# AN AUTOMATIC TEST REGENERATION TECHNIQUE FOR IMPROVING STATE MODEL COVERAGE

**AFRINA KHATUN**
**BSSE 0411**

A Thesis
Submitted to the Bachelor of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

**BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING**

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

AN AUTOMATIC TEST REGENERATION TECHNIQUE FOR IMPROVING STATE MODEL COVERAGE

AFRINA KHATUN

Approved:

*Signature* *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Dr. Md. Shariful Islam

Committee Member: Alim Ul Gias

Committee Member: Amit Seal Ami

To *Farida Yasmin*, my mother
who has always been there for encouraging me

# Abstract

Automated test regeneration intends to ensure high coverage of the system model from an existing test suite. Most of the existing coverage analysis techniques conclude their task by measuring coverage achieved so far by existing test suite. Hence, regenerating test cases for the untested elements is a research issue. To consider the coverage result, and incorporate UML diagram and source syntax information for regenerating useful test cases is a challenging task.

An automatic test regeneration technique to achieve high coverage of state model is proposed in this thesis. The architecture of the proposed technique is supported by three modules for the test regeneration. The first module processes the inputted UML diagrams, source code and test suite, as XML elements, source class and test steps respectively. The second module measures the model coverage result achieved through the test steps which directs the final module to regenerate test cases for the uncovered methods. The final module combines coverage result, the extracted UML and source syntax together to regenerate unit and integration test cases. The combination of coverage result, UML element and source syntax leads to executable and effective test cases.

Experiments have been conducted on three test projects to assess the effectiveness of the proposed technique. Two of these projects were used by existing techniques and another one was developed as an academic project. On average 97% of the regenerated test cases are executable transition sequences. The incorporation of state model method signature and software source syntax leads to executable test cases. The experimental result also shows that on average 54.23% and 43.23% improvements of the existing test suite are achieved in transition and state coverage respectively. The consideration of the coverage result allows the proposed technique to focus in covering the untested interaction sequences in the classes and thus, improves coverage of the existing test suite.

# Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Kazi Muheymin-Us-Sakib for his patience, support, motivation and immense knowledge during the thesis compilation. His continuous guidance while doing the research and writing the thesis has enabled me to complete the thesis.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Automatic test regeneration refers generating test cases to achieve certain goals such as increased test suite effectiveness, higher test coverage using the knowledge of existing test suite. Automatic regeneration for achieving coverage ensures the quality of a software product. Automatic regeneration ensuring coverage mitigates the efforts for manual inspection of the software model to identify uncovered elements, and allows the tester to regenerate test cases for those uncovered elements in a cost effective way. This chapter describes the overview of the automatic test regeneration for improving coverage and introduces the research challenges in this area. It also outlines the research question, and states the contribution and achievement of this thesis. At the end, it represents a guidance for the readers about the organization of this thesis.

## 1.1 Overview

Test case regeneration from existing test suite is done to ensure high test coverage of a System Under Test (SUT). Test coverage is a quality assurance metric which indicates how thoroughly a test suite exercises a given system [1]. A system model describes the behavior and requirements of the system. If a test suite fails to achieve full coverage of the system model, it indicates that the test cases

have missed important requirements of the SUT. In model based coverage analysis techniques, manual or auto generated test suite is executed against the system UML diagrams like - class, state, sequence diagrams and GUI screens. It is also referred as structural coverage criteria. In this approach, coverage result is measured by parsing the test cases and finding corresponding elements in the UML diagrams. The coverage result is represented to the tester in a graphical or documented form. However, the existing coverage analysis techniques conclude their task by only identifying the covered and uncovered elements. These techniques do not regenerate test cases for the untested elements. As a result the tester needs to manually regenerate test cases for those untested elements. Here comes the need of an automatic approach that automatically measures the coverage of the existing test suite and regenerate executable test cases to cover the untested elements.

## 1.2 Motivation

Achieving full coverage of the model generally fails, as test automation tools lack predefined coverage checking while generating test suites. Existing coverage analysis tools measure the coverage of a system model, based on some predefined coverage criteria like - state coverage, transition coverage, all path coverage etc. These tools only identify the tested and untested elements of the system model by the exiting test suite. Therefore, to ensure coverage either existing test generation algorithms need to be changed or regeneration needs to be done. On the other hand, most of the automated model based test generation techniques generate test cases from an abstract model of SUT [2] or source code [3] or both [4]. However, those fails to find how much coverage is achieved through the generated test suite. As a result, important system requirements may remain untested. Even if the number of uncovered elements is small, this may lead to manual inspection of the whole model again to find those elements.

A dependence graph-based test coverage analysis technique for object oriented programs has been proposed by Najumudheen et al [5]. In this paper the source code was instrumented to measure traditional coverage criteria like - statement, branch, method coverage etc and a call based system dependence graph was constructed by parsing the source code. During coverage analysis phase, the graph is marked based on coverage criteria and a coverage analysis report is produced for the tester. Therefore, the approach does not generate test cases for the uncovered part of the graph.

A state model based test coverage analysis technique of UML state machines has been proposed by Ferreira et al. [6]. The proposed tool receives XML formatted state model and auto generated test suite to identify covered and uncovered elements by the test suite. Therefore, the test suite still lacks test cases that are required to ensure the full coverage of the state model.

Minsong et al. has proposed a technique for automatic test case generation for UML activity diagrams [7]. This approach executes random generated test cases against the UML activity diagram and selects test cases that satisfy predefined coverage criteria. However, redundant test cases are generated at the phase of random generation and the reduced test suite does not ensure full coverage of the activity diagram. Fraser et al. has proposed a tool called, EvoSuite for automatic test generation of object oriented programs [3]. Dehla et al. proposed a technique for generating test cases from a combination of sequence and state diagrams in [8]. This technique does not consider model requirements and coverage for test generation. A technique to produce test cases from UML models is offered by Sarma et al. [9]. In this technique UML use case and sequence diagram have been considered for generating test sequences. However, these techniques only focus on test sequence generation and totally ignore coverage analysis of the system model.

An integration testing coverage tool for object oriented program has been proposed in [10]. The tool automatically instruments the source code and collects

coverage data while test execution. After execution, it identifies the uncovered methods from coverage result and extracts inter class method call path from the sequence diagram and generates test case for the path. However, the approach generates inexecutable test cases which requires the tester to manually generate executable test cases.

Few test regeneration techniques have also been proposed in literature. A test case regeneration approach based on sequential pattern mining has been proposed by Wei He et al. in [11] to produce test cases from existing test repositories. It uses a GA based approach to regenerate test cases from existing test repository. Alshahwan et al. proposed two test regeneration techniques using standard and value based Def-Use testing accordingly for improving test coverage of web applications and fault detection [12]. However, both of these techniques ignore coverage criteria of system model elements while test regenerating process.

The existing researches focus on either only finding the coverage result or regenerating test cases. The coverage analysis approaches identify uncovered elements, but do not regenerate test cases for those elements. Therefore, untested requirements lead to unknown risks. Although the number of untested elements is small, this lead to manual inspection of whole model. On the other hand, most of the test generation techniques do not consider model coverage criteria while generating test cases. As a result, the generated test cases do not ensure full coverage of the model. Few coverage based test generation techniques also exist in literature which generate test cases for covering specific test criteria. However, those techniques generate same test cases repeatedly, each time the techniques are applied. These problems can be mitigated if existing test suites are considered to identify uncovered elements and regenerate test cases for those uncovered elements only.

## 1.3 Research Questions

To overcome the limitations of the above mentioned approaches, the coverage achieved through existing test cases needs to be measured, and this coverage result needs to be analyzed while regenerating test cases. Thus, this leads to the primary question of this research

- How can the coverage analysis process be incorporated with test regeneration technique to produce valid test cases that ensure full coverage of the system state model?

In a nutshell, this research will incorporate coverage analysis result with test regeneration to generate test cases for the uncovered paths of the model. A technique for automatic test regeneration will be proposed that will use the information extracted from UML class and state diagrams, source code syntax and auto generated test suite for the test regeneration process. That can be achieved through answering following sub questions-

1. How to analyze coverage of the auto generated test suite by executing the test cases against the system state model?

The structure and elements of the system model will be extracted from UML class, state diagrams, and existing test steps[1] will be extracted from the auto generated test cases. These test steps will be compared against the state model to identify the covered and uncovered elements. The result of the coverage analysis will be stored that will be considered to regenerate test cases.

2. How the coverage analysis result will be used while regenerating test cases?

The coverage analysis result will be considered to find uncovered areas from the model. Then the test cases will be regenerated for the uncovered elements and paths, and stored. While regenerating test cases, the source code syntax will

[1]Each statement in a test method is a test step

also be matched with state diagram method call syntax to check consistency and produce executable test cases.

Answering these questions and providing a solution for automatic test regeneration for achieving full state model coverage, are the objectives of this research.

## 1.4 Contribution and Achievement

This research incorporates coverage analysis result with test case regeneration technique to produce unit and integration test cases to ensure high coverage. Each state diagram represents a class of the system under test which is considered basic unit of testing. Therefore, full coverage of a state diagram represents the coverage of internal interactions in the class.

The technique contains three modules to manage the whole regeneration process. The Parser, Coverage Analysis and Test Regeneration - each module has some predefined responsibilities to support the technique implementation. The Parser is the input data provider of the proposed technique. It extracts test steps, source syntax and UML elements by parsing an existing test suite, source code and UML diagrams respectively and processes these information in a structured way to be used later. The Coverage Analysis module uses the processed test suite, UML class and state elements to identify the covered and uncovered elements of the state model by simulating the execution of test cases against the model. The Test Regeneration module first uses the coverage result to generate all possible uncovered transition paths from the state diagram assuming it as a directed graph. It then regenerates unit and intra class integration test cases for the generated uncovered transition paths. While regenerating tests, method call signature of transitions from the UML state diagrams and method call syntax from the source code are also considered to check the consistency, that ensures regeneration of executable test cases.

The proposed technique has been implemented using java programming language. For evaluating competence, the techniques has been applied on three experimental projects and the results are analyzed. These experimental projects are also implemented in java. The proposed technique is applied on an automated test generation framework, SSTF [4]. The coverage of the existing test suite and regenerated test suite are measured using an existing coverage analysis technique called MoCAT [6]. On average 54.23% and 43.23% improvement, in transition and state coverage respectively of SSTF is achieved by the regenerated test suite of the proposed technique.

