is to prove that the design and coding

correctly respond to the requirements [3]. Correctness means that function, performance, and timing requirements match acceptance criteria. Software testing is further complicated by the fact that system acceptance criteria usually involve hardware, procedures, and operators so that acceptance tests involve more than just the software. Software tests are designed to force failures. In that regard, software testing is intrinsically destructive.

Following are the objectives that software testing follows:

- Testing is a process of executing a program with the intent of finding an error.
- Testing cannot show the absence of defects, it can only show that software errors are present.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.

Objective of testing is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort. If testing is conducted successfully it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to the specifications and that performance requirements appear to have been met. In addition, data collected as testing is conducted provides a good indication of software quality as a whole. But there is one thing that testing cannot do:

- Testing cannot show the absence of defects, it can only show that software errors are present**.**
- Testing do not remove errors, debugging is used to remove and to find nature of errors.

1.2 Testing Principles

Following are principles of Software Testing [4]:

- All tests should be traceable to customer requirements. The objective of system testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can

begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- Testing should begin “in the small” and progress toward testing “in the large”. The first test planned and executed generally focus on individual program modules. As testing progresses, testing shifts focus in an attempt to find errors in integrated clusters of modules and ultimately in the entire system.
- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.
- To be most effective, testing should be conducted by an independent third party. By “most effective “, means testing that has the highest probability of finding errors. For this reason, the software engineer who created the system is not the best person to conduct all tests for the software.

1.3 Characteristics of a “Good” Test

Following are the characteristics of a good test [4]:

- A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.
- A good test should be “best of breed”. In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.4 Software Testing Techniques

1.4.1 Static Testing

The term static testing refers to testing the software requirement specification, software design specification and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through, etc [3]. Static testing is employed to verify the correctness of requirements, designs, and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards. A successful static test of a software module depends upon several things going right:

- A correct allocation of requirements to the software components.
- A correct partitioning and sub allocation of software requirements to the module.
- Successful/correct module design.
- A successful translation of the intermediate code i.e. pseudo-code, POL, etc. into programming language statements. However, true representative test cases must be successfully executed before the testing and integration team certifies a software module. Usually this step is not a part of static testing.

The purposes of the static testing are:

- Validating the requirement specifications.
- Looking for omissions, inconsistencies, redundancies in all documents and source code.
- To ensure that the documents of design and coding conform to the specification

1.4.2 Dynamic Testing

Dynamic testing is a term that describes the development of test cases and test procedures, the execution of test cases, and the structure and use of test logs and anomaly or incident reports. There are three popular ways to perform dynamic testing, namely, black box testing, grey box testing and white/glass box testing. Either of these methods requires a set of well-developed and well-structured test cases.

Dynamic testing cannot prove the absolute correctness of a software product unless it is performed in an exhaustive manner. An exhaustive test requires a set of test cases that guarantees the following: explicitly exercises every possible, module

path and every possible combination of paths with every possible module input and every possible combination of module inputs.

1.5 Types of Testing

Testing is generally performed by three ways: white- box testing, black box testing and gray-box testing.

White-box testing is testing that takes into account the internal mechanism of a system or component (IEEE, 1990). White-box testing is also known as structural testing, clear box testing, and glass box testing [3]. The connotations of “clear box” and “glass box” appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code. Using the white-box testing techniques outlined, a software engineer can design test cases that (1) exercise independent paths within a module or unit; (2) exercise logical decisions on both their true and false side; (3) execute loops at their boundaries and within their operational bounds; and (4) exercise internal data structures to ensure their validity [5].

Black-box testing is also called behavioral based testing, focuses on functional requirements of the software. That is, black box testing enables the software engineer to derive sets of input conditions that will fully exercise functional requirements for a program. Black box testing attempts to find errors in: (I) Incorrect or missing functions, (2) Interface errors, (3) Errors in data structures or external database access, (4) Behavior or performance errors, (5) Initialization or termination errors. [5].

Gray-box testing method, which was proposed in recent years [6] in the designer’s viewpoint, generates test cases based on high level design models which represent the expected structure and behavior of the System under test. The design specifications are the intermediate artifact between requirement specification and final code. They preserve the essential information from the requirement, and are the basis of the code implementation. Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model, which describes the expected behavior of an operation. It can find problems, which used to be ignored by both black and white method.

1.6 Test Case

During testing generally test cases are generated. According to IEEE definition of Test case is (IEEE Std 829-1983) “A Documentation specifies inputs, predicted results, and a set of execution conditions for a test item”. [3] defines a test as “A sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. The word ‘test’ is used to include subtests, tests proper, and test suites. Test case is a mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle. Typical written test case format is:

- Test case ID.
- Test case description.
- Test step or order of execution number.
- Related requirements like preconditions and post conditions.
- Depth.
- Test category.
- Author.
- Check boxes for whether the test is automatable and has been automated.

Additional fields that may be included and completed when the tests are executed:

- Pass/fail.
- Remarks.

1.7 Object Oriented Testing

Object-oriented programming is build upon a sound engineering foundation whose elements are collectively called as the object model. The object model encompasses the principles of abstraction, data hiding, inheritance, overloading, and polymorphism. The important feature of object model is that these elements are brought together in a synergistic way. Object-oriented programming is defined as a method of implementation in which programs are organized as a cooperative collection of classes, each of which represent an instance of some class, and whose classes are all members of

a hierarchy of classes united via inheritance relationship. The Object-oriented programming use objects as basic building blocks, where each object is an instance of some class, and classes are related to each other through inheritance. There are some concepts that are extensively used in Object-oriented programming like: Objects, classes, abstraction, encapsulation, data hiding, inheritance, overloading, polymorphism, dynamic binding, and message passing. The object-oriented software has series of layered subsystems that encapsulate collaborating classes. Each of these system elements i.e. subsystems and classes performs functions that help to achieve system requirements. It is necessary to test an OO system at a variety of different levels to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Testing object-oriented software with the features of inheritance and polymorphism presents new technical challenges for the testers. Thus object-oriented software needs different tactics for testing software effectively during all phases of a development effort. Testing OO program begins with the review of analysis and design models. Once code has been generated, OO testing begins in the small with a series of tests designed to exercise class operations and examine whether errors exists as one class collaborates with other classes. As classes are integrated to form a subsystem, use-based testing, along with fault based testing approaches, is applied to fully exercise collaborating classes. Finally, use-cases are used to uncover errors at the software validation level. Object-Oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

1.8 UML Based Testing

Software testing is a widely used and accepted approach for verification and validation of a software system, and it can be regarded as the ultimate review of its specification, design, and implementation. Testing is applied to generate modes of operation on the final product that show whether it is conforming to its original requirements specification, and to support the confidence in its safe and correct operation [7, 8]. Appropriate testing should be primarily centered on requirements and specification not on code, which means that testing should always aim to show conformance or non-

conformance of the final software product with some requirements or specification documents. Source code provides a great deal of information to guide the testing efforts according to testing criteria [3], but it cannot replace specification documents as a basis for testing. This is because code is a concrete representation of abstract requirements and design documents, and testing is supposed to show conformance of the concrete implementation with the abstract specifications. Testing based merely on source code documents shows that the tested program does what it does, but not what it is supposed to do. Due to this UML diagrams are used for test case generation

The Unified Modeling Language/UML is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such systems. It is intended for use with all development methods, lifecycle stages, application domains, and media. The Unified Modeling Language has received much attention from academic software engineering research and professional software development organizations. It has almost become a de-facto industry standard in recent years for the specification and the design of software systems. Due to this reason UML diagrams can be used for testing. This type of testing comes under the category of gray-box testing. Testing activities that are based on UML models, or use models, are becoming increasingly popular. There are number of publications that have been emerging over the last few years [9-13], that are based on UML models. UML models represent specification documents which provide the ideal bases for deriving tests and developing testing environments. A test always requires some specification, or at least a description or documentation of what the tested entity should be, or how it should behave. Testing that is not based on a specification is entirely meaningless. Even code-based, or so-called white box testing techniques, that initially only concentrate on the structure of the code, are based on some specification. The code is used only as a basis to define input parameter settings that lead to the coverage of distinct code artifacts. Models are even more valuable if UML tools that support automatic test case generation are used. The technique used to derive test cases from UML diagrams is called Model based testing.

1.8.1 Model Based Testing

Model-Based Testing with the UML is concerned with deriving test information out of UML models. In other words, the models provide the primary information for developing the test cases and test suites, and for checking the final implementation of a system.

The most fundamental difference between traditional and more modern model-based approaches is probably the type of notations upon which testing is based. Traditionally, test cases have been derived from very abstract specifications, based on natural language at one extreme of the notations spectrum, and from the source code at the other extreme of this spectrum. Natural language is not very suitable as a basis for testing since it is not formal enough to derive all required testing artifacts. Source code is not really valuable any more if component-based development has been considered. Component-based development will not allow to look at the implementations of most components. Models on the other hand are extremely useful for deriving all kinds of test artifacts at all levels of decomposition and abstraction, and they support all test development phases well.

The UML provides diagrams according to the different views that a system has. 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. In essence, UML diagrams specify what a system should do, how it should behave, and how it will be realized. The entirety of all UML models therefore specifies the system completely and sufficiently.

1.9 Motivation & Objective

Test cases are usually generated from the requirement or the code while the design is seldom concerned; the present work proposes an approach to generate test

cases directly from UML Class diagram, Sequence diagram, and Statechart diagram using gray-box method, where the design is reused to avoid the cost of test model creation. Individual diagram can be used for test case generation i.e. class diagram or Statechart diagram or Sequence diagram. Test cases generated from individual diagram depends only on one view, like class diagram represents static view. Test cases generated from class diagram represent static view of test cases only. In present work test cases are generated from combination of class diagram, sequence diagram, statechart diagram. Combining different views to generate test cases more produce concrete test cases than individual view generated test cases. 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/viability to generate test cases.
- To design and develop a new test case generation approach which is based on UML Class diagram, Sequence diagram, and Statechart diagram.
- To develop a tool that generates test cases automatically from petal files/mdl files of UML Class diagram, Sequence diagram, and Statechart diagrams.

1.10 Organization of Thesis

This report is divided into six chapters. Chapter 1 gives brief introduction about testing, object oriented testing, UML design based testing. Objectives of the proposed work are also considered here. Chapter 2 gives literature review where work related to class diagram, sequence diagram, statechart diagram has been discussed. Chapters 3 specify problem statement. This chapter discusses what is there in old system and what is new in proposed work. Chapter 4 gives information of proposed system. This section explains methodology and design of proposed system. Chapter 5 describes working of a tool with the help an example and shows experimental results. An example of SES/Student Enrolment System has been discussed in chapter 5 and results i.e. test cases generated with the help of tool have been shown in figures. Chapter 6 focuses on conclusion and future scope.

