ENHANCEMENT IN EFFECTIVENESS OF TEST CASE PRIORITIZATION FOR REGRESSION TESTING

A Dissertation Submitted

By

Monika Pathania (10810760)

To

Department of CSE/IT

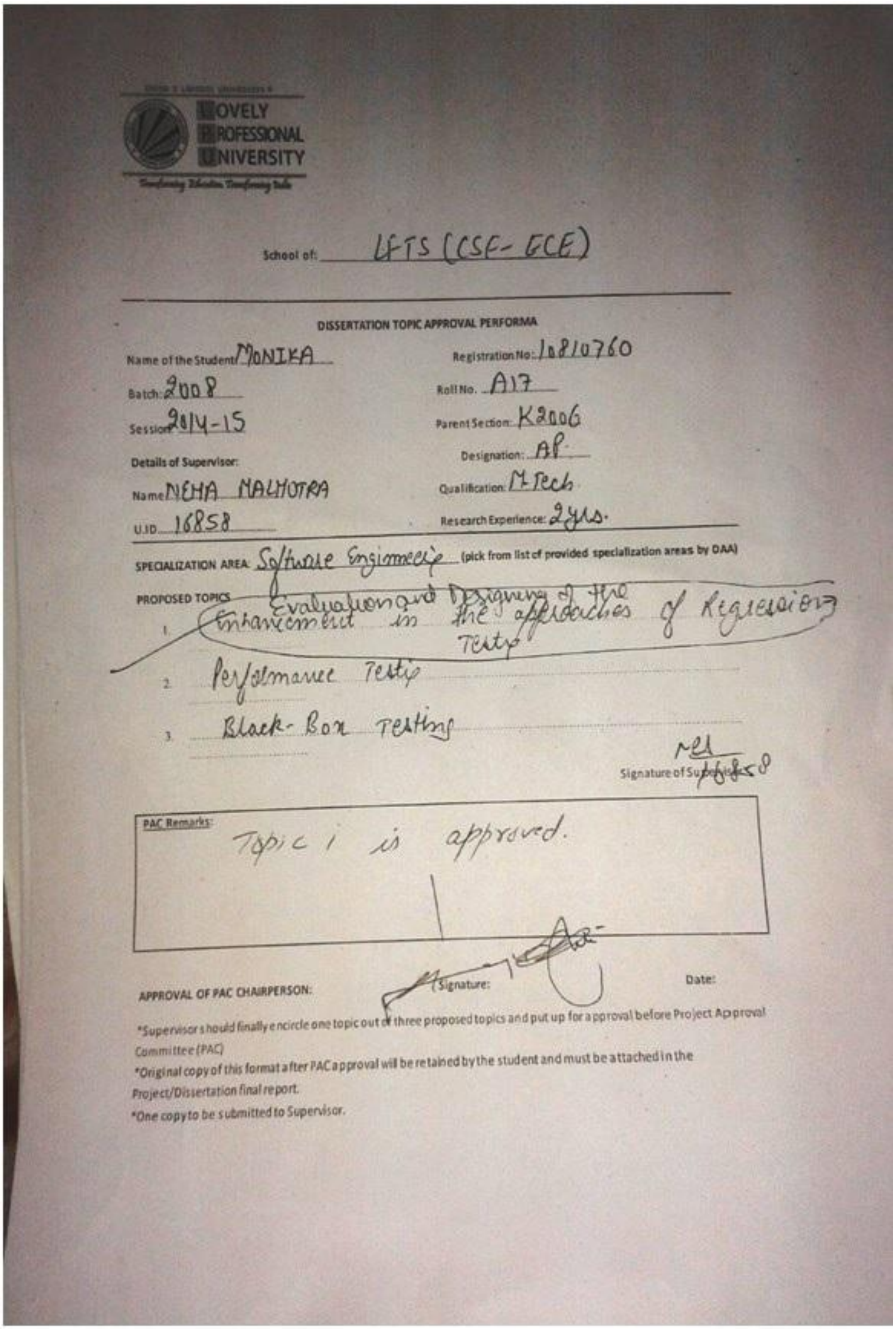
In partial fulfillment of the Requirement for the
Award of the Degree of

Master of Technology in CSE

Under the guidance of

Mr. Makul Mahajan

(April 2015)



LOVELY PROFESSIONAL UNIVERSITY

Transforming Education Transforming India

School of: LFTS (CSE- ECE)

DISSERTATION TOPIC APPROVAL PERFORMA

Name of the Student: MONIKA	Registration No: 10810760
Batch: 2008	Roll No. A17
Session: 2014-15	Parent Section: K2006
Details of Supervisor:	Designation: AP
Name: NEHA MALHOTRA	Qualification: M.Tech
U.ID. 16858	Research Experience: 2yrs.