## 1.5 Organization of the Thesis

This section gives an overview of the remaining chapters of this thesis. The chapters are organised as follows –

**Chapter 2:** The view of basic concepts and terminologies related with test coverage and regeneration is represented in this chapter.

**Chapter 3:** To the best of author's knowledge, no existing state based coverage analysis technique regenerates test cases for the parts which remains untested by existing test suite . This chapter represents the existing coverage analysis and test regeneration approaches.

**Chapter 4:** The architecture of the proposed technique is briefly demonstrated in this chapter.

**Chapter 5:** The implementation and result analysis are presented here.

**Chapter 6:** This chapter contains a discussion about the threats to the validity of the results and the future direction to enhance the technique.

# Chapter 2

# Background Study

Software testing is one of the most important phases of Software Development Life Cycle (SDLC). In a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost are expended in testing the program or system being developed [13]. The purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users [14]. The history of software testing is a vast area. In 1969 the term Testing was coined by Edsger Wybe Dijkstra [15]. In the same year Bender created the first static and dynamic analysis tools for improved test coverage [16]. After that the separation of debugging from testing was initially introduced by Glenford J. Myers in 1979 [13]. This chapter presents the different levels of testing in practice as well as the types of coverage criteria used for measuring test adequacy. Automated test suite generation, regeneration and coverage analysis processes are also described in the following parts.

## 2.1 Testing Levels

*"The goal of the testers is to make the program fail. If his test case makes the program or system fail, then he is successful; if his test case does not make the program fail, then he is unsuccessful." "A good test case is a test case that has a*

*high probability of detecting an undiscovered error, not a test case that shows that the program works correctly."* [17] — Glenford Myers

*Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software. Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.* [18]

Testing is a vast area. Software testing is considered as verification and validation process. Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements [19]. In other words, to make sure the product is built as per customer requirements. It is usually associated with traditional execution-based testing, that is, exercising the code with test cases. Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [19]. In other words, to make sure the product behaves the way we want it to. It is usually associated with activities such as inspections and reviews of software deliverables.

Various testing levels are introduced as a means of clearly defining the objective of a certain level or goal for a program or project. Software testing is done at each level of software development. Based on objectives of testing, different testing techniques are applied in software development life cycle. Those testing levels are described in the following section -

### 2.1.1 Unit Testing

*"Unit testing means testing the smallest separate module in the system."* [20] – Runeson.

A unit test is an automated piece of code that invokes a unit of task in the system (for example: a method) and then checks the actual behavior of the unit task

to make an assumption. A unit of work can be a single method, a whole class, multiple classes or a set of closely related classes [21].

Unit testing is just one of the levels of testing which go together to make the big picture of testing a system. It complements integration and system level testing [22]. For any system of more than trivial complexity, it is highly inefficient and ineffective to test the system solely as a big black box. Therefore, Unit testing is generally seen as a white box test class as it focuses on evaluating a piece of unit as it is implemented. As unit testing is primarily focused on the implementation, and requires an understanding of the design intent, it is much more efficiently done by the developers rather than by independent testers.

### 2.1.2 Integration Testing

Integration testing is the testing applied when all the individual modules are combined together to form a software project [23]. Unlike unit testing which is done at statement levels, testing is done at the module level. Integration testing emphasizes the interactions between modules and their interfaces. There are different integration testing techniques like: bottom-up, top-down, modified top-down, sandwich, modified sandwich, big-bang testing, threads integration, and critical modules integration [24]. Integration testing is also a general way of specifying also intra and inter class integration testing.

In integration testing the entire system is viewed as a collection of set of classes, determined during the system design. Like unit testing as it is confined with the internal interactions of the system modules, therefore it is also carried out by developers [25]. As integration tests identifies the errors that can occur while the interaction of system classes, therefore it has great emphasize in software testing.

### 2.1.3 System Testing

*"System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution."* [26]

System testing refers testing the complete application as a whole. The goal at this level is to evaluate whether the system has complied with all of the outlined requirements and to see that it meets quality standards. System testing is undertaken by independent testers who havent played a role in developing the program [25]. This testing is performed in an environment that closely mirrors production. System testing is very important because it verifies that the application meets the technical, functional requirements, and specification requirements that were set by the customer. In other words, the implementation under test is compared to its intended specification [27]. It also considers non functional requirements like performance, reliability, security, and maintainability testing etc [28].

### 2.1.4 Acceptance Testing

Acceptance testing refers the testing process that is applied to ascertain whether the software products under development have met the requirements specified by agreement, for example business contract, official agreement etc [29]. Developers write unit tests to determine if their code is doing things as expected. On the other hand, customers write acceptance tests to determine if the system is doing the expected things. The acceptance tests represent customer interests and give the customer confidence that the application has the required features [30]. In theory when all the acceptance tests pass the project is done.

Acceptance testing is categorized in four types User, Operational, Contract and Compliance acceptance testing. The User acceptance testing focuses on the customer requirements and is performed by the customers as well. The Operational acceptance testing validates whether the system meets production requirements and is performed at developers end. Contract acceptance testing is performed against the requirements written in the business contract and Compliance acceptance testing is executed against governmental, legal or safety regulations.

## 2.2 Testing Methods

Software testing methodologies are the different approaches and ways of ensuring that a software application is fully tested. As with emergence of technology, software products get even more complex and intertwined, it is more important than ever to have a robust testing methodology for making sure that software products systems being developed meet their intended requirements. Therefore different testing methodologies are applied to be sure about software qualities. In the following section different testing methods are demonstrated.

### 2.2.1 Static Testing and Dynamic Testing

Based on the actual execution type of test cases, there are two types of testing method Static testing and Dynamic testing. Static testing refers testing the system under test without executing the software. It used to analyze software in a non-runtime environment when the software is inactive, and not in operation. It is basically done manually or using some static analysis tools. Static analysis is performed using reviews, walk through, code inspection, program analysis, symbolic execution and model checking techniques [31].

Static testing is about prevention whereas dynamic testing is about cure. Dynamic testing refers describing the dynamic behaviour of the code. In Dynamic

testing software executed by giving set of inputs, examined its output and compared what is expected. It is feedback driven, a sequence of test requirements are given to tester and the development of the requirements continue with the testing process with the feedbacks from developers [32]. Dynamic testing is not cost effective as compare to Static testing. Unit testing, Integration testing, System Testing, Acceptance Testing are the techniques of performing dynamic tests. Various automated tools are also available and used for performing those tests.

### 2.2.2 Functional Testing and Structural Testing

Different testing techniques reveal different quality aspects of a software system. Based on the information flow, there are two major categories of testing techniques Functional testing and Structural testing. Functional testing is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. [33]. Functional testing is also referred as black box testing. It considers the test modules as big black box, where the tester cant see inside the box. The tester knows only that information can be input into to the black box, and the black box will send something back out. That is functional testing assures whether a specified module provide the expected output regardless of the internal mechanism of the modules. Structural testing is testing that takes into account the internal mechanism of a system or component. [33]. Structure-oriented test methods define test cases on the basis of the internal program structures [34]. Structural approach is also known as white box testing approach. It considers the test modules as white box, where the tester also takes into account the internal structure of the software modules. The reason of considering internal structure is to measure a set of coverage criteria that are expected to be achieved by the tests. The coverage of coverage criteria includes path coverage, branch coverage, and statement coverage.

### 2.2.3 Model Based Testing

Model-based testing (MBT) is software testing in which test cases are generated in whole or in part from a model that describes functional aspects of the SUT. Model-based testing has recently gained attention with the popularization of UML models in software design and development [35]. A model of software is a depiction of its behavior. Behavior can be described in terms of the input sequences accepted by the system, the actions, conditions, data flow etc. There are various such models, for example UML use case, activity diagrams, state diagrams, class diagrams, control flow, data flow, and program dependency graph etc.

Model based testing process comprised of five steps: model creation, test generation, test concretization, test execution and result analysis [36]. In model creation phase, a model of a system is constructed that defines the systems desired behavior for specified inputs to it. It is a structural analysis of the system. Scenarios are described as a sequence of actions to the system along with the correct responses of the system. In test generation phase, test coverage is understood and test plans are developed in the context of the SUT, the resources available and the coverage that needs to achieved. In the next phases, the abstract tests are converted into tests to be executed and analyzed against the specified test coverage criteria. The benefit of MBT is its reusability [37], all of this work is not lost. The next test cycle can start where this one left off. If new features are added to the system, they can be incrementally added to the model, and the tests are expanded.

## 2.3 Test Coverage

An important issue in the management of software testing is to ensure that before any testing the objectives of that testing are known and agreed and that the objectives are set in terms that can be measured. Such objectives should be quantified, reasonable and achievable. [38]. There are various ways of classifying test

adequacy criteria. Program based and Specification based are two common types of test adequacy criteria.

Program based test coverage criteria specifies testing requirements in terms of the program under test and decides if a test suite is adequate to thoroughly exercise all parts of the program. Program based test coverage criteria includes statement coverage, branch coverage, condition coverage etc.

Specification based criteria specifies the required testing in terms of the requirements of the software. This criteria ensures that a test set is adequate if all the identified features or requirements of the system have been fully exercised. For this purpose, models are required to generate test cases and measure test coverage. UML use case, class diagram, state diagram, sequence diagram, control flow, data flow graph etc are used as the models and are built using different languages or notations for processing. Different model coverage criteria are measured based on different type of models.

## 2.4 Model Coverage Criteria

In model based testing techniques a system is represented as using UMLs. The UML is used as a language for specifying, visualizing, constructing and documenting the artifacts of software system. The UML provides a variety of diagrams that can be used to present different views of an OO system at different stages of the development life cycle [39]. Models based testing techniques extract and generate test cases from these UML models based on model coverage criteria. Different coverage criteria based on UML diagrams are outlined in the following part.