Chapter 2
Literature Review

A literature review is a body of text that aims to review the critical points of current knowledge on a particular topic. Literature work regarding test case generation from class diagram, sequence diagram and statechart diagram is mentioned below.

2.1 Class Diagram Based Testing

The fundamental unit of an object-oriented program is a class. Class testing comprises those activities associated with verifying that the implementation of a class corresponds exactly with the specification for that class. If an implementation is correct, then each of the class's instances should behave properly. Class testing is roughly analogous to unit testing in traditional testing processes and has many of the same problems that traditional testing has been facing. Class testing must also address some aspects of integration testing since each object defines a level of scope in which many methods interact around a set of instance attributes.

2.1.1 Class Diagram Related Work

This has been 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. John D. McGregor, David A. Sykes [14] presented a work to generate test cases on the basis of class diagram. This has been assumed that the specification is expressed in a specification language such as the Object Constraint Language/OCL [15] or a natural language, and/or as a state transition diagram. If more than one form of specification is used for a class, it has been assumed that 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. They developed test cases on the basis of class operations. Pre condition and Post condition for a particular operation is expressed in the form of OCL and test cases have been generated. They also used an approach for the execution-based testing of classes in an inheritance hierarchy. They analyzed the inheritance relationship from a testing perspective to identify what needs to be tested in a subclass.

[5] Has presented work based on generating test cases based on class. They presented random test cases generation based on Object oriented classes, in these

test cases had been generated from combination of operations of a class. They divided operations in two parts, firstly those operations that can be used only once in class and secondly those operations that can be used multiple times in a class. Applying permutations on second type of operations and using first type of operations singly test cases had been generated. They also presented Partitioning Testing at Class Level. Three types of partitioning is used: State based partitioning, Attribute based partitioning, Category based partitioning.

- State Based Partitioning: It categorizes class operations based on their ability to change the state of the class. All operations that changes the state are called state operations and those do not change the state are called non state operations.
- Attribute based partitioning: It categorizes class operations based on the attributes that they use. Operations are divided into three categories: (1) Operations that use a particular attribute, (2) Operations that modify a particular attribute, (3) Operations that do not use a particular attribute.
- Category based partitioning: It categorizes class operations based on the generic function that each performs. Operations are categorized as: (1) Initialization operations, (2) Computation operations, (3) Termination operations.

Kirani and Tsai [16] suggested "inter class test case generation" approach. Like testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario based testing and behavioral testing. The approach for multiple class testing or inter class testing is similar to the approach used for partitioning testing of individual classes. However, the test sequence is expanded to include those operations that are invoked via a message to collaborating classes.

2.2 Sequence Diagram Based Testing

Sequence diagram along with collaboration diagram comes under Interaction modeling. It represents a combination of dynamic and structural modeling. It concentrates mainly on the dynamic interactions between instances.

Sequence and collaboration diagrams are typical control flow diagrams, although with slightly different foci. As the term interaction diagram implies, they concentrate on control flow through multiple interacting instances. For testing, the two

diagram types they may be represented as abstract control flow graphs that span multiple entities. With that respect all typical traditional control flow graph-based test coverage criteria can be applied as outlined in [17]. This includes path and branch coverage criteria as well as more exotic test case selection techniques such as round-trip scenario coverage [18]. Since UML diagrams are always also more abstract than traditional control flow graphs, the test targets may be more abstract.

2.2.1 Sequence Diagram Related Work

Philip Samuel, Rajib Mall and Sandeep Sahoo [19] presented a novel testing methodology to test object-oriented software based on UML sequence diagrams. This paper is based on dynamic slicing technique and generates test cases automatically from UML sequence diagrams. Firstly the message guards on sequence diagrams were identified and dynamic slices with respect to each conditional predicates were created. The test data was generated with respect to the slice. A test adequacy criterion named slice coverage criterion is also used to validate the test cases generated. This paper approach achieves adequate test coverage without unduly increasing the number of test cases. This paper also describes how dynamic slicing is used in testing.

Li Bao-Lin, Li Zhi-shu, Li Qing and Chen Yan Hong [20] presented a new test cases generation approach that is based on UML sequence diagrams and Object Constraint Language/OCL. In this approach, a tree representation of sequence diagrams is constructed. Firstly traversal of the constructed tree for selecting conditional predicates from the sequence diagram is carried out. Then, post- and precondition is described by OCL. They transform the conditional predicates on the sequence diagram and apply function minimization technique to generate the test data. The generated test cases achieve message paths coverage and constraint attribute coverage of all objects which relate to the message. Lastly, the results of experiments show that this method has a better performance.

Monalisa Sarma, Debasish Kundu, Rajib Mall [21] presented a novel approach of generating test cases from UML design diagrams. Their approach consists of transforming a UML sequence diagram into a graph called the sequence diagram graph and augmenting the SDG nodes with different information necessary to compose test vectors. These information are mined from use case templates, class diagrams and data

dictionary. The SDG is then traversed to generate test cases. The test cases thus generated are suitable for system testing and to detect interaction and scenario faults.

Binder identifies some typical problems that may be discovered through sequence diagram-based testing [18]:

- Incorrect or missing output.
- Action missing on external interface.
- Missing function/feature (interface) in a participating object.
- Correct message passed to the wrong object.
- Incorrect message passed to the right object.
- Message sent to destroyed object.
- Correct exception raised, but caught by the wrong object.
- Incorrect exception raised to the right object.
- Deadlock.
- Performance.

2.3 Statechart Diagram Based Testing

The UML supports behavioral modeling through statechart diagrams and activity diagrams. Statechart diagrams represent the behavior of an object by specifying its responses to the receipt of events. Statecharts are typically used to describe the behavior of class or component instances, but they can also be used to describe the behavior of use cases, actors, or operations.

State-based testing concentrates on checking the correct implementation of the component's state model. The test case design is based on the individual states and the transitions between 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. In that instance, there is no test case without the notion of a state or state transition. In other words, pre and post conditions of every single test case must consider states and behavior.

2.3.1 Statechart Diagram Related Work

P. Samuel R. Mall A.K. Bothra [22] developed a novel method to automatically generate test cases based on UML state models. They present an approach in which, the control and data flow logic available in the UML state diagram to generate

test data are exploited. The state machine graph is traversed and the conditional predicates on every transition are selected. Then these conditional predicates are transformed and function minimization technique is applied to generate test cases. They present test data generation scheme which is fully automatic and the generated test cases satisfy transition path coverage criteria. The generated test cases can be used to test class as well as cluster-level state-dependent behaviors.

Supaporn and Wanchai [23] proposed the automatic testing technique to solve partially the testing process. This technique can automatically generate and select test cases from UML statechart diagrams. Firstly, Statechart diagram is transformed into intermediate diagram, called Testing Flow Graph, explicitly identify flows of UML statechart diagrams and enhance for testing. Secondly, from TFG test cases were generated using the testing criteria that is, the coverage of the state and transition of diagrams. Finally, the evaluation is performed using mutation analysis to assess the fault revealing power of our test cases.

Binder [18] 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. He proposed the following criteria for test case generation from statechart diagram:

Piecewise Coverage: Piecewise coverage concentrates on exercising distinct specification pieces, for example, coverage of all states, all events, or all actions. These techniques are not directly related to the structure of the underlying state machine that implements the behavior, so it is only incidentally effective at finding behavior faults. It is possible to visit all states and miss some events or actions, or produce all actions without visiting all states or accepting all events.

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.

Round-trip Path Coverage: Round-trip path coverage is defined through the coverage of at least every defined sequence of specified 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.

Chapter 3
Problem Statement

Problem statement describes the gap in the existing work and problem formulation. The Gap in existing work shows, what are the limitations 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 **"A Practical Guide to Testing Object-Oriented Software"** [14] test cases has been generated on the basis of class diagram using OCL, 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 **"Inter Class Test Case Generation"** [5] 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 test cases have been generated from only class diagram. Test cases generated are not effective and not efficient as test cases depend on only static view of system. 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 Sequence Diagram's Existing Work

In **"UML Sequence Diagram Based Testing Using Slicing" [19]** this method concentrates on dynamic slice in sequence diagram and uses that slice to generate test cases automatically from UML sequence diagram. But these dynamic test cases depend upon variable value and its change affect with respect to the message sequence.

In **"Test Case automates Generation from UML Sequence Diagram and OCL Expression" [**20] Sequence diagram and Class Diagrams are used to generate test cases along with OCL, but the test cases generated are not effective and efficient. Test cases tell about class, operations, attributes, data limits and objects. Where class,

attributes, operations are taken from class diagram, data limit from OCL and objects from Sequence diagram. These test cases do not tell about pre conditions and post conditions, states of the system.

In **"Automatic Test Case Generation from UML Sequence Diagrams"** [21] sequence diagram is used to generate test cases. Test cases generated shows input, output, pre condition and post condition only. Input state and output state and cardinalities related test cases are not shown.

3.1.3 Gap in Statechart Diagram's Existing Work

In **"Automatic Test Case Generation Using Unified Modeling Language (UML) State Diagrams"** [22] 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 Pre condition, Post condition and class related data is not present in these test cases.

In **"Automated-Generating Test Case Using UML Statechart Diagrams" [23].** They generate test case using the testing criteria that is the coverage of the state and transition of diagrams. Test cases generated depends on only transitions and guard conditions. Test cases do not tell about initial state and final states of a transition. Although test cases are automatically generated but still these are not effective and efficient. These test cases depends on only dynamic aspect of system, static aspect has not been considered.

In ***"Testing Object-Oriented Systems: Models, Patterns and Tools"*** **[18]** 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 Problem Formulation

Generally testing is performed on coding part, which contains variables, loops, executable statements, classes, functions etc. This type of testing is quite tedious 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 Basis Path Testing it has two predicates, thus its cyclomatic complexity will be three and it has three 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 four and has four 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. Coding review is performed manually by individuals, 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 generation 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 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 only 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, state chart, 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. An automated testing performed on UML design is difficult. Automated testing means automatically generate test cases from deign.
- How to combine three UML diagrams 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-types of behavior i.e. static and dynamic. So finding static test cases and dynamic test cases that are optimal solution.

Above said problems are addressed in proposed work. In the proposed system, test case generation has been performed on three diagrams i.e. class diagram, sequence diagram and statechart diagram. Proposed system focus on these three diagrams and with the help of combination of these diagrams an effective and better test cases have been generated than previous approaches which are based on either single UML diagram or two UML diagrams.