SPECIALIZATION AREA: Software Engineering (pick from list of provided specialization areas by DAA)

PROPOSED TOPICS

1. Evaluation and Designing of the
   Enhancement in the approaches of Regression Testing
2. Performance Testing
3. Black-Box Testing

Signature of Supervisor [illegible]

PAC Remarks: Topic 1 is approved.

APPROVAL OF PAC CHAIRPERSON: Signature: [illegible] Date:

*Supervisor should finally encircle one topic out of three proposed topics and put up for approval before Project Approval Committee (PAC)

*Original copy of this format after PAC approval will be retained by the student and must be attached in the Project/Dissertation final report.

*One copy to be submitted to Supervisor.

ABSTRACT

A vast study has been made in the field of Regression Testing. Regression testing is testing in which retesting of system, components and related units is done and it is ensured that modifications made in the system are working correctly and are not effecting other modules of system. Regression testing is testing in which modified system is tested using the old test suites. The procedure is to test modified parts of the system at first tested and then the whole system needs to be retested using old test suits to make sure that modification have not affected the other parts of the previously working system. Developers are interested in detecting faults in the system as early as possible, due to the resource and time constraints. As in regression testing, re-execution of large test suites is there so these constraints are always taken into much consideration. There remains a need to prioritize test cases in order to reduce time and other resources use. So in this dissertation, work is on test case prioritization technique. Test cases were generated from model of the system that can depict the behavior of the system and (Average Percentage of Faults Detected) APFD and (Average Percentage of Block Coverage) APBC metrics were used to calculate the effectiveness of it. Execution time was also taken into consideration while prioritizing automatic generated test cases.

ACKNOWLEDGEMENT

This work on dissertation entitled as **"ENHANCEMENT OF EFFECTIVENESS OF TEST CASE PRIORITIZATION FOR REGRESSION TESTING"** prepared by me could not have been completed without the moral support of my mentor Mr. Makul Mahajan. A Special Thanks to him for guiding my efforts to perfection. I would also like to thank my father Mr. Santosh Pathania who supports me in every part of my life.
Finally, I would like to thank all my friends.

Monika Pathania
10810760

DECLARATION

I hereby declare that the dissertation entitled "**ENHANCEMENT OF EFFECTIVENESS OF TEST CASE PRIORITIZATION FOR REGRESSION TESTING",** submitted for the M.Tech. Degree is entirely my original work and all the ideas and references have been duly acknowledged. It does not contain any work for the award of any other degree or diploma.

Date:

Monika Pathania

Reg. No. 10810760

CERTIFICATE

This is to certify that Monika Pathania has completed M.Tech dissertation titled "**ENHANCEMENT IN EFFECTIVENESS OF TEST CASE PRIORITIZATION FOR REGRESSION TESTING"** under my guidance and supervision. To the best of my knowledge, the present work is the result of her original investigation and study. No part of the dissertation proposal has ever been submitted for any other degree or diploma.

The dissertation is fit for submission and the partial fulfillment of the conditions for the award of M. Tech Computer Science & Engineering.

Date: Name: Mr. Makul Mahajan

UID:14575

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Software Testing

Software testing is the process by which evaluation of software is done to detect that whether their lies any differences between the given input and expected output or not. Through this process assessment of the features of a software item is done. Now the question is “when it should be done?” Most of the developers and researchers say that software testing should be done during the development process. The process software testing is done to confirm that whether software item is build according to the customer requirements or not. Now talking about its objectives, the main objective of software testing is to find out errors in the system/software being developed or modified by applying different types of testing like performance testing, regression testing, smoke testing, stress testing. By software testing, the quality parameters like effectiveness of software can be improved. It can also be said that software testing is both verification and validation. Now what is meant by these two terms? [18].

1.1.1 Verification

Verification means “are things right or not?” There always remain some conditions which are imposed on the product at the begin of the development phase. It is the method to make positive the product satisfies all those requirements.

Verification is to make confident that the behavior of product according to us is called verification.

1.1.2 Validation

Validation means “are we doing right things?” Through validation process, we make sure that at the end of development phase, product meets the specified requirements.

Validation is to make confident the system is performing as per requirements of the customer or not.

1.2 Software Testing Types

To make the things clear some of the types of software testing are described here. In this research work, main focus is on regression testing which is described in section 1.3. For the sake of understanding other various types of testing are as follows:

1.2.1 Black Box Testing

Black box testing is a testing method in which focus is not on the internal mechanism of the system. Its main focus remains on the output which is produced next to any input provide to the system. This method of testing is also known as functional testing.

1.2.2 White Box Testing

As opposed to the black box testing, in case of white box testing the main focus remains on the code i.e. it is a testing method whose main focus is on internal mechanism of a system. This method of testing is also called structural testing or glass box testing.