### 2.4.1 Class Diagram Based Coverage Criteria

A class diagram shows the static structure of a system. It identifies all the entities, along with their attributes, in the system and specifies the relationships between

the entities [40]. These relationships or associations are represented by links among the entities. Class diagram based coverage criteria contains Association-End Multiplicity criteria (all association relations are tested), Generalization criteria (refers inheritance relations are covered), Class Attribute criteria (refers test cases at least once instantiate the class attribute with specified values) etc [41].

### 2.4.2 Sequence Diagram Based Coverage Criteria

A Sequence diagram is an interaction diagram that shows how processes operate with one another and in what order. It depicts the exchange of messages between a set of objects through message calls among the objects. All the events or interactions among the objects are represented in a order of time. Each path in the sequence diagram represents a sequence of messages that begins with an externally generated event and ends with the production of a response that satisfies this event. Therefore, All-Path coverage, All Branch coverage, All Restricted Control Flow Graph Paths coverage are considered in sequence based coverage analysis [42]. These sequence based criteria are used while generating integration test cases for testing interactions among the system classes.

### 2.4.3 State Diagram Based Coverage Criteria

A state diagram represents the states an object can attain as well as the transitions between those states of a software system. In this context, a state defines a stage in the evolution or behavior of an object, which is a specific entity in a program or the unit of code representing that entity. Each state diagram of a system is generally associated with a class of the system and represents the status change of an object of that specific class. Therefore, state diagrams are directed graphs containing a finite number of nodes, transition arcs among the nodes, guard conditions of the transitions and actions at occurrence of certain transitions.

State diagrams are useful in all forms of Object-Oriented Programming (OOP)

[42]. The concept is more than a decade old but has been being used in different context of object oriented paradigms. As each state diagram represents a class and a class is considered as the basic unit of testing, state diagrams are used while generating intra class integration test cases. All transition coverage, All Transition Pair (adjacent transitions are tested) coverage, Full Predicate Coverage, All Path coverage are considered as state machine based coverage criteria.

### 2.4.4 Structural Model Coverage

These criteria exploit the structure of the model, such as the nodes and arcs of a transition-based model, or conditional statements in a model. The structural coverage criteria can also be viewed from two perspectives  control flow oriented coverage and transition oriented coverage. Control flow oriented coverage intends to exercise the decisions of the model. The decisions can be any guard condition of a model event. Statement coverage, decision coverage, condition coverage, condition/decision coverage are considered as control flow oriented coverage criteria. Transition oriented coverage criteria uses explicit graphs like state diagrams, sequence diagrams, activity diagrams containing a number of nodes and arcs. There are many graph coverage criteria that are used to control test generation. The transition sequences found in those model graphs are useful while generating integration test case. Transition oriented coverage comprises all transition coverage, all state coverage, all path coverage, all transition pair coverage etc.

### 2.4.5 Data Coverage

Data criteria deal with how to choose a few test values from a large data space. The basic idea is to split the data space into equivalence classes and choose one representative from each equivalence class. A data coverage criterion helps cutting a portion of the possible values depending on the approach selected. The two most common approaches for data coverage analysis are  One value and All value. One

value refers selecting only one value from the possible domain. On the other hand, All value approach selects every possible value of a variable.

## 2.5 Automatic Coverage Analysis

With emergence of technology, software has become the only or a key component of a product, customer satisfaction may be highly correlated with software quality. As testing usually does not contribute directly to adding new features, testing may be considered overhead that must be as effective as possible. Indeed, testing is a very labor and resource intensive process that often accounts for between 40 and 80% of the total cost of software development. Besides, high coverage is another point that is required to ensure the quality of software. To produce a quality software with in a budget and time limit, automated coverage analysis tools is a must. Some significant issues are pointed in [43] that highlight the importance of automated coverage analysis tools. The issues are for example - (1) Developers and testers have no good way of knowing how much of their code had been covered in testing. (2) Developers and testers are so busy in meeting customer requirements that they miss important coverage criteria.

Various automated source based coverage analysis tools have been proposed in literature. However, these tools ignore model coverage criteria which may fail to cover a broad overview of the system. Also no significant effort is shown in regenerating test for the uncovered portions based on the coverage analysis result. These limitations indicate that further effort is needed to automatically support good coverage of a software product.

## 2.6 Automatic Test Regeneration

Regeneration refers generating test cases using knowledge from existing test cases. Various automated test generation approaches exist in the literature. However,

due to ignoring model coverage criteria or manual or random generation of test cases, existing test cases do not properly test all the requirements of the model. Therefore, automated coverage analysis techniques are used to identify uncovered parts. Although the number of uncovered elements is small, it leads to manual inspection for testers. Therefore, to test the uncovered parts either test existing test generation techniques needs to be changed or test regeneration is done. Changing existing generation algorithm or proposing another algorithm, result in dropping the existing test cases. Therefore, to increase the effectiveness and re-usability of existing test suite, further effort in automatic test regeneration is required.

## 2.7 Summary

A brief discussion about software testing, test coverage criteria and test regeneration has been discussed in this chapter. With the emergence of technology, the processes of software testing to obtain predefined coverage criteria have become developed and are presented in the above sections. A view of test regeneration and coverage analysis is also outlined. Existing techniques for automatic test regeneration and coverage analysis have been discussed in the next chapter.

# Chapter 3

# Literature Review

In the literature, several techniques for test case regeneration and test coverage analysis have been proposed. Most of the test coverage analysis techniques measure the coverage result, by identifying the covered model elements through existing test suites. These techniques do not generate text cases for the untested model elements. Therefore, the existing test suite fails to test all the elements of the software models. Few test case regeneration techniques have also been proposed in the literature. These techniques regenerate test cases from existing test repositories. However, these techniques ignore already achieved model coverage result of existing test repositories while regenerating test cases. Existing test generation techniques focus only generating test cases from model diagrams or source code or both. Therefore, those techniques fail to cover all the elements of the models, which could lead to manual inspection of whole model. In this chapter, these existing techniques are outlined.

In 1969 Edsger defined testing : Testing shows the presence, not the absence of bugs [15]. The field of software testing is a rich field. Various automated test generation techniques have been proposed by researchers. The continuous study in this field shows that testing is crucial to the success of a software project [44]. The measure of testing completeness and goodness requires test completion cri-

teria. Coverage is one such measure. Test coverage is a quality assurance metric which indicates how thoroughly a test suite exercises a given system [1]. Test coverage can help in monitoring the quality of testing, and assist in directing the test generators to regenerate tests that cover areas that have not been tested before.

## 3.1 Test Case Regeneration Approaches

Test case regeneration from existing test suite is done to ensure high test coverage of a System Under Test (SUT). Many existing approaches generate test cases for the programs under test without considering the available test cases. Considering existing test cases while generating test cases leads to the opportunity of getting useful information from those. Few test case regeneration techniques from existing test repositories have also been proposed in the literature survey.

A sequential pattern mining based test case regeneration technique for object oriented projects was presented by Wei et al. [11]. The approach applied Bi Directional Extension mining strategy to obtain frequent subsequences of method call from existing test suites. It calculated a threshold value by getting the average method invocation frequency in the corresponding test repository. Then it constructed a set of method subsequences, which occurs no fewer than the threshold value. A Genetic Algorithm (GA) based approach was applied on the method subsequences to regenerate test cases. Each test case was a method call sequence and contained random generated method arguments. The approach did not identify whether the existing test repositories exercised all the elements of the program. As a result, the regenerated test cases contained the same but optimized method sequences, which failed to test the uncovered parts.

Alshahwan et al. proposed test regeneration techniques for web applications [12] using standard and value based Def-Use (DU) testing accordingly. This approach considered HTTP requests from a test suite to form client side requests and com-

bined fragment of these requests to regenerate test cases to execute server side requests. At first a simple static analysis was performed to determine the locations of state variables and database tables. For state based DU the assignment of variables and database update, delete, insert operations were considered as definitions. On the other hand, any reference of state variables and database select operation were uses of the definitions. For constructing a test sequence, the algorithm identified the HTTP requests that defined the state variables or database table name and then identified the usage of the variables. For value based DU, the value of the variables after a request is processed and the location of the definitions were also considered. This approach uses the existing test cases without considering the coverage achieved through those test cases. Therefore, considering coverage achieved so far and combining model coverage criteria along with the regeneration process could produce more effective test cases.

## 3.2 Coverage Analysis Approaches

Najumudheen et al. proposed a dependence graph based test coverage analysis technique for object oriented projects [5]. The proposed technique consists of three phased graph construction, instrumentation and coverage analysis. A directed multigraph was constructed using three types of vertices and seven types of edges at the graph construction phase. The vertices and edges represent different object oriented features. For example, statement vertices indicate simple statements and method call edges indicate method invocations. At instrumentation phase, the source code was instrumented for storing test execution traces. As instrumentation introduces considerable time over-head therefore user requirements were analyzed and transformed into instrumentation criteria. After the test execution, a coverage analysis was done based on some coverage criteria like - statement, branch, path, method coverage etc and the edges of the graph was

marked as covered by a graph marker algorithm. The coverage result was calculated and represented to the users in a report format. However if test regeneration had been done for the unmarked edges of the graph based on the coverage report, high coverage could have been achieved.

A state based coverage analysis and behavioral equivalence checking for C++ programs had been proposed by Heckeler et al. [45]. The approach received source Class files Under Test (CUT) and XMI formatted state diagrams as input. It then transformed the UML state model to a transition table to detect behavioral deviation from the C++ implementing finite state machine. This behavioral checking was done manually and the source code was then instrumented to encode the states of the FSM with one or more class members (attributes). All state variables are also registered manually for encoding. The proposed approach used Gcov, an open source coverage analysis tool for the state machine behavioral checking, as the tool provides line coverage and the ability to dump coverage data during execution. After the instrumentation phase, existing unit, system and integration tests were executed to find which states were covered. However, the approach uses manual instrumentation and considers only state coverage. Considering transition sequence coverage along with state coverage could ensure high integration coverage of the integration tests. Again combining test regeneration technique with the coverage analysis process could produce more accurate test suites.