3.3 Justification

Lot of work has been done on UML diagram based testing. Most of the work has been done on single UML diagram i.e. test cases have been generated from only one type of UML diagram at a time out of nine diagrams. Test cases generated by single diagram covers only one aspect of UML, static or dynamic. Some work has been done on combination of two diagrams. Major work is based on graphical parsing of UML

diagrams which is very complex way. Proposed work extends this approach and use combination of three diagrams to generate test cases. Proposed work is based on petal file also called MDL File. Tool reads petal files of three diagrams rather than graphical parsing.

The proposed work has advantage of generating test cases from UML diagrams rather than code. Earlier code is used to generate test cases, but code that contains loops, executable statements, variables which changes continually causes change in the number of test cases. So coding reviews has to be performed regularly to know all the functions that are affected by this change. This is very complex task. Also coding is later phase in Software Development Life Cycle. If there is any fault in initial phase of development like requirement or design. It will be multiplied and becomes very large in count at the time of coding and testing. So a lot of effort must be spent to correct that fault or error. If in the initial phase of development, the error has been found then it will not be multiplied. It leads to lesser cost and lesser time spent on fixing that errors.

Chapter 4
Proposed System and Implementation

Generally test cases have been generated from individual UML diagram, but the test cases generated by single diagram are not efficient and effective. These test cases generated depend on single view. If combination of diagrams has been used then test cases generated will be more efficient and effective as they will cover both static and dynamic aspect of system. Thus the proposed system focuses on the three UML diagrams.

- Class Diagram
- Sequence Diagram
- Statechart Diagram

Various types of information can be extracted from these diagrams to generate test cases. For static information class diagrams are used and for dynamic type of information sequence and statechart diagrams are used.

4.1 Test Case Information Extraction

4.1.1 Information Extracted From Class Diagram

Class diagram is used to get the following information: -

- Classes
- Relationship between classes/Association
- Dependency
- Generalization/ Parent-child relationship
- Operations
- Cardinality
- Attributes

4.1.2 Information Extracted From Sequence Diagram

Sequence diagram is used to get the following information: -

- Object interaction i.e. messages sent to other objects.
- Pre-condition for a particular message.
- Post-condition for a particular message.

4.1.3 Information Extracted From Statechart Diagram

Statechart diagram is used to get the following information: -

- Initial state of a system
- Final state of a system
- Guard conditions
- Transitions

4.1.4 Combination of class, sequence and statechart diagrams

Combination of class, sequence and statechart diagrams means all the required information from each diagram is extracted to generate test cases for a particular system.

4.2 Methodology

The major steps of methodology are given in fig 4.1. Following are the major steps in prepared technique:

- Input design Code Petal File:

  The Petal file of class diagram is given as input to the developed tool.
- Read petal file:

  Petal file is read by the tool and string matching mechanism is used to find a pattern like class name, class attributes, class cardinality, class operations, inheritance, dependency etc.
- Is pattern found:

  If pattern is found then it is entered to the queue else next pattern has been found.
- Store it in a queue:

  All patterns found are 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.
- Search for another pattern:

  Tool searches for various patterns in petal file. For example if *class name* is found than *class name* is it entered into the class name queue Else if *class attributes* is found tool enters it to the class attributes queue similarly so on.
- Is EOF (End of File):

  Tool keeps on searching the patterns until EOF is reached.

- 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, Oracle load data files feature, loads all the data from the text file to the oracle database.
- Are all Petal files input:

  After class diagram, Petal files of sequence diagram and statechart diagram are entered to the tool and loop back to second step in flow chart.
- 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 Netbeans IDE tool fires queries written in java to generate test cases.

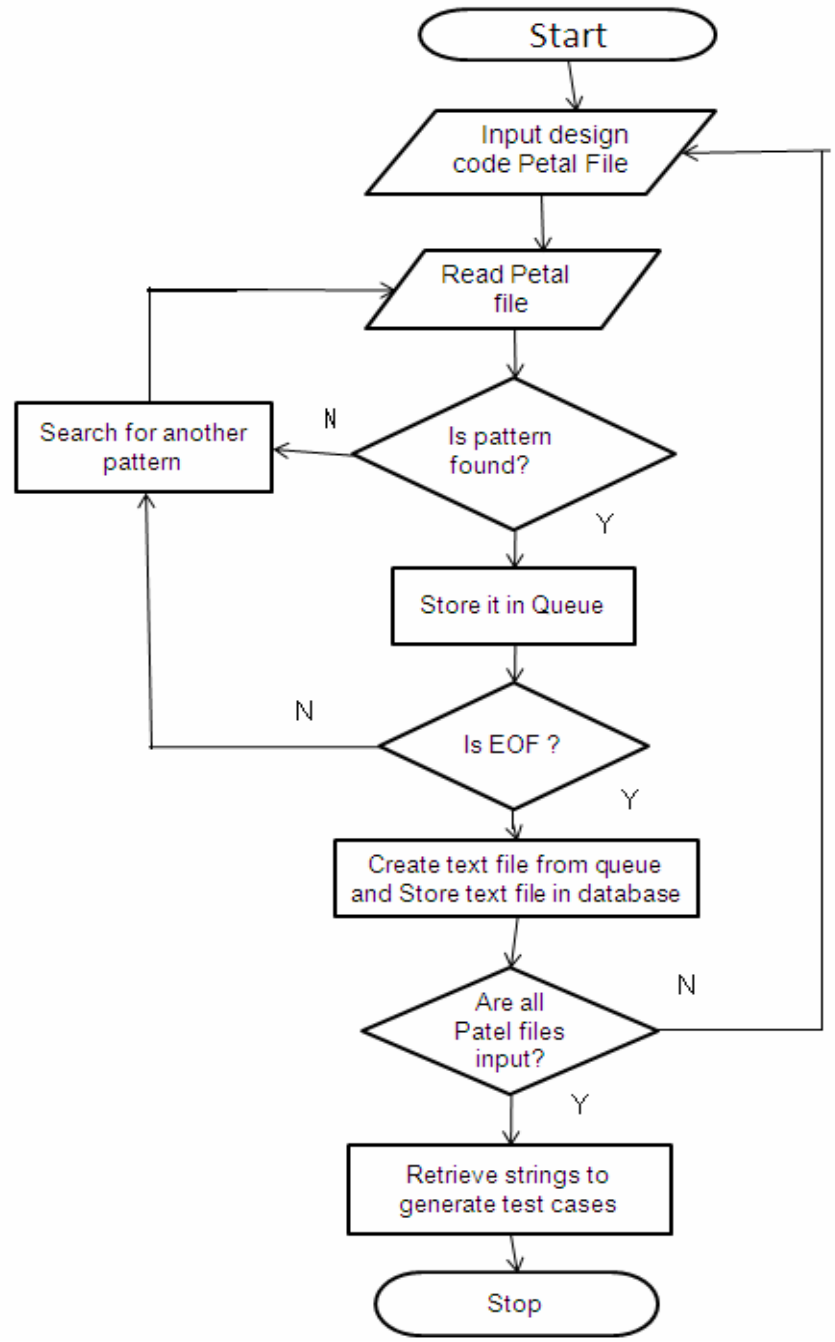

Fig 4.1 Flow Chart for Methodology Used

4.3 Description of Implementation

Draw class diagram and statechart diagram for a particular system using Rational Rose. Class diagram has classes, which in turn have operations and attributes. For a particular operation of a class in a class diagram, draw a sequence diagram. These diagram's petal files (mdl files) are given as input to the tool. First of all the class

diagram's petal file is entered to the tool. Fig 4.2 shows the Petal File of class diagram. Red colored words are class name, orange colored letters are Dependency relation, blue colored words are class attributes, dark red colored words are inheritance relationship and green colored are class operations. Tool reads the petal file line by line. It tokenizes the line into words and then matches the string with the pattern to find the class name, class attributes, class cardinality, class operations, inheritance, and dependency.

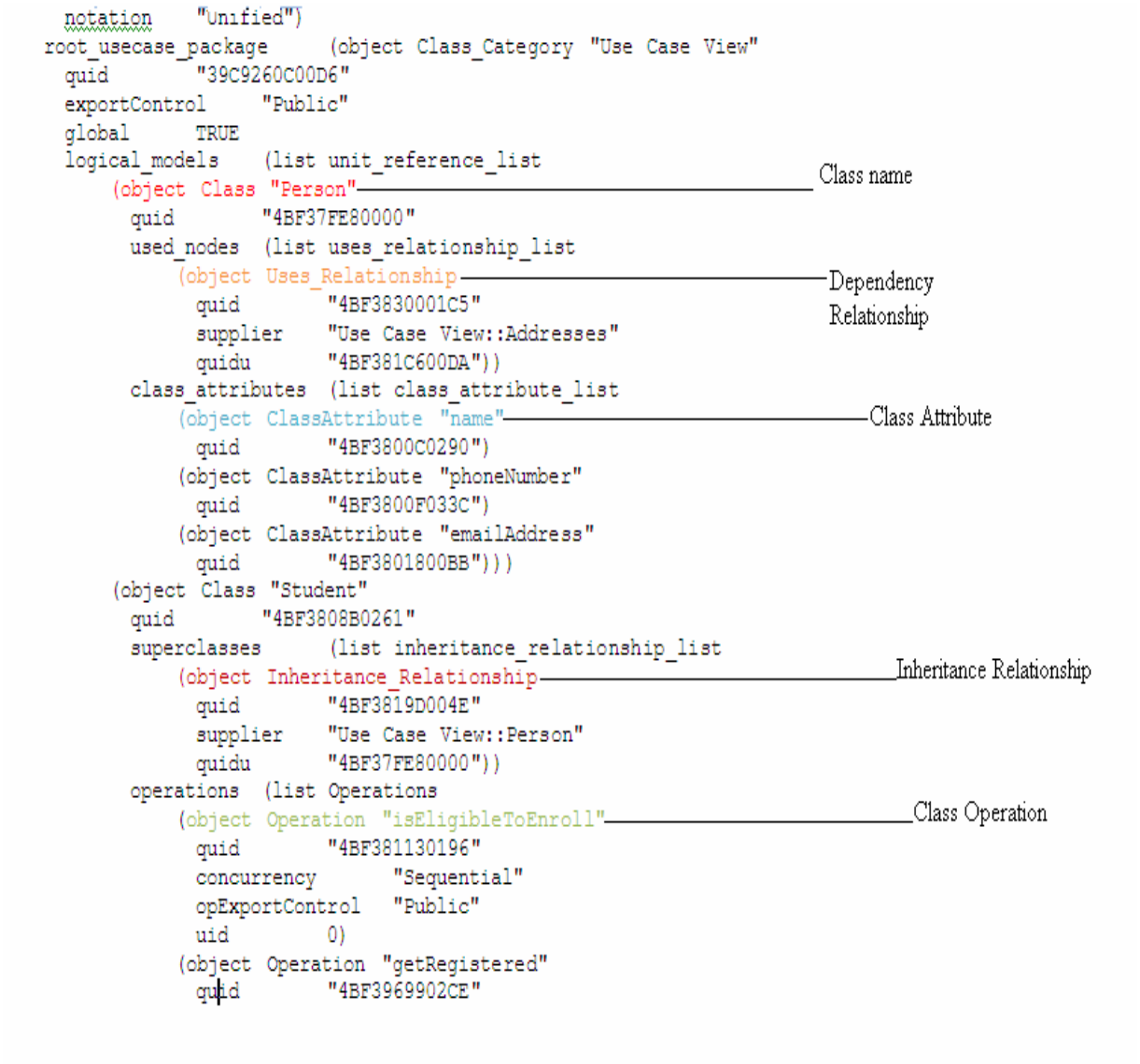

Fig 4.2 Petal File of Class Diagram

When class name is found it enters it to the queue named class name queue and searches for another pattern i.e. its attributes, its operations, its inheritance classes, its dependency, its cardinality and every string is entered to queue of a particular