“Black box testing is used for validation and white box testing is used for verification.”

There are some other approaches to testing which are mentioned below:

a) Unit Testing
b) Integration Testing
c) Functional Testing
d) System Testing
e) Stress Testing
f) Performance Testing
g) Usability Testing
h) Acceptance Testing
i) Regression Testing
j) Beta Testing

In the research proposal, main focus is on regression testing. So what this regression testing is and what are the various approaches of regression testing are as follows:

1.3 Regression Testing

Regression testing is testing in which retesting of system, components and related units is done and ensure that modification is working correctly and does not effecting other modules of system. Regression testing is testing in which old test suites are used by the modified system. In regression testing we have to test modified parts of the system firstly. After that whole system will be retested using old test suits. If large size system retesting is done then consumption of time will be large and more resources will be use. Ordering test cases for execution is the one of main issue faced by developer. Regression testing is a necessary process as well as expensive process in the software lifecycle. The main focus of regression testing is to ensure the modification is working correctly and is not harmful for other modules to produce unexpected results. Regression testing is type of black box testing. In simple language, regression testing is the method of testing changes to computers programs to make confident that the older module still works with new changes.

Figure 1.1: Regression Testing [12]

1.3.1 Regression testing Approaches

There are different types of regression testing approaches:

a) Model Based Regression Testing
b) Code Based Regression Testing
c) Selective Regression Testing

a) Model Based Regression Testing: Model based approach is the approach of regression testing which is a black-box method. This approach selects test case based on model modification and uses relationships between test cases and model elements for retest. This approach generates regression tests using different models of system. The technique use some system models like activity diagrams, class diagrams and other various UML diagrams.

b) Code Based Regression Testing: Code based technique is the technique of regression testing that it is white- box method. This approach selects test case based on the difference between original code and new modified code. Code based approach always uses relationships between test cases and code part and when code is modified then this approach order the test cases for re-execution.

c) Selective Regression Testing: Selective regression testing is the important activity of MBT (model based testing).According to Donald Bren et al [9] this technique selects test cases for retest based on model modifications, rather than source-code modifications.

1.4. Test Case Prioritization

Test case prioritization is the approach which ordered the test cases in executing manner according to some criterion. These criterion can be time and cost based. The main aim of test case prioritization approach is to detect various faults in minimum cost and time. Test case prioritization approach can be used in two applications; a).first is in the initial testing of software, b). Second is in the regression testing of software. The difference between these two used applications in regression testing that prioritization can use in sequence collected in earlier execution of existing test cases. Test case prioritization is the technique which is used to assist with regression testing. After arrangement of test cases in some criterion, test cases with higher priority are executed earlier.

1.4.1 Test case prioritization techniques

a). Model Based Test Case Prioritization

b). Code Based Test Case Prioritization

c). Requirement Bases Test Case Prioritization

a) Model Based Test Case Prioritization

Model based test case prioritization is a technique of test case prioritization in which we used the various system models to prioritize the test cases. Model based test case prioritization may help to improve early fault detection as compare to other approach of test case prioritization which is code based test case prioritization. Model based test case prioritization may be a sensitive approach for information provided by developers. So we can say that model based test case prioritization approach is best as compared to code based test case prioritization. Existing model based test case prioritization methods can only be used during system maintenance when models are modified [1]. There are many model based test case prioritization heuristics are:

- Selective prioritization
- Heuristic #1 prioritization
- Heuristic #2 prioritization
- Heuristic #3 prioritization
- Model dependence based prioritization
- **Heuristic #1 prioritization**

  In heuristic #1 prioritization main concept was that when execution of higher number of modified transactions is occur then it should have higher priority as compare to execution of smaller number of modified transaction. And higher number of modified transaction has also higher possibility to revealing faults [17].

- **Heuristic #2 prioritization**

  Heuristic #2 method of prioritization is the modified version of previous method i.e. heuristic #1. As we know that in heuristic #1 tests with higher priority to be selected first from the bucket then tests with lower priority to be selected. When tests are failed can be placed in lower priority bucket. So heuristic #2 methods tries to solve this issue and give more and more chance to lower priority tests to be selected [17].

- **Heuristic #3 prioritization**

  The concept of heuristic #3 method of prioritization was based on frequency. In this method tests with higher frequency should be selected first than tests with the lower frequency. The main idea of this method was that test that executes transaction more frequently has higher possibility to insert fault [17].

- **Model dependence based prioritization**

  Model dependence based prioritization is an approach of model based test case prioritization. In this approach tests with higher priority are always prioritized using model dependence analysis. In this approach new added and deleted transactions are communicate with model's outstanding part.