An integration testing coverage tool for object oriented program had been proposed in [10]. The tool automatically received the source code files and instrumented the source code to collect coverage data while test execution. The instrumentation was done by adding specific counter variables in the class, method body. After instrumentation the source code was executed and original test cases were used. When test execution is finished, the values of the counter values were summarized and stored in a method called summary file. After execution, it identifies the uncovered methods from coverage result and creates a scenario graph using

sequence and use case diagrams. In case of uncovered methods, the tool received UML use case and sequence diagrams. A scenario graph was constructed using the sequence diagram. It then extracted path from the graph that contained uncovered method calls and created test cases for the uncovered path. The test cases contained test case id, name, description, input and expected output. However, the approach does not consider unit and intra class integration test cases and also generates inexecutable test cases.

Ferreira et al. proposed a state model based test coverage analysis tool, called MoCAT [6]. The tool was developed as part of a GUI testing environment. The tool used a UML class, state model and a test suite as input. It first received UML diagrams from user specified location and converted into XMI format. The visual UML model was translated into formal Specsharp [46] model and test cases were generated automatically using Spec Explore based on user defined parameter specifications. It then simulated the execution of the test suite over the model to determine the coverage achieved through the test suite. The MoCAT tool supported transition and state coverage criteria and the coverage result gained through the test suite was represented in a colored UML state machine model to the tester. However the tool ends its task by only concluding the incompleteness of the test suite. Therefore, a tester would have to manually prepare test cases for the uncovered elements.

## 3.3 Related Approaches

Fraser et al. has proposed a tool called, EvoSuite for automatic test generation of object oriented programs [3]. The tool used an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time. The approach divided its task into two steps, whole test suite generation and mutation based assertion generation. It implements whole test suite genera-

tion as a search based approach. A population of candidate solutions is evolved using operations as crossover and mutation. The candidate solutions were taken based on their fitness value representing how close they are to the solution. A mutation testing approach was taken for the effective assertion generation and it was implemented as a tool. For assertion generation, artificial defects (mutants) were seeded into different places of the program and test cases were evaluated with respect how many mutants were identified through the test cases. Therefore, a reduced set of assertions was selected that is sufficient to find out all mutants. However, the approach does not consider model requirements and coverage while generating test cases.

An automatic test generation technique to detect operational, use case dependency and scenario faults had been proposed by Srama et al. [9]. The technique generated test cases from Use case and sequence diagrams. The technique converted the use case and sequence diagrams into use case Diagram Graph (UDG) and Sequence Diagram Graph (SDG) considering use case actors, start and end states, message invocation etc. It then integrated the UDG and SDG graphs into a system testing graph. It then generated test cases based on all use cases and sequence path coverage criteria. The technique also identified three types of faults: use case initialization faults, use case dependency faults and operational faults. However, the test generation process totally ignores source syntax information and intra class interactions of the available model elements. Nahar et al. had proposed a framework for automatic test generation using software semantics and source syntax [4]. The technique collected software semantic information by parsing XML formatted UML class, state, and sequence diagrams. It also extracted software syntax information from the source code files. Comparing consistency between semantic and syntax information, it generated unit and inter class integration test cases. However, the technique did not consider the valid sequence of interactions, as well as it also ignored intra class integration test cases.

Chen et al. has proposed a technique for automatic test case generation for UML activity diagrams [7]. The approach first randomly generated abundant test cases for a JAVA program under testing. Then, by inserting probes into the member function of the source code and running the program with the generated test cases, corresponding program execution traces were identified. The approach considered activity coverage, transition coverage and simple path coverage criteria. Based on these coverage criteria, it identified test cases that exercised the activity diagram elements and constructed a reduced set of test cases. Instead of reducing the test suite, regenerating new test cases ensuring the predefined coverage could make the test suite more accurate. Dehla et.al. proposed a technique for generating test cases from a combination of sequence and state diagrams in [8]. This technique did not consider model requirements and coverage for test generation and as a result, failed to achieve higher coverage.

## 3.4 Summary

Automatic test case regeneration from existing test suite allows the tester to regenerate test cases for untested elements in a cost effective way. A small number of untested requirements lead to unknown risks and manual inspection of the whole model. Therefore, research in the field of automatically fixing untested elements is needed. Various approaches for identifying untested elements have been applied by the researchers. However, those approaches do not fix the untested elements automatically. Therefore, further research in automatically regenerating test cases for the untested elements of the software model can come up with a potential solution. The proposed automatic test case regeneration approach will be discussed in the next chapter.

# Chapter 4

# An Automatic Test Regeneration Technique for Improving State Model Coverage

In this chapter, a technique for automated test case regeneration ensuring state model coverage is proposed. As mentioned in the previous section, existing test regeneration approaches ignore to cover all the requirements of the software model because they regenerate test cases considering combination of existing test cases or using evolutionary approaches. On the other hand, coverage measurement tools conclude their task by only identifying the tested and untested elements. So the tester needs to manually rewrite the test cases for the untested parts. In some cases the test generation algorithms are changed or test suite generation task is performed again. To overcome the above limitations, an automated approach is required which incorporates coverage result of existing test suite with the test case regeneration process to ensure model coverage, and can be combined as a module with existing test case generation techniques. Therefore, an automated test regeneration technique that considers the coverage analysis result for achieving state model coverage is proposed.

## 4.1 Overview of the Proposed Test Regeneration Technique

The proposed technique measures model coverage result by existing test suites and incorporates the result with test regeneration process to ensure high model coverage of the system. The top level overview of the proposed technique artifacts is shown in Figure 4.1. The artifacts perform core tasks for the functionality of the technique. The design of the overall technique is constructed in a way, so that the coverage measurement result of the state model can be used while test regenerating process. The *Source Code Parser*, *Test Suite Parser* and *UML Parser* shown in

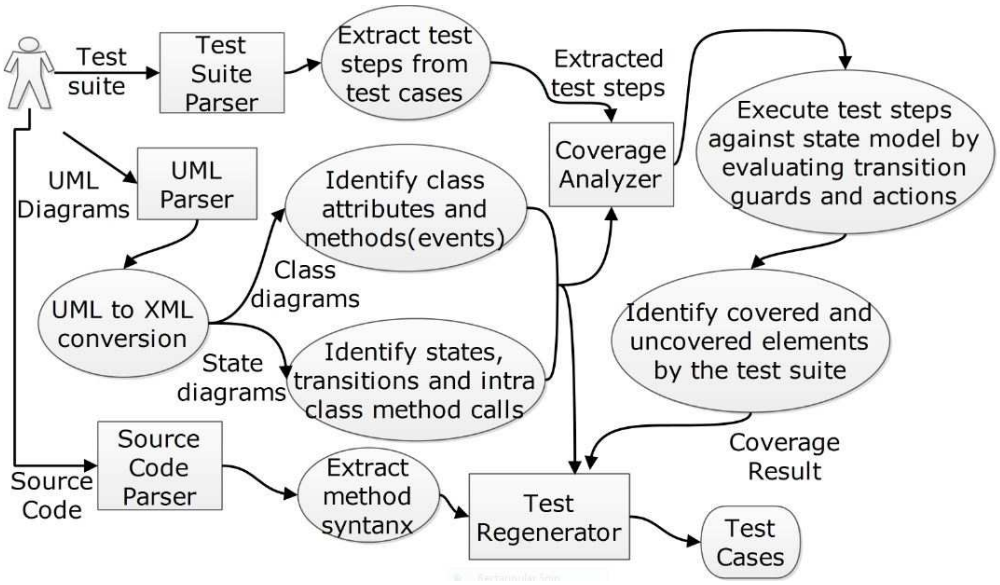


Figure 4.1: Top Level View of the Test Regeneration Technique

Figure 4.1 process the inputted data into a structured form. This modules extract information from the source code, test suite and UML class, state diagrams which are provided by the user. The extracted information is used later by the *Coverage Analyzer* and *Test Regenerator*.

*Coverage Analyzer* of Figure 4.1 considers each test step of a test case which is a method call event along with parameter values, and visits the possible transitions of the state diagrams based on the test step by evaluating the transition guard

and actions. It then identifies the covered and uncovered elements of the state model based on state, transition and simple path coverage criteria and measures the coverage result achieved through the existing test suites.

*Test Regenerator* illustrated in Figure 4.1 performs the task of test regeneration. It uses the coverage result, extracted UML elements and method syntax by the *Source Parser* to regenerate unit and integration test cases, for uncovered states, transitions and intra class method call sequences.

## 4.2 Internal Architecture of the Test Regeneration Technique

The internal architecture of test regeneration technique shown in Figure 4.2.The architecture is divided into three major modules. Each module performs some predefined responsibilities.To aid the implementation of the modules, each module consists of some components which support its functionality.
The overall implementation architecture of the proposed technique is divided in to the three following modules :

- Input Parser Module
- Coverage Analysis Module
- Test Regeneration Module

The functionalities of these modules are described below.

### 4.2.1 Input Parser

The *Input Parser Module* acts as input data provider for the proposed technique. It receives XML formatted UML diagrams, source code and test suite files from a user defined location. The *Input Parser Module* consists of three components

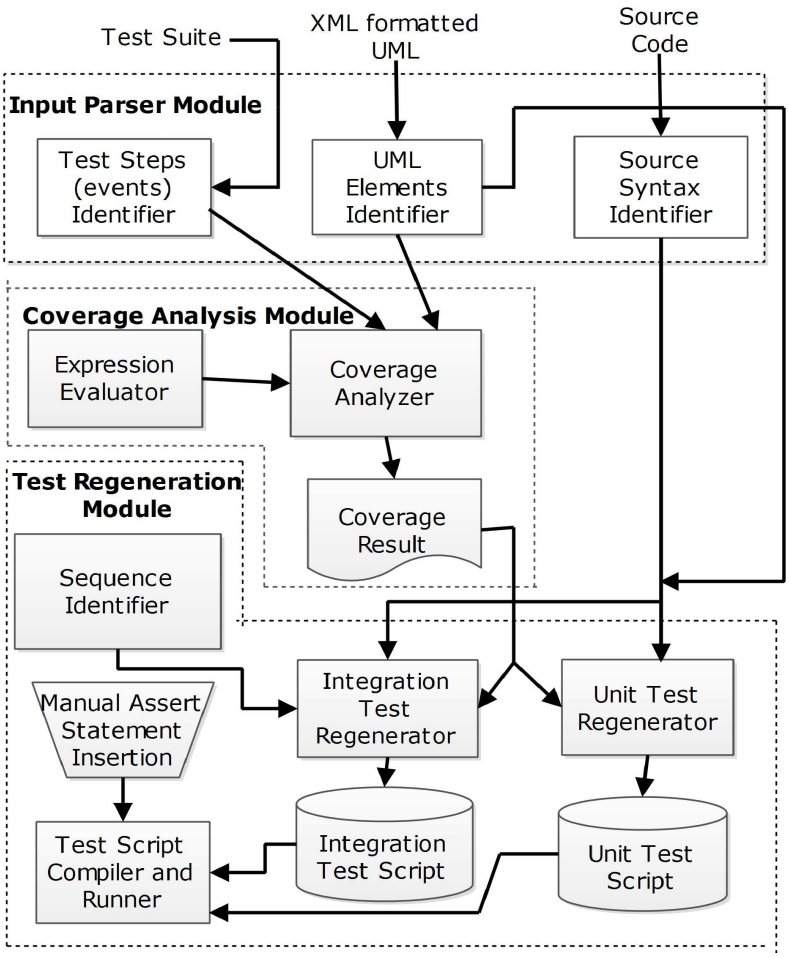


Figure 4.2: Internal Modules of the Test Regeneration Technique

– *Source Syntax Identifier*, *Test Steps Identifier* and *UML Element Identifier* as illustrated in Figure 4.2.