type. There are 6 queues for class diagram Petal file one for class name, attributes, operations, inheritance classes, dependency, and cardinality. Tool keeps on reading the file and searching the pattern until EOF (End of File). As EOF is reached, two text files are generated by tool that contains information from class diagram. First one contains information about class name, class attributes, class operations, inheritance, dependency and second one contains class cardinality information. Similarly the Petal files of sequence diagram and statechart diagram are read by the tool. From sequence diagram Pre-conditions, Post-conditions and messages send to other objects are searched by the tool, and from statechart diagram initial state, final state, guard conditions are searched by the tool. Once searching is over all the data in the form of text file is generated by the tool. Text file of sequence diagram contains information in the form of tuples as Pre-condition, Post-conditions and messages. Text file of statechart diagram contains information in the form of tuples as guard conditions, initial state, and final state. Now tables are created in Oracle. The sql to create table for class data is shown in fig 4.3, and for class cardinality data is shown in fig 4.4, Similarly SQL to create table for sequence diagram and statechart diagram is shown in fig 4.5 and 4.6.

```
create table class1(
Classname   VARCHAR(20),
Operation   VARCHAR(20),
Attribute   VARCHAR(20),
Dependency  VARCHAR(20),
Inheritance  VARCHAR(20));
```

Fig 4.3 SQL to Create Table for Class Data

```
create table classassociation2(
Classname          VARCHAR(20),
cardinality        VARCHAR(10),
Associationclass   VARCHAR(20),
Ass_cardinality    VARCHAR(21));
```

Fig 4.4 SQL to Create Table for Class Cardinality Data

```
create table sequence1(
Precondition       VARCHAR(250),
Testsequence       VARCHAR(200),
Postcondition      VARCHAR(250));
```

Fig 4.5 SQL to Create Table for Sequence Diagram

```
create table statechart1(
Transition         VARCHAR(200),
InitialState       VARCHAR(50),
FinalState         VARCHAR(50));
```

Fig 4.6 SQL to Create Table for Statechart Diagram

Data from text file is entered to the tool with the help of SQL *Loader [24]. 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. Fig 4.7 shows the syntax of control file.

LOAD DATA
INFILE 'C:\Documents and Settings\Rohin\Desktop\classresult.dat'
APPEND INTO TABLE class1
FIELDS TERMINATED BY","
(Classname,Operation,Attribute,Dependency,Inheritance)

Fig 4.7 Control file syntax used in Oracle

All the data from class diagram, sequence 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. The screen shoots of tool is shown in fig 4.8 below:

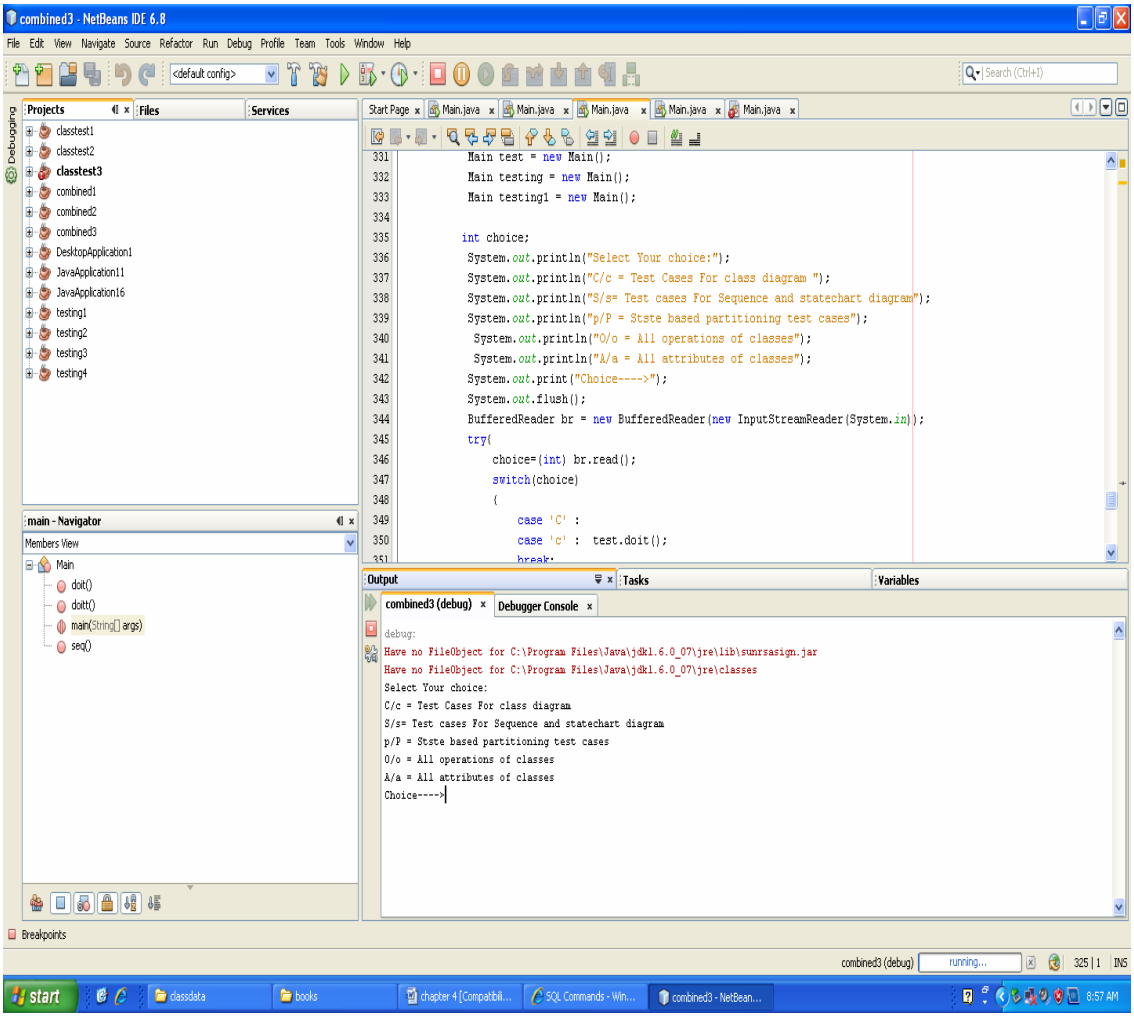

Fig 4.8 Screen Shot of Tool to Generate Test Cases

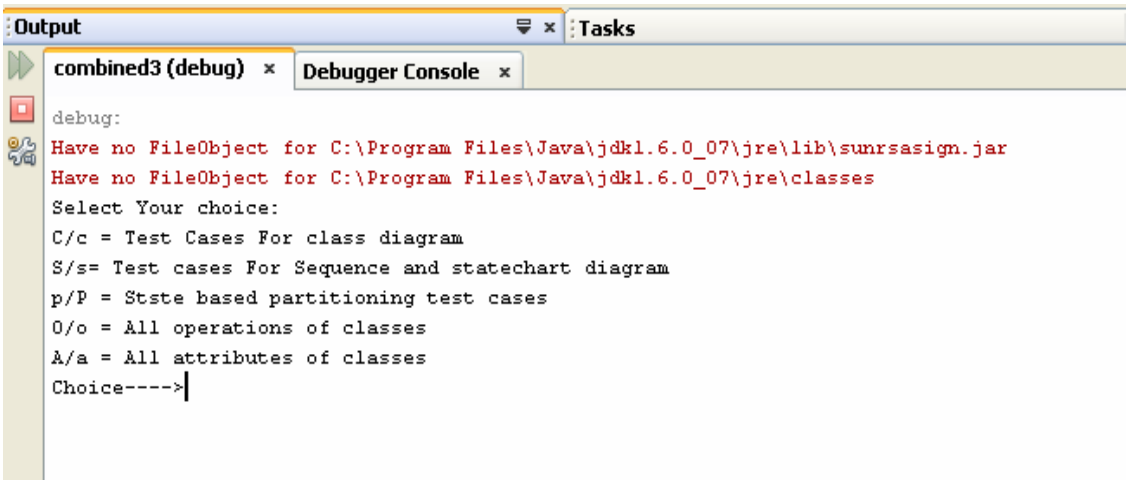

Fig 4.9 User Interface for Output

There are 5 options in the tool. C/c is pressed to generate test cases from class diagram. S/s is pressed to generate test cases from combination of sequence diagram and state chart diagram. P/p is pressed to generate State based partitioning test cases. O/o is pressed to know all operations of class diagram and A/a is pressed to know all attributes of class diagram.

Chapter 5
Experimental Results

5.1 An Example of SES (Student Enrollment System) To Test Tool

The tool developed has been tested with the help of an example of Student Enrollment System (SES). The class diagram for SES is shown in Fig 5.1 this represents the static view of the system now each operation, attribute, association in class can be used to derive test cases. A single class in a class diagram can be used to derive test cases. Similarly Statechart Diagram for whole system can be drawn as shown in Fig 5.2 and test cases can be derived by covering all states, all events, all actions and guard conditions.

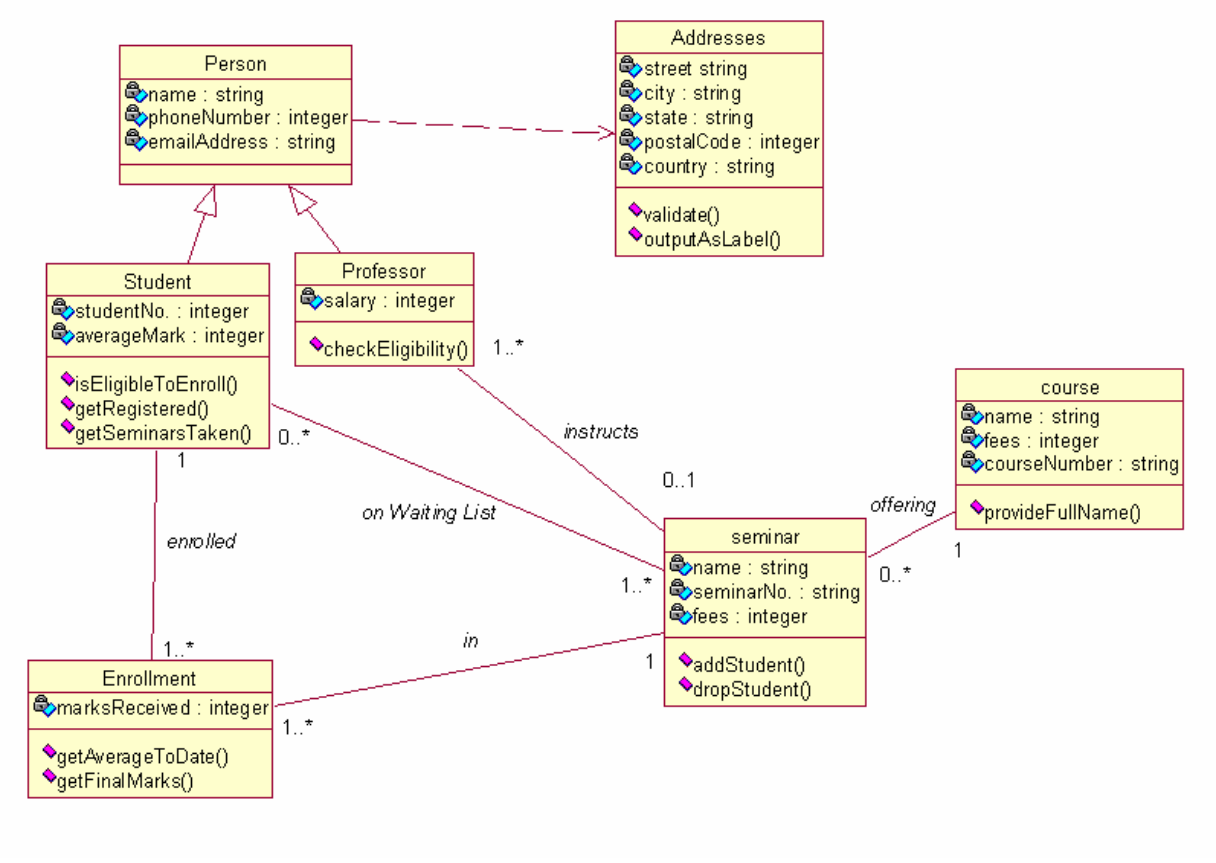

Fig 5.1 Class diagram for SES

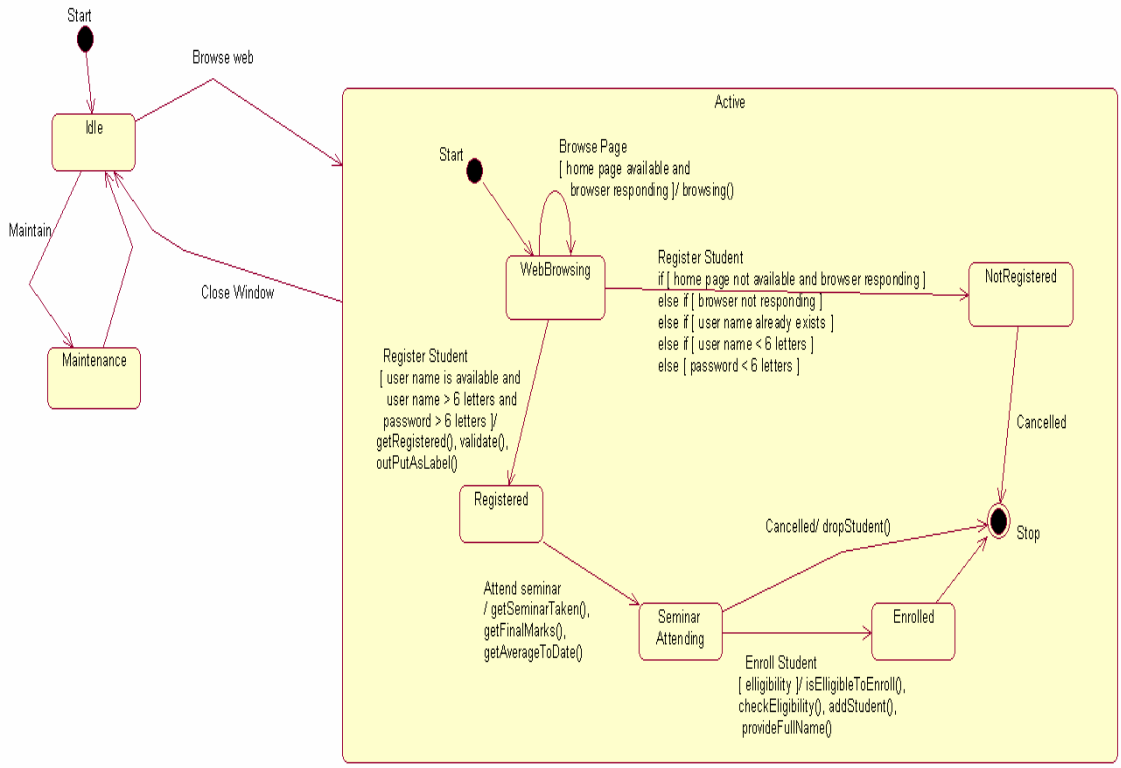

Fig 5.2 Statechart Diagram for Overall System

A novelty in proposed work is that single operation in a class is used along with its sequence diagram and state chart diagram of overall system to generate test cases. As in this example, the class student has one operation getRegistered() is used and corresponding to that operation one sequence diagram has been drawn shown in Fig 5.3. Now sequence and state diagram's petal file for this operation is used to give test cases as shown in Table 5.1. Test cases have been generated with the help of stored strings in Oracle database in the form of tables. In database the table of sequence diagram has following columns Pre condition, Test Sequence, Post condition shown in Fig 5.4. Preconditions and postconditions in sequence diagram have been given in documentation of each particular message of sequence diagram. Column Precondition and Post condition in sequence diagram represents pre condition and post condition from sequence diagram. Test sequence column in sequence diagram is nothing but a message from one object to

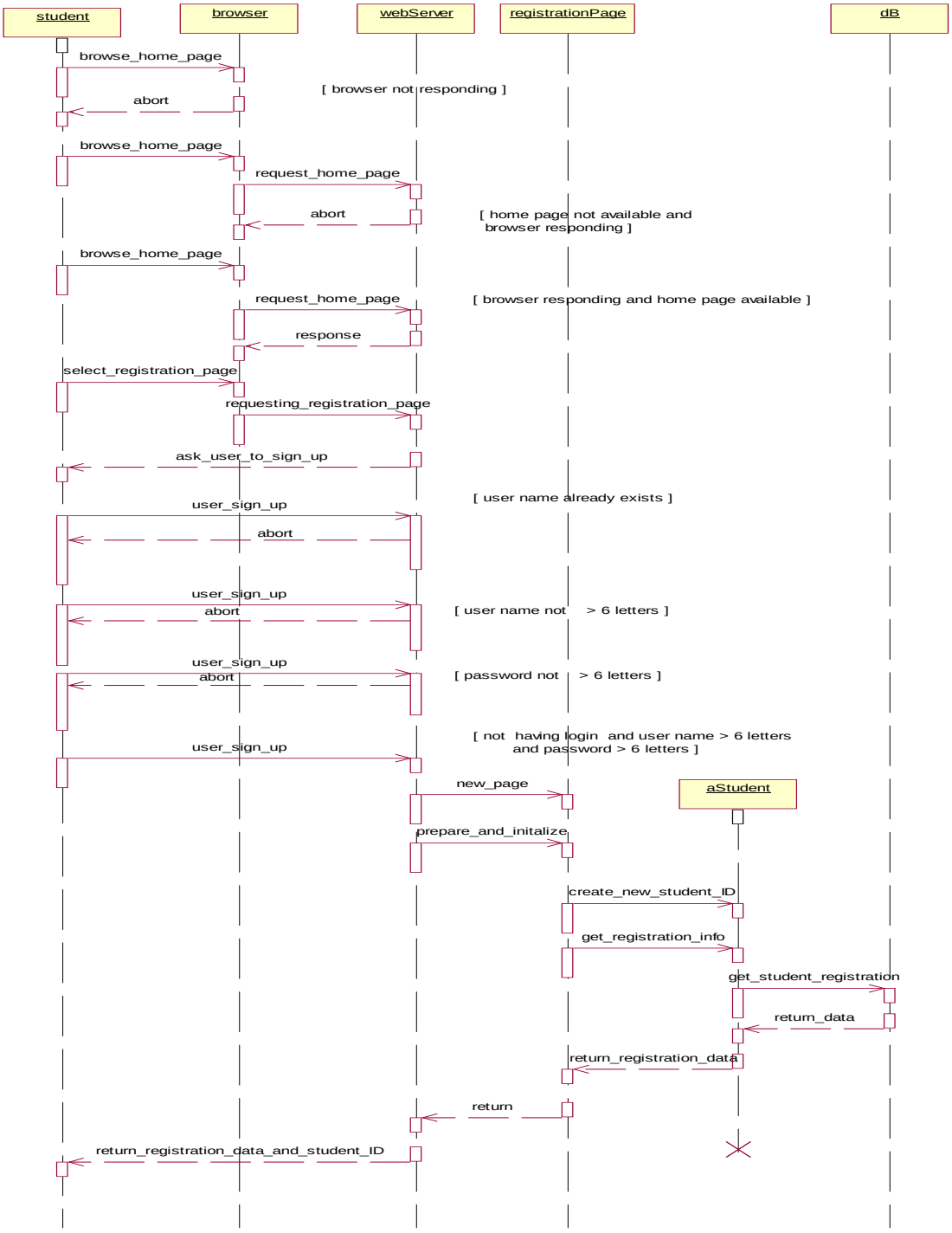

Fig 5.3 Sequence Diagram for Operation getRegistered()

Table 5.1 Test Cases Generated from Statechart & Sequence Diagram for Operation getRegistered()