The *Source Syntax Identifier* component extracts class object construction, attributes, method syntax from the existing source code files, and passes those to *Test Regeneration* module. On the other hand, *Test Steps Identifier* component shown in Figure 4.2 reads the java files containing existing test suites from predefined location. It then reads the files and parses each test case step by step. From each test case, it extracts each method call statement along with its parameter values as a test step and passes those to the *Coverage Analysis* module.

*UML Element Identifier* receives XML formatted UML class and state diagrams. It then identifies class attribute name, their initial values, method name, method

parameter list from class diagrams. As each state machine is usually associated to a class, the class methods define the signature of transition call events in the state machine. The information of states, transitions, corresponding guards and action events are also extracted from state diagrams. This information is used later by the *Coverage Analysis* and *Test Regeneration* module as shown in Figure 4.2 to measure model coverage and regenerate unit and integration tests.

**Algorithm 1** Coverage Analysis Algorithm

**Input:** Existing $TestSuite$
**Output:** Marked state diagram based on coverage

1: **Begin**
2: **for** each $TestScript \in TestSuite$ **do**
3: get state diagram of $TestScript$ corresponding class
4: **for** each $TestCase \in TestScript$ **do**
5: initialize an empty list $V$ to store state variables ,class attributes and insert all variables into $V$
6: $currentState \leftarrow$ GETSTATE($TestStep[0]$)
7: **for** each $TestStep \in TestCase$ **do**
8: Initialize an empty list $T$ of possible transitions
9: Initialize boolean variables $S, G$
10: $Transitions \leftarrow$ GETOUTGOINGTRANSITIONS($currentState$)
11: **for** each $t \in Transitions$ **do**
12: $S \leftarrow$ MATCHSIGNATURE($t.event, TestStep$)
13: $G \leftarrow$ EXECUTEEXPRESSION($t.guard, V$)
14: **if** $S$ AND $G$ **then**
15: insert $t$ into $T$
16: **end if**
17: **end for**
18: **if** $T = 1$ **then**
19: EXECUTEEXPRESSION($t.action, V$) and mark $currentState$ and $t$ as covered
20: $currentState \leftarrow t.destinationState$
21: **else if** $T > 1$ **then**
22: Continue with next test case
23: **end if**
24: **end for**
25: **end for**
26: **end for**
27: **End**

### 4.2.2 Coverage Analysis

*Coverage Analysis* module is responsible for the computation of the coverage achieved by existing test suite. Coverage criteria must be measurable and be an indicator of the test adequacy. One of the coverage criteria applicable to system state models is transition based coverage. All state and transition coverage represent the full coverage of all methods of the class that is unit test and all simple path coverage represents the full coverage of intra class method call sequence as well as integration test. Therefore, the proposed technique supports state, transition and all possible simple path coverage criteria. A state is considered fully covered if all possible outgoing transitions from the state is traversed at least once. A partially covered state represents a subset of all possible transitions out of the state is exercised. An uncovered state is a state which is never reached through the test suite. This coverage criteria is used by the *Coverage Analyzer* component later to identify fully, partially covered and uncovered states. This module consists of two components – *Expression Evaluator* and *Coverage Analyzer*.

The *Coverage Analyzer* component of this module receives structured UML class, state diagram elements and extracted test statements from *Input Parser* module as shown in Figure 4.2. It then executes the test suite against the state models based on the algorithm shown in Algorithm 1. The algorithm receives test suite files of the project as input. It is assumed that each class has one intra class integration test script associated to it. For each test script, the algorithm identifies the corresponding state diagram as shown in line 3 of Algorithm 1. In this step, the algorithm gets the extracted UML class and state elements from the *Input Parser* module. The extracted class diagram information of UML contains class name, attribute types, initial value, method name, method parameter name, type and method return type. The extracted state diagram elements contains state name, transition, guard condition of each transition (if any) and corresponding action of each transition (if any). The guard condition of a transition is a mathematical

boolean expression which decides whether a transition will be taken or not. The action of a transition is an assignment operation which occurs after a transition is taken. It generally is used to update value of state variables of the state diagram. A state variable is one of the set of variables that are used to describe the status of the states and changes with the flow of transition.

For each test case in the test suite, the class attributes and state variables, and their corresponding initial values are also loaded by the program as illustrated in line 5. Once all the attributes and state variables are loaded, the algorithm then executes the test cases of the test suite. For the first test step of each test case the corresponding *currentState* is identified from the state diagram in line 6. For each test step an empty list $T$ for enlisting possible transitions and three empty boolean variables $S,G$ are initialized in line 8 and 9 respectively. In line 10 function *GetOutGoingTransitions* returns all the outgoing transitions from the current state. This function takes the *currentStep* as parameter and identify the set of outgoing transitions from the *currentStep*.

For each outgoing transition $t$ in list *Transitions*, function *MatchSignature* checks whether the test step method call signature matches with the event call signature of transition $t$. While matching the method call signature, *MatchSignature* method checks the method name as well as method parameter list and assigns a boolean value to variable $S$ in line 12. *ExecuteExpression* evaluates the boolean guard expression of tansition $t$. This method evaluates the condition based on the value of method arguments and state variables. It returns a boolean value that is assigned to variable $G$ as illustrated in line 13. A possible transition $t$ from the current state is selected and inserted into list $T$, if its event call signature matches with the test step and its corresponding guard condition evaluates to be true as shown in line 14 of Algorithm 1.

If a possible transition is found, the corresponding action of the transition is also executed by the *ExecuteExpression* function as illustrated in line 19 and the cur-

rent state and transition are marked as covered. Both the guard condition and the action expression are performed by the *Expression Evaluator* component of Figure 4.2. Line 20 of the algorithm then updates the *currentState* with the corresponding transition destination state for the next iteration. If more than one transition is found, it is discarded and the process continues with the next test case as shown in line 22. The similar process continues for all the test scripts. The coverage result is stored in a document to compare with the coverage achieved by the regenerated test cases.

### 4.2.3 Test Regeneration

*Test Regeneration* module performs the major responsibility of regenerating test cases for the uncovered paths and transitions. There are three components that support the whole test regeneration procedure illustrated in Figure 4.2. As mentioned above states and transitions represent available methods of a class. Therefore, full state and transition coverage ensure high coverage of unit test cases. *Unit Test Regenerator* and *Integration Test Regenerator* performs the task of generating test cases. Before generating test cases, the event call structure of each uncovered transition is matched with source code syntax to ensure consistency. For unit test generation, only uncovered transitions is checked. In case of integration test cases, possible uncovered path sequences are also needs to be extracted.

The *Integration Test Regenerator* component requires the uncovered intra class method call sequences for regenerating integration test cases. The *Sequence Identifier* component supports the *Integration Test Regenerator* by providing the uncovered method call sequences. The *Sequence Identifier* receives the coverage result produced by *Coverage Analyzer* as input. To generate integration tests the uncovered path of the state diagram needs to be extracted. Therefore, this component applies Depth First Search algorithm on the state model. The state machine model of a system is a directed graph and all possible simple paths that

is paths which do not contain same transition twice and contains uncovered transition are selected for test case regeneration. The Integration Test Regenerator component also takes the source code syntax information and UML information. Before generating test cases, this component matches the event call signature of uncovered transitions derived from state diagram with the actual method syntax of source code to avoid inconsistent and inexecutable test cases. It then generates integration test cases based on the extracted paths and stores the newly generated test cases in the test script.

The *Unit Test Regenerator* component performs the test generation task in a similar form. It only requires the uncovered transition and checks the consistency between source and UML method call signature for regenerating unit test cases as shown in Figure 4.2.

The *Manual Assert Statement Insertion* component requires human interaction to manually set the assert statements and method parameter values by replacing the default values in the test scripts. Although the proposed approach is implemented in java, the concept of the proposed technique is platform independent. Therefore, *Test Compiler and Test Runner* of Figure 4.2 is responsible for setting up required libraries like JUnit for Java programs to run the test scripts. The regenerated test cases can be executed against the state model in the similar way to check the increased coverage.

## 4.3 Summary

This chapter demonstrates the overall architecture of the proposed technique. It also briefly describes the internal components of the technique. The interactions among the components and the information flow through the full technique is described step by step. The three major modules *Input Parser Module*, *Coverage Analysis Module* and *Test Regeneration Module*, which support the implementa-

tion of the proposed technique are briefly described. Each component of those modules and their functionalities are also outlined. To find out the effectiveness of the proposed technique, an application of its methodology on experimental projects is required. Therefore, in the next chapter, the result of the proposed technique on three experimental projects is analyzed and described.

# Chapter 5

# Implementation and Result Analysis

This chapter demonstrates the effectiveness of the proposed technique by applying it on three experimental projects. The effectiveness is measured in terms of transition as well as transition sequence coverage, state coverage and the number of valid executable test sequence generated by the proposed test regeneration approach. This demonstration represents how the proposed technique can mitigate the limitations of the existing techniques which justifies the proposed technique. An archetype of the proposed technique was implemented in java programming language. The experimental projects which were used for result analysis were also prepared in the same environment. After that, different experimentation were done to prove the effectiveness of the proposed technique. The coverage achieved by both the existing test suite and regenerated test suite is measured by existing coverage analysis technique MoCAT [5].