Sr No.	Initial State	Pre-condition	Test Sequence	Post-condition	Final State
1	webBrowsing	Browser not responding	browse home page	Abort	Not registered
2	webBrowsing	Home page not available and browser responding	browse home page	Abort	Not registered
3	webBrowsing	Home page available and browser responding	browse home page	Response	WebBrowsing
4	webBrowsing	user name already exists	user sign up	Abort	Not registered
5	webBrowsing	user name < 6 letters	user sign up	Abort	Not registered
6	webBrowsing	password < 6 letters	user sign up	Abort	Not registered
7	webBrowsing	user name is available and user name > 6 letters and password > 6 letters	user sign up	Return Student ID And registration data	registered

PRECONDITION	TESTSEQUENCE	POSTCONDITION
browser not responding	"browse_home_page"	abort
home page not available and browser responding	"browse_home_page"	abort
home page available and browser responding	"browse_home_page"	response
user name already exists	"user_sign_up"	Abort
user name < 6 letters	"user_sign_up"	Abort
password < 6 letters	"user_sign_up"	Abort
user name is available and user name > 6 letters and password > 6 letters	"user_sign_up"	Student ID and registration data

Fig 5.4 Database of Sequence Diagram

another in a sequence diagram. Table of statechart diagram has following columns Transition, initial state, final state as shown in fig.5.5. In statechart diagram column transition contains the guard condition of each transition. Column initial state and final state contains information of initial state and final state of a particular transition. Test cases have been generated using inner join of two tables sequence diagram table and

statechart diagram table. Precondition column in sequence diagram table Fig 5.4 is similar to Transition column in statechart diagram fig 5.5, thus inner join query is used to generate test cases from sequence diagram and statechart diagram. All the data from two tables are picked to generate test cases.

Similarly test cases are derived for other operations in class diagram. Now class diagram has been combined to generate more effective test cases. Every class diagram has *uses* (dependency), *inheritance* and *association* relationships. These relationships are used now to generate test cases. As shown in Fig 5.1, a class *person* depends on a class *addresses* so any test case derived for class *addresses* is also used in class *person* for testing. Similarly class *student* and class *professor* has *inheritance* relationship with class *person*, so all test cases derived for operations of class *person* is used as such in all child classes for testing child classes, also if there is any other method in child class or there exists any method in parent class that is overridden in child class then it is needed to derive test cases for that method too.

TRANSITION	INITIALSTATE	FINALSTATE
user name is available and user name > 6 letters and password > 6 letters	"WebBrowsing"	":Active:Registered"
getRegistered	"WebBrowsing"	":Active:Registered"
validate	"WebBrowsing"	":Active:Registered"
outPutAsLabel	"WebBrowsing"	":Active:Registered"
home page available and browser responding	"WebBrowsing"	":Active:WebBrowsing"
browsing	"WebBrowsing"	":Active:WebBrowsing"
home page not available and browser responding	"WebBrowsing"	":Active:NotRegistered"
browser not responding	"WebBrowsing"	":Active:NotRegistered"
user name already exists	"WebBrowsing"	":Active:NotRegistered"
user name < 6 letters	"WebBrowsing"	":Active:NotRegistered"
password < 6 letters	"WebBrowsing"	":Active:NotRegistered"
getSeminarTaken	"Registered"	":Active:SeminarAttending"
getFinalMarks	"Registered"	":Active:SeminarAttending"
getAverageToDate	"Registered"	":Active:SeminarAttending"
elligibility	"SeminarAttending"	":Active:Enrolled"
isElligibleToEnroll	"SeminarAttending"	":Active:Enrolled"
checkEligibility	"SeminarAttending"	":Active:Enrolled"
addStudent	"SeminarAttending"	":Active:Enrolled"
provideFullName	"SeminarAttending"	":Active:Enrolled"
dropStudent	"SeminarAttending"	"$UNNAMED$1"

Fig 5.5 Database of Statechart Diagram

Fig 5.6 and Fig 5.7 shows database tables extracted from class diagram. There are two tables for class diagram first one contains information about Classname, Operation, Attribute, Dependency, and Inheritance and second contains the information of cardinalities of class and associated class. Class diagram's database table contains columns Classname, Operation, Attribute, Dependency, and Inheritance. There is another table for association of class diagrams. The table for association contains columns Classname, Cardinality, Associationclass, Ass_cardinality.

CLASSNAME	OPERATION	ATTRIBUTE	DEPENDENCY	INHERITANCE
"Person"	null	"name"	"Addresses"	null
"Person"	null	"phoneNumber"	null	null
"Person"	null	"emailAddress"	null	null
"Student"	isEligibleToEnroll	"studentNo."	null	"Person"
"Student"	getRegistered	"averageMark"	null	null
"Student"	getSeminarsTaken	null	null	null
"Professor"	checkEligibility	"salary"	null	"Person"
"Addresses"	validate	"street"	null	null
"Addresses"	outputAsLabel	"city"	null	null
"Addresses"	null	"state"	null	null
"Addresses"	null	"postalCode"	null	null
"Addresses"	null	"country"	null	null
"Enrollment"	getAverageToDate	"marksReceived"	null	null
"Enrollment"	getFinalMarks	null	null	null
"seminar"	addStudent	"name"	null	null
"seminar"	dropStudent	"seminarNo."	null	null
"seminar"	null	"fees"	null	null
"course"	provideFullName	"name"	null	null
"course"	null	"fees"	null	null
"course"	null	"courseNumber"	null	null

Fig 5.6 Database of Class Diagram

CLASSNAME	CARDINALITY	ASSOCIATIONCLASS	ASS_CARDINALITY
"Enrollment"	1..*	Student"	1
"Student"	1	Enrollment"	1..*
"seminar"	1	Enrollment"	1..*
"Enrollment"	1..*	seminar"	1
"seminar"	0..1	Professor"	1..*
"Professor"	1..*	seminar"	0..1
"seminar"	1..*	Student"	0..*
"Student"	0..*	seminar"	1..*
"course"	1	seminar"	0..*
"seminar"	0..*	course"	1

Fig 5.7 Database of Class Association Data

Older test cases of parent class will not apply in this case where a method is overridden in child class or for a new method in child class. In class diagram each class has associations with other classes and cardinality is associated with each association. Cardinality also helps in deriving test cases. Partitioning testing at class level helps to

derive test cases from class diagram by partitioning the operations into specific partitions. In this example, class diagram has class student it has operations isEligibleToEnroll, getRegistered and getSeminarsTaken, browsing. State based partitioning can be done to derive test cases as shown in table 5.2 below.

Table 5.2 Test Cases for Class S*tudent*

Test Case No.	Operations
1	getRegistered, getSeminarsTaken, isEligibleToEnroll
2	browsing

It is state based partitioning, here those operations that change the state will be treated in state operations and those operations that do not change the state will be treated in non state operations. Operation isEligibleToEnroll will change the state from Seminar Attending to Enrolled and getRegistered will change the state from webBrowsing state to registered state, getSeminarsTaken will change the state from Registered to Attending Seminar as shown in Fig 5.2, but operation browsing will not change the state. Test cases derived are shown in Table 5.2. Test case 1 will change the state of system but test case no.2 will not change the state. These operations have further test cases derived from sequence diagram for a particular operation and statechart diagram of overall system. Those test cases can be used to see whether test cases of these operations will change the state of system or not.

5.2 Results

The test cases generated by the tool have been shown in the figure below.

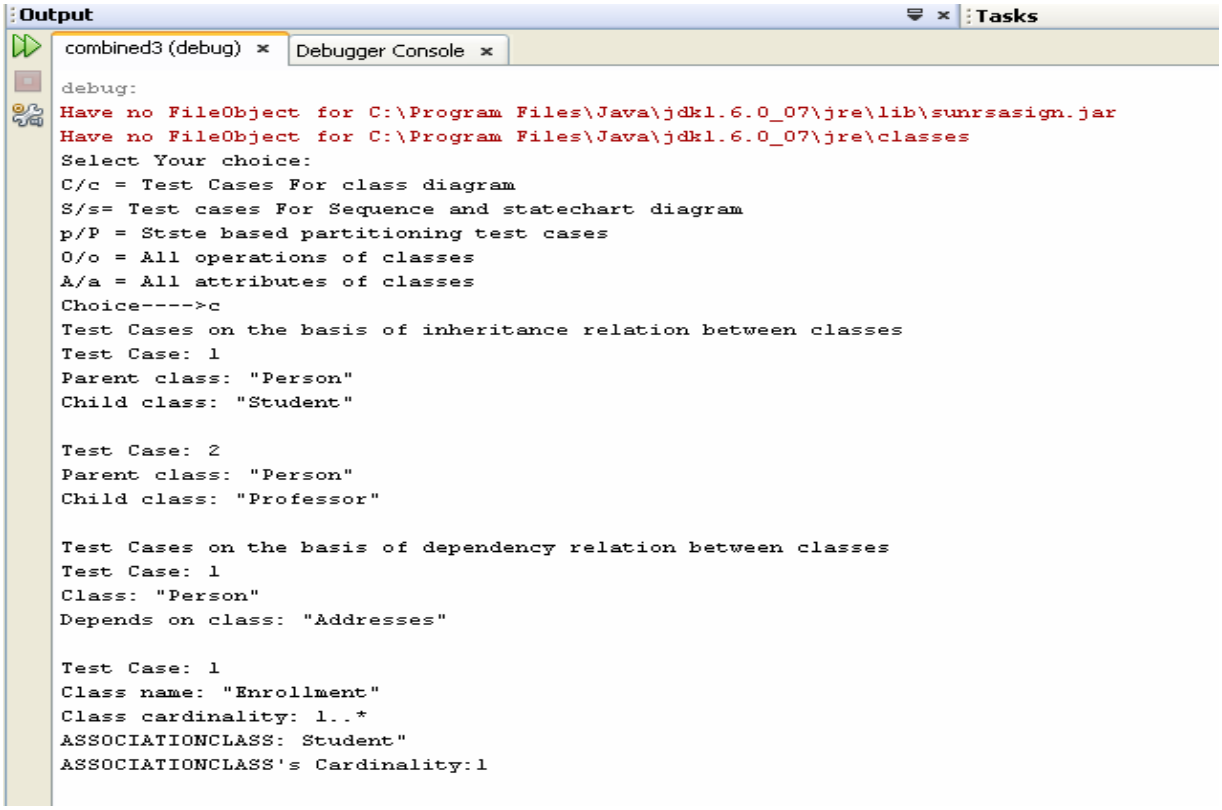

Fig 5.8 Test Case Generation Based on Inheritance and Dependency Relation

Test cases based on *inheritance* relation between classes shows which one is parent class and which one is child class. Test cases generated for operation of parent class can be applied on child class if child class does not override that operation. Similarly for *dependency* relationship all test cases derived for a particular operation of a class can be applied to dependent class.

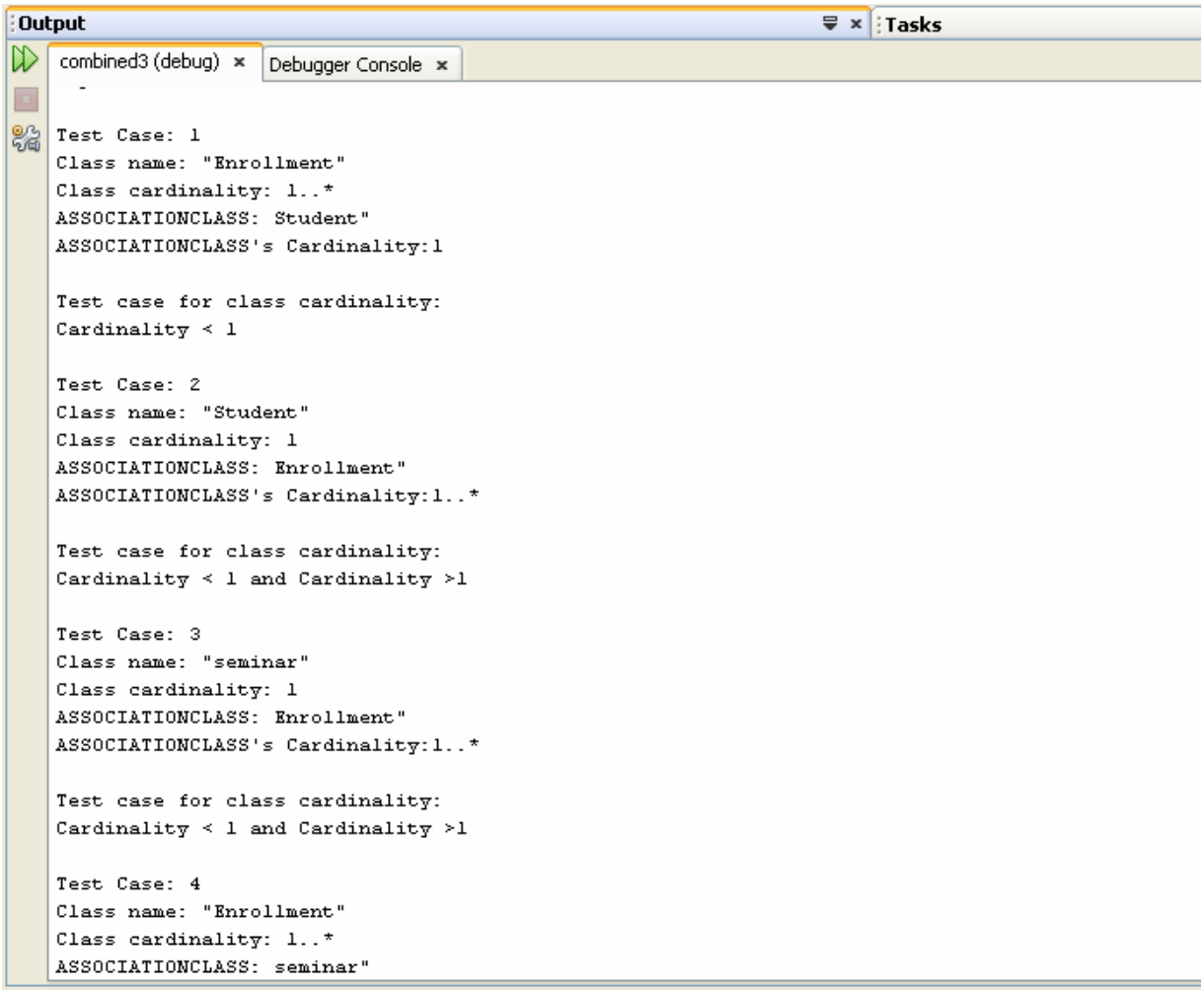

Fig 5.9 Test Case Generation Based on Class Cardinality

Above test cases are derived on the basis of class cardinality. Class name and class cardinality shows a name of the class and its cardinality. Association class and association class cardinality shows name of class and its cardinality which is associated with above stated class.