## 5.1 Environmental Setup

The proposed technique was developed using Eclipse Juno Version 4.2 [47] which provides facilities to use JUnit [48] testing framework for creating and running test cases. The proposed technique evaluates the guard and action condition of the state diagram transitions. Therefore, Java Expression Parser [49] library was used for parsing and evaluating those mathematical conditions. For checking the method syntax with state diagram method call structure, the source syntax information was parsed using Java Parser [50]. The UML state, class diagrams were inputted in a XML format using an open source UML modeling tool, Enterprise Architect [51]. Following is the list of tools used for development and result analysis of the proposed technique:

- Eclipse Juno Version-4.2 [47]
- Java Expression Parser (JEP) [49]
- Java Parser Version-1.0.9 [50]
- Enterprise Architect (EA) [51]
- Eclipse Plugin JUnit 4 [48]

The development and result analysis of the proposed technique was done on the following desktop configuration -

- CPU : 3.20 GHz Intel Core i5 Processor
- RAM : 4.00 GB
- Operating System : Windows 7 Ultimate
- Platform : 32 bit

For running the proposed technique on experimental projects some preorientation of the input files needs to be done. As the proposed technique takes source code,

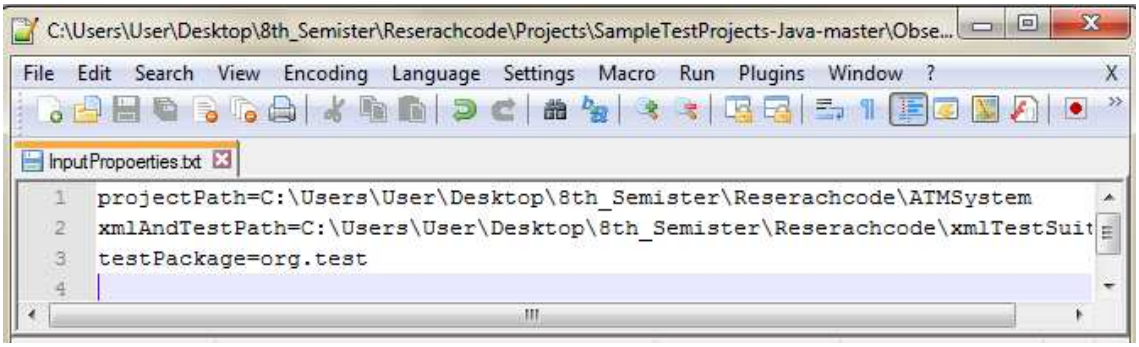


Figure 5.1: Sample Input Property File

UML class, state diagram and existing test suite as input therefore, these input files needs to be placed in predefined directory structure. The path for an experimental project which will be analyzed, is specified in a text document as shown in Figure 5.1. The corresponding UML diagrams and test suite are also read from a specified path. The corresponding directory structure of the specified path for UML diagrams and test suite input is shown in Figure 5.2. As all the state diagrams of a project are generated under the class diagram of corresponding project using Enterprise Architect, therefore, there is only single XML file input. The test files are placed in a separate folder in the same directory structure of Figure 5.2. If no test script is available for corresponding each inputted state diagram, a new test script is generated for storing the regenerated test cases. If a test script is already available, regenerated test cases are appended to the existing test script for achieving improved coverage.

The proposed technique regenerates test cases based on the existing test suite and coverage result. Two test case regeneration tools are available in literature. The first tool is SART [12], which is a test case regeneration tool for improving web application branch coverage and fault detection. The tool is based on web applications and fault detection, therefore the regenerated test cases of this tool cannot be compared with the regenerated test cases of the proposed technique, as the context of regenerating test cases of SART (web based branch coverage) and proposed technique (state model based intra class interaction coverage) is dif-

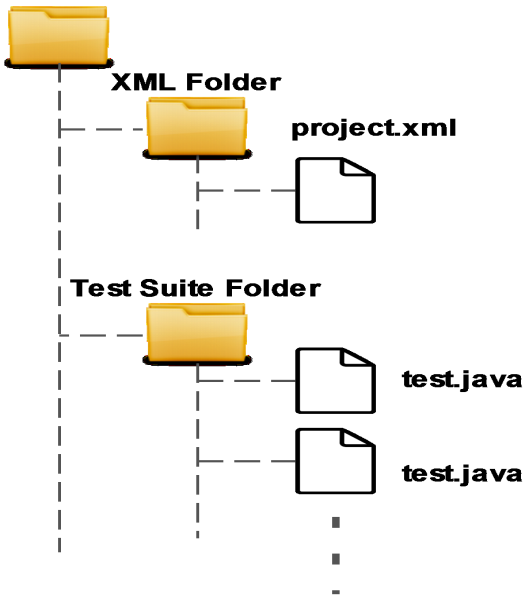


Figure 5.2: Directory Structure for XML File and Existing Test Suite

ferent. On the other hand, SPM-RGN [11] tends to regenerate test cases using sequential pattern mining to cover source code based statement and branch coverage, ignoring model coverage criteria. Hence, comparing the regenerated results of SPM-RGN with the proposed technique will not be fruitful. As the proposed technique considers state based model coverage criteria while regenerating test cases, therefore the results of the proposed technique can be applied on any existing technique that considers both model based and source based information while generating test cases.

SSTF [4] is an automated test generation technique that considers both UML class, state, sequence diagram and source code based syntax information while generating test cases. Therefore, the proposed technique can be applied on existing test suite generated by SSTF to measure the effectiveness of regenerated test suite of the technique. Hence, SSTF generated test suite is considered for result analysis and the effectiveness of the proposed technique is estimated by measuring the extent of coverage improvement, it can add to the existing test suite. The coverage of SSTF generated test suite and the proposed technique regenerated test suite is measured using existing coverage analysis tool MoCAT [6].

## 5.2 Experimental Projects

For result analysis, the proposed technique is applied on three open source projects which are listed in Table 5.1 and also available at [52]. The table represents the project name along with the number of classes in the class diagram, the number of state diagrams available, and number of test scripts in the test suite which are generated using SSTF. All these projects are implemented considering the actual model requirements, therefore are taken as experimental projects for analysis.

'Alarm System' is a simple java project that has a control panel which is connected

Table 5.1: Experimental Projects

| Project Name | No of Classes in Class Diagram | No of State Diagram | No of TestScript |
|---|---|---|---|
| Alarm System [6] | 2 | 1 | 1 |
| ATM System [52] | 4 | 2 | 4 |
| Observer Pattern [4] | 5 | 1 | 1 |

to one siren and multiple sensors. To activate the system, the user has to press a specific button in the control panel. To deactivate the system, the user has to insert the right pin. After 3 failed attempts to insert the right pin, the system becomes locked. This project was used for state based coverage analysis in [6]. A class diagram containing 2 classes, 1 state diagram and 1 test script generated by SSTF, are inputted for experimentation on this project as shown in Table 5.1.

'ATM System' has some user accounts where the accounts are protected through account number and pin number. If a user enters the right pin number, the user is allowed to perform some transactions like checking balance, withdrawing and depositing money. A user is allowed to withdraw money if enough money is available in the corresponding user account and in the system cash dispenser. After two failed attempts to insert a pin, the account is locked. This project was developed for academic purpose in Institute of Information Technology. It contains 4 classes, 2 state diagrams and 4 test scripts.

The 'Observer Pattern' is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods [53]. This project was used for automatic test generation in [4] and consists of 5 classes, 1 state diagram and 1 generated test script as input.

## 5.3 Metrics Used to Analyze Results

As the proposed technique regenerates test cases considering coverage result of existing test suite, therefore the regenerated test suite achieves improved coverage than the existing test suite. The improvement result is analysed by the number of covered transition sequences, states in the state diagrams and the number of executable regenerated test cases.

If full transition coverage is achieved for a state diagram, it refers that all the method call interaction in the corresponding source class are tested. Transition coverage is the ratio of the number of covered transitions and total transitions in the state diagrams. The ratio is measured using the following equation which is modified based on [10] -

$$TransitionCoverage = \frac{\sum_1^n T_n\,(Covered)}{\sum_1^n T_n\,(Covered) + T_n\,(Uncovered)} \tag{5.1}$$

where $T_n$ represents the number of transitions in the $n$-th inputted state diagram. On the other hand, 100% state coverage of a state diagram depicts that all possible method invocation from the states of the class are tested. State coverage is the ratio of the number of covered states and total number of states in the state diagrams which is measured using the following equation which is also modified based on [10]-

$$StateCoverage = \frac{\sum_1^n S_n\,(Covered)}{\sum_1^n S_n\,(Covered) + S_n\,(Uncovered)} \tag{5.2}$$

Table 5.2: The Number of Covered and Uncovered Transitions of SSTF Generated Test Suite and Proposed Technique Regenerated Test Suite Measured by MoCAT, and the Number of Regenerated Integration and Unit Test Case

| Project Name | No of Transitions by SSTF Generated Existing Test Suite | | No of Transitions by Proposed Technique Regenerated Test Suite | | No of Regenerated Test Cases using Proposed Technique | |
|---|---|---|---|---|---|---|
| | Covered | Uncover -ed | Covered | Uncover -ed | Integration Test Case | UnitTest Case |
| Alarm System | 8 | 9 | 15 | 2 | 7 | 9 |
| ATM System | 7 | 9 | 15 | 1 | 3 | 5 |
| Observer Pattern | 2 | 5 | 7 | 0 | 1 | 2 |

where $S_n$ represents the number of states in the $n$-th inputted state diagram.