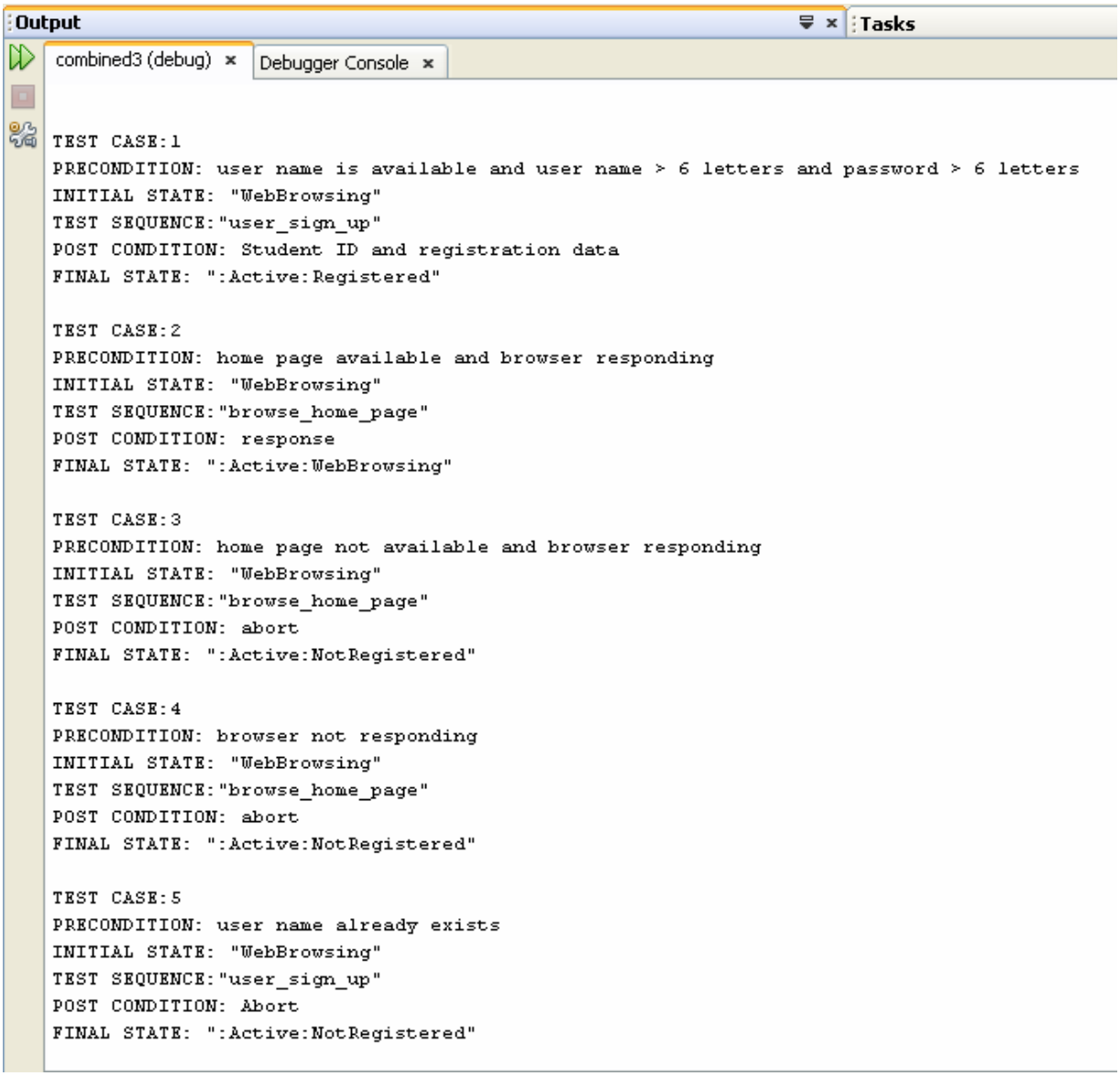

Fig 5.10 Sequence and Statechart Diagram Based Test Case Generation

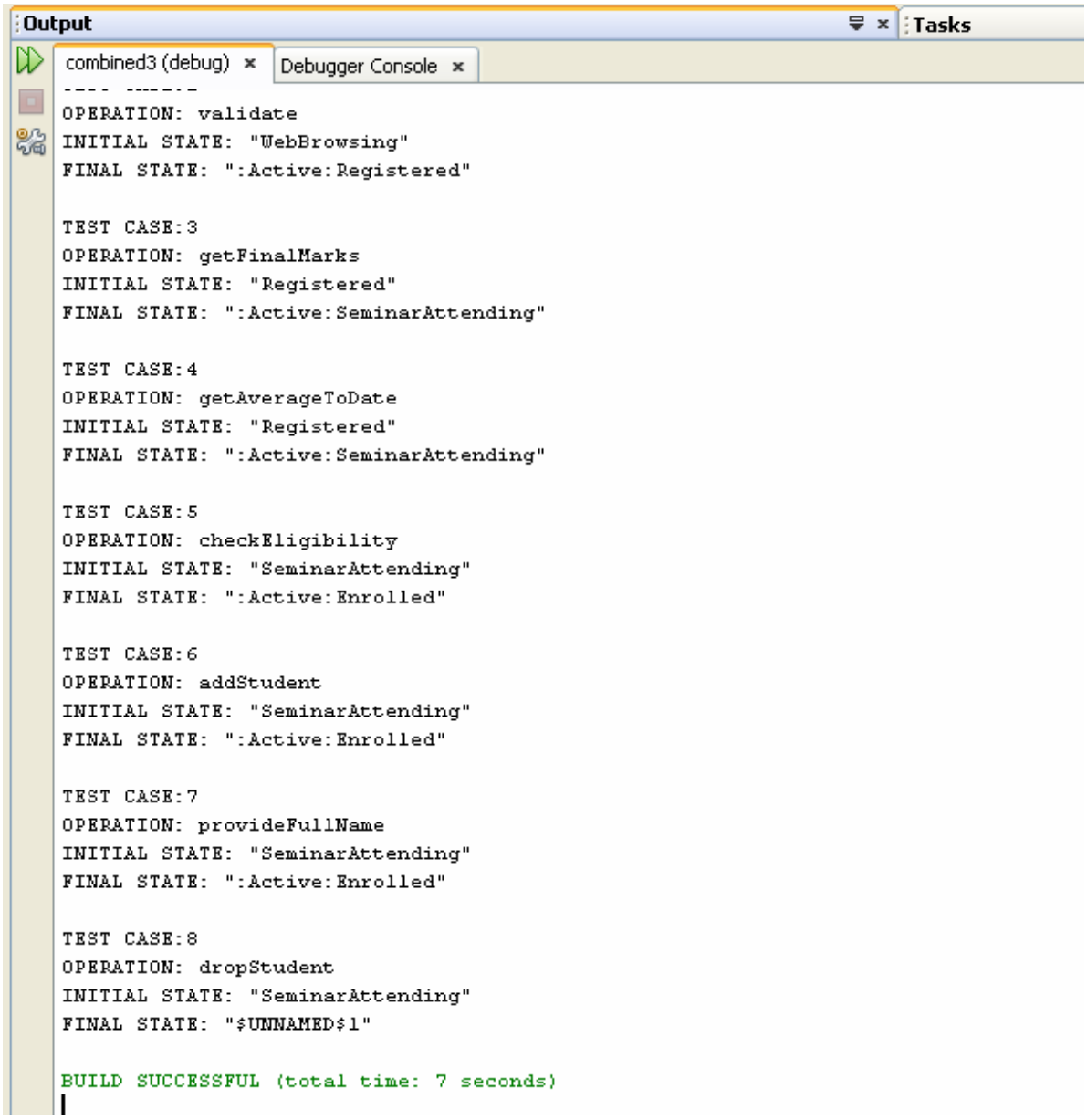

Fig 5.11 State Partitioning Based Test Cases

Fig 5.12 represents state based partitioning test cases. It describes all operations, their initial state and their final state. Operations that change their state are said state operations and operations that do not change state are called non state operations. First test cases are generated for all those operations that change state and second for all those operations that do not change the state.

It is clear from the above figures that the test cases generated by the tool are better than the test cases discussed in all the existing work. These test cases are more efficient and effective than the test cases discussed.

In "**A Practical Guide to Testing Object-Oriented Software**" [14] test cases has been generated on the basis of class diagram using OCL, 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. Present work uses combination of diagrams thus any loss in consistency in one diagram can be recovered from combining three UML diagrams Class, Sequence and Statechart diagrams. Present work automatically generates test cases and test cases shows initial state, final state, pre condition, post condition and test sequence.

In **"Inter Class Test Case Generation**" [5] 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 test cases have been generated from only class diagram. So test cases depend on only static view of system. Another problem is that it is assumed that model/Class diagram is consistent; if model is not consistent then test case generation will result in error in most cases. Present work solves this problem, it generates test cases automatically and using combination of UML diagrams consistency problem is also solved.

In **"Test Case automates Generation from UML Sequence diagram and OCL Expression"[**20] Sequence diagram and Class Diagrams are used to generate test cases along with OCL, but the test cases generated tell about class, operations, attributes, data limits and objects. Where class, attributes, operations are taken from class diagram, data limit from OCL and objects from Sequence diagram. These test cases do not tell about the changing states of the system, cardinality related test cases, inheritance and dependency related test cases. These problems are solved in present work.

In**"Automatic Test Case Generation from UML Sequence Diagrams"** [21] sequence diagram is used to generate test cases. Test cases generated shows input, output, pre condition and post condition only. Input state and output state and cardinalities related test cases are not shown. Present work shows input state, output state, class cardinalities related test cases and state partitioning based test cases.

In **"Automatic Test Case Generation Using Unified Modeling Language (UML) State Diagrams"** [22] 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. Information like Pre condition, Post condition and class related data is not present in these test cases. Present work also focuses on transition path coverage but it also shows initial state, final state, pre condition, post condition and test sequence. More over class cardinality related test cases are also shown.

In **"Automated-Generating Test Case Using UML Statechart Diagrams" [23].** They generate test case using the testing criteria that is the coverage of the state and transition of diagrams. Test cases generated depends on only transitions and guard conditions. Test cases do not tell about initial state and final states of a transition. Although test cases are automatically generated but these test cases depend on only dynamic aspect of system, static aspect has not been considered.

In ***"Testing Object-Oriented Systems: Models, Patterns and Tools"*** **[18]** test cases generated depends on only transition path coverage criteria, 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. Present work is automated. It generates test cases automatically and more consistent due to combination of three UML diagrams.

In present work test cases have been generated with the combination of three UML diagrams i.e. Class diagram, Sequence diagram and Statechart diagram. Each diagram presents a different view Class diagram presents design view, Sequence diagram and Statechart diagram presents implementation view. So the test cases generated with the combination of different views is more effective and efficient than all later discussed work.

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 types of diagrams that are used to specify the static and dynamic behavior of the software. UML diagram methodology is applied on object-oriented approach, not on structured approach. In the proposed system test cases have been generated from three diagrams class diagram, statechart diagram and sequence diagram. Proposed system focuses on these three diagrams, with the help of these diagrams tool can find out effective and efficient test cases and perform better testing than existing approaches. Class diagram based testing solves the problem of static test cases with automated method; sequence diagram and statechart diagram based test case generation approach solves the problem of dynamic test cases.

The proposed work is divided into several modules that perform automated test case generation on UML models. The modules are Petal file reader, SQL *Loader and Test case retrieval. Petal file reader module—this module reads the petal files of class diagram, statechart diagram and sequence diagram. Then tool creates the text files which contain the information extracted from class diagram, statechart diagram and sequence diagram. SQL *Loader—It is used to store the data into database. Here first tables are created and then data is loaded to database. Test case retrieval— in this module Netbeans IDE is used to generate test cases. SQL queries are written in java and test cases are shown on user output screen.

6.1 Conclusion

- A new algorithm has been proposed to extract information from petal files of Class Diagram, Sequence Diagram and Statechart Diagram.
- Proposed technique of combining information from combination of UML diagrams is simpler as compared to graphical parsing.

- Generated test cases are more effective due to:

- ❖ Combining three diagrams to generate test cases, as these test cases include static and dynamic aspects of the system.
- ❖ These test cases show initial state, final state, pre condition, post condition, test sequence, parent-child relationship, cardinalities based test cases, *uses* relationship based test cases and state based test cases.

6.2 Future Research

- ➢ Further work can be explored to use formal methods to make system suitable for real and large systems.
- ➢ More diagrams can be combined to generate test cases as Use cases can provide user view , activity diagram can give information of flow of control from activity to activity thus represents dynamic aspects of system. Combining all diagrams can result in generation of test cases that are more efficient and effective.
- ➢ Concept of polymorphism can be used to generate automated test cases.
- ➢ UML based integrated test case execution environment can be thought/planned for future.

References

[1] Myers, J. Glenford , The art of software testing, Publication info: New York: Wiley, c1979. ISBN: 0471043281 Physical description: xi, pp. 177.

[2] Hetzel, 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.

[3] B. Beizer. “Software Testing Techniques”, Van Nostrand Reinhold, 2nd edition, 1990

[4] R. S. Pressman. “Software Engineering: A Practitioner’s Approach”, 3rd Edition, McGraw Hill, New York, 1992, p. 559

[5] R. S. Pressman. “Software Engineering: A Practitioner’s Approach”, 6rd Edition, McGraw Hill, New York, 2005, pp. 424, 434, 449.

[6] Q. Nguyen Hung Testing Application on the Web:Test Planning for Internet-Based SystemsJohn Wiley & Sons2003

[7] H.-G. Gross. Measuring Evolutionary Testability of Real-Time Software. PhD thesis, University of Glamorgan, Pontypridd, Wales, UK, June 2000.

[8] IEEE. Standard Glossary of Software Engineering Terminology, Volume IEEE Std. 610.12- 1990. IEEE, 1999.

[9] A. Abdurazik and J. Offutt. “Using UML collaboration diagrams for static checking and test Generation”. In International Conference on the Unified Modeling Language (UML 2000), York, UK, October 2000.

[10] J. Hartmann, C. Imoberdorf, and M. Meisinger. “UML-based integration testing”. In International Symposium on Software Testing and Analysis (ISSTA 2000), Portland, USA, August 2000.

[11] R. Heckel and M. Lohmann. “Towards model-driven testing”. Electronic Notes in Theoretical Computer Science, 82(6), 2003.

[12] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. “Test cases generation from UML state Diagrams”. IEE Proceedings Software, 146(4), 1999.

[13] J. Offutt and A. Abdurazik. “Generating tests from UML specifications”. In International Conference on the Unified Modeling Language (UML 1999), Fort Collins, USA, October 1999.

[14] John D. McGregor, David A. Sykes “A Practical Guide to Testing Object-Oriented Software”, Addison Wesley, March 05, 2001, pp. 167

[15] Jos Warmer and Anneke Kleppe. “The Object Constraint Language: Precise Modeling with UML”.Boston, MA: Addison-Wesley. 1999.

[16] Kirani, S., and W.T. Tsai, ”Specification and Verification of Object Oriented programs ” Technical Report Tr 94-64, Computer Science Departement, University of Minnesota, Dec 1994.

[17] B. Beizer. “Black-Box Testing, Techniques for Functional Testing of Software and Systems.” Wiley, New York, 1995.

[18] R. Binder. “Testing Object-Oriented Systems: Models, Patterns and Tools”. Addison-Wesley, 2000.

[19] Philip Samuel, Rajib Mall and Sandeep Sahoo, “UML Sequence Diagram Based Testing Using Slicing”, IEEE Indicon 2005 Conference, Chennai, India, 11-13 Dec. 2005

[20] Li Bao-Lin, Li Zhi-shu, Li Qing, Chen Yan Hong, “Test Case automate Generation From UML Sequence diagram and OCL Expression” School of Computer Sichuan University, Chengdu 610064, China

[21] Monalisa Sarma Debasish Kundu Rajib Mall, “Automatic Test Case Generation from UML Sequence Diagrams”, 15th International Conference on Advanced Computing and Communications

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

[23] Supaporn Kansomkeat and Wanchai Rivepiboon, “Automated-Generating Test Case Using UML Statechart Diagrams” Proceedings of SAICSIT 2003

[24] Jonathan Gennick and Sanjay Mishra, Oracle SQL*Loader The Definitive Guide, O’Reilly & Associates, Inc., April 2001.

List of Papers/Publications

[1] Rohin Verma and Rajesh Bhatia, "Class Diagram and Behavioral Diagrams Based Test Case Generation" ISSRE 2010 21st IEEE, The 21st annual International Symposium on Software Reliability Engineering, November 1-4 2010 San Jose, CA USA (Communicated).

[2] Rohin Verma and Rajesh Bhatia," Behavioral Diagrams Based Test Case Generation", SCAM 2010, Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, 12-13 September, Timisoara, Romania. (Communicated)