The proposed technique checks the source code method syntax while regenerating test cases, as a result executable test cases are generated. However, the effectiveness of the technique is also measured by checking the number of test cases consisting valid integration paths. The result is represented by the ratio of valid executable test sequences and total generated test sequences. The ratio is calculated using the following equation -

$$ValidSequenceRatio = \frac{Seq_{Valid}}{Seq_{Valid} + Seq_{Invalid}} \tag{5.3}$$

## 5.4 Result Analysis

The proposed technique is applied on the above experimental projects and based on the existing test suite and coverage analysis result, test cases are regenerated for achieving high coverage. The number of regenerated integration and unit test cases for each experimental project as well as the number of covered and uncovered transitions by the regenerated test suite are also enlisted in Table 5.2. The table clearly shows that for each project the number of covered transition of SSTF generated test suite is low, whereas significant increase in the number of covered

Table 5.3: The Improvement in Transition Coverage of SSTF by the Regenerated Test Suite of the Proposed Technique

| Project Name | Transition Coverage Achieved by SSTF Generated Existing Test Suite | | | Transition Coverage Achieved by Proposed Technique Regenerated Test Suite | | | Improv -ement Ratio % |
|---|---|---|---|---|---|---|---|
| | Covered (Trans.) | Uncover -ed (Trans.) | Ratio % | Covered (Trans.) | Uncover -ed (Trans.) | Ratio % | |
| Alarm System | 8 | 9 | 47 | 15 | 2 | 88.25 | 41.25 |
| ATM System | 7 | 9 | 43.75 | 15 | 1 | 93.75 | 50 |
| Observer Pattern | 2 | 5 | 28.75 | 7 | 0 | 100 | 71.43 |

transition is achieved by applying the proposed technique. For 'Alarm System', the number of covered transition is only 8 and uncovered transition is 9. After applying the proposed technique, 7 integration and 9 unit test cases are generated which improves the number of covered transition to 15.

The proposed technique intends to achieve high transition and state coverage. Therefore, the effectiveness of the technique is represented by the ratio of the number of covered transitions and states of the state diagrams. The number of covered and uncovered transitions for each experimental project, achieved by SSTF generated existing test suite is also measured by the same existing coverage analysis tool, MoCAT [6]. The ratio of transition coverage result achieved by existing test suite and the regenerated test suite is calculated using Equation 5.1 and is enlisted in Table 5.3. In every experimental project the ratio of the transition coverage acheived by the proposed technique is twice than the existing coverage ratio. The improvement ratio represents how much improvement in transition coverage of SSTF generated test suite is achieved by applying the proposed technique. The results clearly depicts that on average 54.23% coverage improvement is achieved by the proposed technique.

Table 5.4: The Improvement in State Coverage of SSTF by the Regenerated Test Suite of the Proposed Technique

| Project Name | State Coverage Achieved by SSTF Generated Existing Test Suite | | | State Coverage Acheived by Proposed Technique Regenerated Test Suite | | | Improv -ement Ratio % |
|---|---|---|---|---|---|---|---|
| | Covered (States) | Uncover -ed (States) | Ratio % | Covered (States) | Uncover -ed (States) | Ratio % | |
| Alarm System | 2 | 4 | 34 | 5 | 1 | 83.33 | 49.33 |
| ATM System | 8 | 6 | 57.14 | 7 | 1 | 87.50 | 30.36 |
| Observer Pattern | 3 | 3 | 50.00 | 6 | 0 | 100 | 50 |

Table 5.4 represents the state coverage ratio of covered and total state numbers in the inputted state diagram of the experimental projects. The results clearly show that only 47% average state coverage is achieved using the existing test suite, as the technique ignores intra class interactions while generating test cases. On the other hand, average state coverage gained through the regenerated test suite is nearly 90%. The remaining 10% coverage could not be achieved as proper test data was not set in the test method parameters by the testers. However, the improvement ratio represents that on average 43.23% improvement in SSTF generated test suite is achieved by the regenerated test suite.

Figure 5.3 and 5.4 represents the measure of transition and state coverage respectively by the existing and regenerated test suite on the experimental projects. In each case, the proposed technique has been successful to increase the coverage of existing test suite. The bar charts in Figure 5.3 and 5.4 show that for each project, the regenerated test suite has higher bar which represents significant improvement in the coverage achieved.

Table 5.5 represents regenerated Valid Sequence Ratio, which is measured by Equation 5.3. The table depicts that nearly 100% of the regenerated test cases are valid executable test sequence. However, in the case of 'Alarm System', one

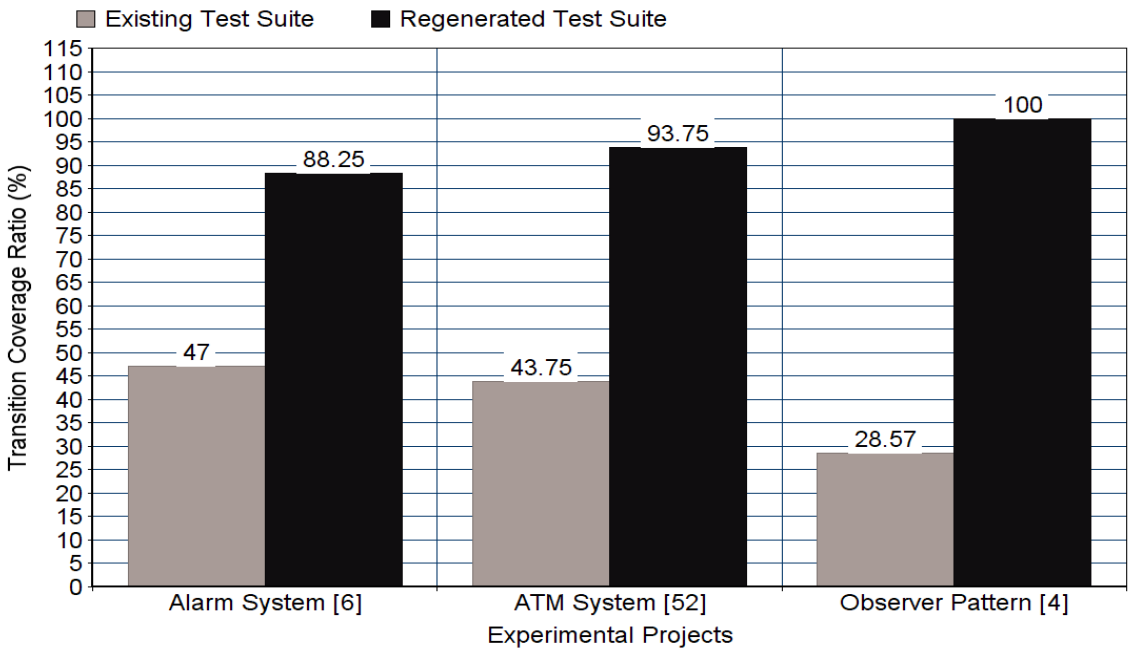


Figure 5.3: Comparison of Transition Coverage achieved by Existing and Regenerated Test Suite

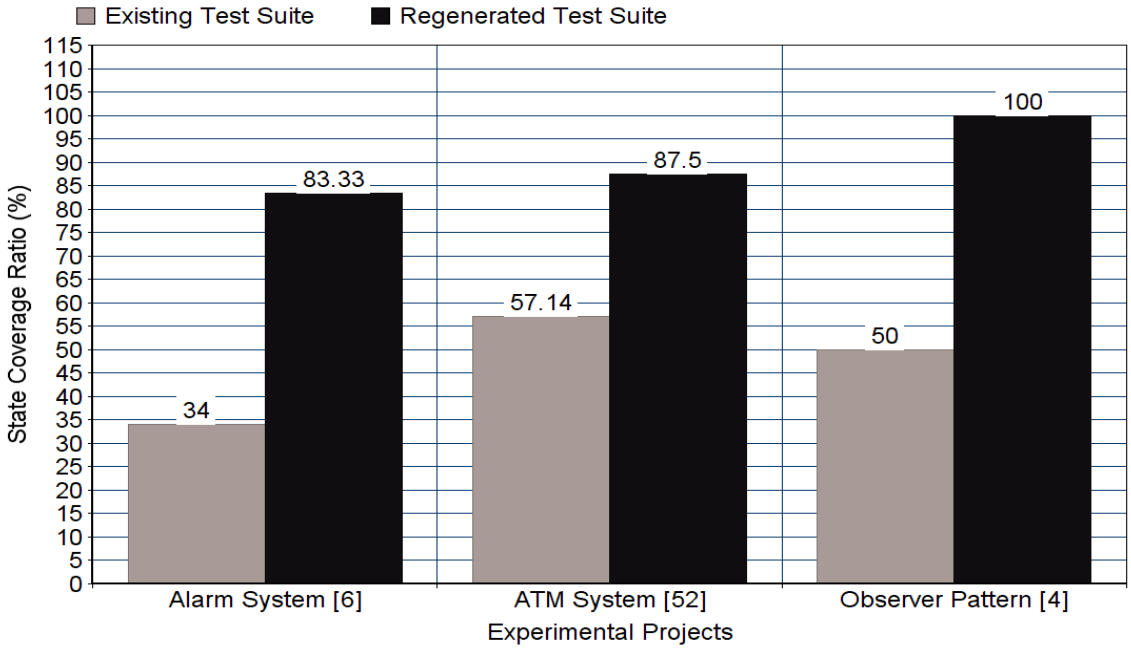


Figure 5.4: Comparison of State Coverage achieved by Existing and Regenerated Test Suite

Table 5.5: The Number of Regenerated Test Cases by the Proposed Technique and Ration of Valid Executable Test Cases

| Project Name | No of Valid Executable Regenerated Test Cases | No of Invalid Test Sequence | Ratio |
|---|---|---|---|
| Alarm System | 15 | 1 | 93.75% |
| ATM System | 8 | 0 | 100% |
| Observer Pattern | 3 | 0 | 100% |

invalid test sequence is generated, as there is a self loop sequence in that path and the technique considered simple path sequences. It is difficult to know the loop breaking constraint of the states in advance from model elements and hence, leads to different research field. However, in the rest of the cases almost all the regenerated test cases are valid executable test sequences.

## 5.5 Summary

This chapter depicts the overall procedure of the implementation and result analysis of the proposed technique. An archetype of the technique was developed using java programming language. The technique was applied on three experimental representative projects. The result is analyzed on the basis of transition and state coverage and number of valid executable test sequences. The coverage achieved by existing and regenerated test suite is measured by existing coverage analysis tool, MoCAT [6]. The result clearly shows that on average 54% coverage improvement can be done by the applying regenerated test suite. Again, most of the regenerated test cases are executable. In short, the proposed technique has been successful in improving coverage of the existing test suite. The next chapter concludes the thesis with identifying its limitations and discussing the future direction.

# Chapter 6

# Conclusion

Most of the existing coverage analysis techniques conclude their task by only identifying tested and untested elements by the test suite. These approaches do not regenerate test cases for the untested elements. This thesis proposes a technique that tends to regenerate test cases for those untested elements to improve the coverage of the existing test suite. The proposed technique analyzes the coverage result of the existing test suite and regenerates test cases to improve the coverage of the existing test suite. This chapter concludes the thesis with brief description about the threats as well as the future direction of the proposed technique.

## 6.1 Discussion

This section discusses the justification of the result achieved by the proposed technique. The proposed technique regenerates test cases by using the knowledge of the existing test suite, hence it achieves increased coverage than the existing test suite. The technique considers the existing coverage result while regenerating, thus it produces valid test sequences which are required to cover the untested parts by the existing test suite.

This thesis introduces a test case regeneration technique to improve state model coverage, that analyzes the coverage achieved so far, and uses both the system

UML elements and source syntax for the regeneration. The Parser, Coverage Analysis and Test Regeneration module operate together to regenerate unit and integration test cases. While Input Parser module processes UML, source and test suite information provided by the user, Coverage Analysis module identifies the covered and uncovered elements of the state model. Test Regeneration module considers the coverage result and extracted information by the Parser and regenerates unit and integration test cases.

Manual efforts in searching and producing test sequences that covers the elements of the software product is time consuming and laboursome. The proposed technique not only allows the tester to achieve high test coverage but also produces executable test cases. It is clearly seen in Chapter 5 that, the regenerated test suite improves the transition and state coverage of the SSTF generated existing test suite. The reason is that, the existing technique ignores intra class interaction sequences as well as coverage criteria. Most of the test generation techniques only focuses on inter class integration tests. SSTF generates both unit and integration test cases, however it ignores intra class transition sequences while generating test cases. The task of finding those uncovered parts and completing those by executable test cases is done easier by the proposed technique. The technique finds the uncovered paths and regenerates executable intra integration test cases as well as unit test cases for the uncovered transition sequences. Hence, the technique can be added as a module with existing test generation approaches to improve the effectiveness and reusability of the existing test suite. The technique successfully increases the transition coverage and state coverage of the SSTF generated test suite on average by 54.23% and 43.23% respectively.

The regeneration of the test cases is done with consideration of both model structure and source syntax. Before regenerating test cases, the state model method call signature is checked with the source method invocation syntax, thus error less test cases could be created.

## 6.2 Threats to Validity

This section discusses the threats which can affect the validity of the proposed technique. The threats are identified from three perspective - internal threats, external threats and construct threats.

**Internal Threats:** The internal threats refers threats that affect the validity of the results depend on the implementation of the technique and the environmental set up of the experimental procedure. The proposed technique as well as the experimental projects are implemented in java programming language. Therefore, the result gained through analyzing the experimental projects can differ when experimented in platforms other than java.

**External Threats:** The experimental projects that are chosen and the existing test suites used may affect the degree to which the results can be generalized. The experimental projects which are chosen are used in existing techniques (For example, 'Alarm System' was used in [6]). Those projects are also selected as those have associated state models. The existing auto generated test suite and UML class, state model diagrams which are inputted in the technique can also affect the results gained from the experiments.

**Construct Threats:** Construct threats are related to the metrics which are used to analyze the effectiveness of the proposed technique. The results are analyzed based on transition coverage, state coverage and number of valid regenerated test cases. Therefore, analyzing the results with other metrics can affect the generalization of the results.

## 6.3 Future Work

The idea of regenerating test cases based on coverage result for improving state model coverage opens a number of directions for future research. The incorporation of sequence diagrams along with the state diagrams can ensure inter class

interaction coverage. The proposed technique measures transition sequence coverage as well as state coverage. Adding more coverage criteria like - branch coverage and guard condition coverage, introduces new research scope.

In some cases the proposed technique could not achieve 100% coverage as proper data was not set in the test method parameters by the tester. Hence test data generation technique can be incorporated. Again the projects which are used for result analysis are small scale project. Therefore, working with real life or industrial project is a future issue.

# Bibliography


[1] M. Shahid and S. Ibrahim, "An evaluation of test coverage tools in software testing," in *2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT*, vol. 5, 2011.

[2] B. Mark Utting, "Pratical model-based testing a tools approach," 2007.

[3] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419, ACM, 2011.

[4] N. Nahar and K. Sakib, "Sstf: A novel automated test generation framework using software semantics and syntax," in *17th International Conference on Computer and Information Technology (ICCIT), 2014*, pp. 69–74, IEEE, 2014.

[5] E. Najumudheen, R. Mall, and D. Samanta, "A dependence graph-based test coverage analysis technique for object-oriented programs," in *6th International Conference on Information Technology: New Generations, 2009. ITNG'09.*, pp. 763–768, IEEE, 2009.

[6] R. D. Ferreira, J. P. Faria, and A. C. Paiva, "Test coverage analysis of uml state machines," in *3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010*, pp. 284–289, IEEE, 2010.

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

[8] D. Sokenou *et al.*, "Generating test sequences from uml sequence diagrams and state diagrams.," in *GI Jahrestagung (2)*, pp. 236–240, Citeseer, 2006.

[9] M. Sarma and R. Mall, "Automatic test case generation from uml models," in *in 10th International Conference on Information Technology,(ICIT 2007)*, pp. 196–201, IEEE, 2007.

[10] P. Augsornsri and T. Suwannasart, "An integration testing coverage tool for object-oriented software," in *International Conference on Information Science and Applications (ICISA), 2014*, pp. 1–5, IEEE, 2014.

[11] W. He and R. Zhao, "Sequential pattern mining based test case regeneration," *Journal of Software*, vol. 8, no. 12, pp. 3105–3113, 2013.

[12] N. Alshahwan and M. Harman, "State aware test case regeneration for improving web application test suite coverage and fault detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 45–55, ACM, 2012.

[13] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[14] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005.

[15] J. N. Buxton and B. Randell, *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[16] R. Bender, "How do you know when you are done testing?," *Cutter IT Journal*, vol. 14, no. 9, pp. 29–35, 2001.

[17] J. M. Glenford and S. Myers, "Software reliability principles and practices," 1976.

[18] I. Burnstein, *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.

[19] E. Iee, "Ieee standard glossary of software engineering terminology," 1990.

[20] P. Runeson, "A survey of unit testing practices," *Software, IEEE*, vol. 23, no. 4, pp. 22–29, 2006.

[21] Y. Cheon, M. Y. Kim, and A. Perumandla, "A complete automation of unit testing for java programs," 2005.

[22] R. Parkin and I. Australia, "Software unit testing," *The Independent Software Testing Specialists, IV & V Australia*, 1997.

[23] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Conference on Software Maintenance, 1990, Proceedings.*, pp. 290–301, IEEE, 1990.

[24] A. Orso, "Integration testing of object-oriented software," *Politecnico di Milano, Milano, Italy*, 1998.

[25] B. Meyer, "Object oriented software construction," 1988.

[26] K. Naik and P. Tripathy, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[27] L. Briand and Y. Labiche, "A uml-based approach to system testing," *Software and Systems Modeling*, vol. 1, no. 1, pp. 10–42, 2002.

[28] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[29] J. J. Marciniak and A. Shumskas, *Acceptance Testing*. Wiley Online Library, 1994.

[30] R. Miller and C. T. Collins, "Acceptance testing," *Proc. XPUniverse*, 2001.

[31] "Dynamic and static testing." `http://www.dcs.gla.ac.uk/~johnson/teaching/safety/powerpoint/15_Testing.pdf`. Accessed: 2015-12-05.

[32] E. L. Grigorenko and R. J. Sternberg, "Dynamic testing.," *Psychological Bulletin*, vol. 124, no. 1, p. 75, 1998.

[33] L. Williams, "Testing overview and black-box testing techniques retrieved from http://agile. csc. ncsu. edu/sematerials," *BlackBox. pdf*, 2006.

[34] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[35] I. K. El-Far and J. A. Whittaker, "Model-based software testing," *Encyclopedia of Software Engineering*, 2001.

[36] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing," 2006.

[37] L. Apfelbaum and J. Doyle, "Model based testing," in *Software Quality Week Conference*, pp. 296–300, 1997.

[38] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[39] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[40] G. Booch, *The unified modeling language user guide*. Pearson Education India, 2005.

[41] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for uml design models," *Software Testing, Verification and Reliability*, vol. 13, no. 2, pp. 95–127, 2003.

[42] J. A. McQuillan and J. F. Power, "A survey of uml-based coverage criteria for software testing," *Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland*, 2005.

[43] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[44] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Transactions on Software Engineering*, no. 12, pp. 1278–1296, 1987.

[45] P. Heckeler, J. Behrend, T. Kropf, J. Ruf, R. Weiss, and W. Rosenstiel, "State-based coverage analysis and uml-driven equivalence checking for c++ state machines," *FM+ AM*, vol. 179, pp. 49–62, 2010.

[46] "Spec - microsoft research." `http://research.microsoft.com/en-us/projects/specsharp/`. Online; accessed 19 December 2015.

[47] "Eclipse.org - juno simultaneous release." `https://eclipse.org/juno/`. Online; accessed 19 December 2015.

[48] "Junit - about." `http://junit.org/`. Online; accessed 19 December 2015.

[49] "Jep - java math expression parser." `http://www.cse.msu.edu/SENS/Software/jep-2.23/doc/website/doc/doc_usage.htm`. Online; accessed 19 December 2015.

[50] "javaparser 1.0.9 - maven repository." `http://mvnrepository.com/artifact/com.google.code.javaparser/javaparser/1.0.9`. Online; accessed 19 December 2015.

[51] "Enterprise architect." `http://www.sparxsystems.com/`. Online; accessed 19 December 2015.

[52] "Experimental projects." `https://github.com/Afrina/Projects`. Online; accessed 19 December 2015.

[53] "Observer pattern." `https://en.wikipedia.org/wiki/Observer_pattern`. Online; accessed 19 December 2015.