  In model –based test case prioritization, a system's model is used to prioritize tests. To model state-based system, system modeling is used. System models confine parts of the system behavior and several modeling languages have been developed, e.g., State Charts, EFSM [6].

b) Code Based Test Case Prioritization

Code based test case prioritization is a technique of test case prioritization in which source code is used to prioritize test cases. Most of the test case prioritization methods are code based. Code based test case prioritization techniques are dependent on information relating the tests of test suite of various elements of a system code of the original system [6].

c) Requirement Based Test Case Prioritization

Requirement based test case prioritization Requirement based test case prioritization developed a prioritization scheme with three main goals: identifying the severe faults earlier, to improve the software field quality and to devise the minimal set of PORT and PFs that can be used to effectively for the test case prioritization.

1.5 Effectiveness

In some cases the term effectiveness is also known as efficiency, affectivity and efficacy. Effectiveness or we can say that efficiency is the capability to produce desired result. It means that it has expected outcomes when something is effectiveness. Software test efficiency is number of test cases executed divided by unit of time. Test efficiency tests the amount of code and testing resource required by a program to perform a particular function. Effectiveness is the potential of producing a preferred result. When something is deem effective, it means it has a planned or predictable result, or produces a deep, bright idea.

CHAPTER 2
REVIEW OF LITRATURE

Sujata *et al* (2014) according to them egression testing is done on the test suite by applying one of its techniques that is test case prioritization which gives maximum number of faults and also provides effectiveness to the software. In this paper, problem was formulated to discover the maximum number of defects by prioritizing the test cases using model based dependencies. They said that in future test cases are prioritized with model based dependencies and efficiency of technique will be evaluated with APFD matrixes. In this research work, prioritization of test cases using their dependencies in model based testing will be performed. In order to detect the defects and provide information about these defects to the developer earlier, it will also reduce the time as the faults will be detected at early phases. Authors said that prioritization is discovering the" functional dependencies". Scenario that will be having more dependencies will produce more faults as well. Techniques can be: a). Open dependency structure which can be described as one test case should be executed prior to other test case anywhere in the program. b). Closed dependency structure which can be described as one test case should be executed just before the other test cases means the other test case must follow the first test case [8].

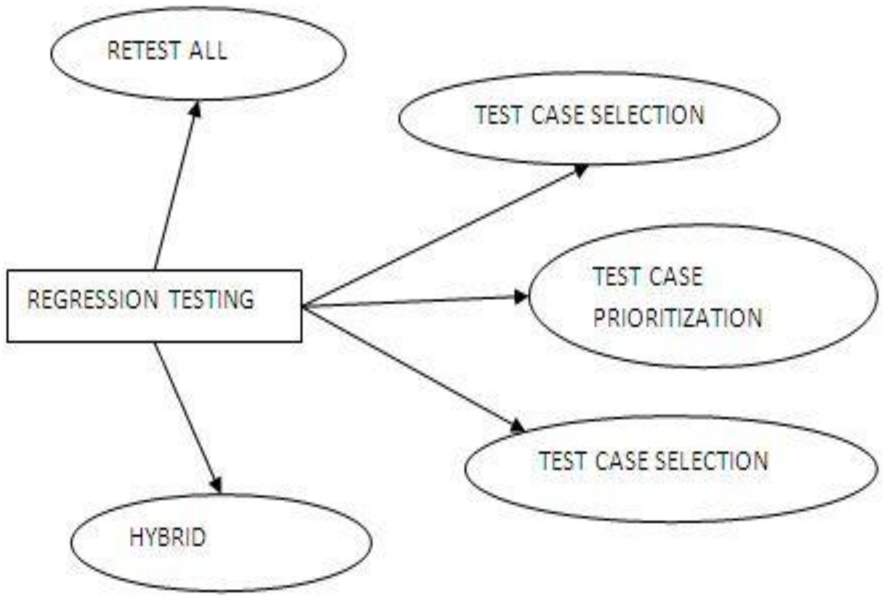


Figure 2.1: Test Case Prioritization Techniques [8]

Chandana Bharti *et al* (2013) conducted a study on different regression testing approaches. According to them there are various types of regression testing approaches like code based and model based. This study represented the analysis of both approaches of regression testing that are code based and model base. This analysis is based on some comparison and assessment standard. Also background of regression testing is presented in this study. Evaluation of code based technique in control dependence graph based test selection told that it is safe technique and support generality, it can be applied to all procedural languages. They also supported both intra-procedural and inter-procedural test selection. In model based technique if we want to generate test suite then we can use class and state diagram model of [3].

L.UmaRani *et al* (2013) according them regression testing is the verification that after a change earlier functioning software remains. During regression testing if we execute large number of test case then they are expensive and time consuming. To reduce it in regression testing TSP is an effective and practical approach . when test cases schedules in order of precedence then it increases their ability to meet some performance goals, such as code coverage, rate of fault detection. They concluded that prioritization of test case or test suits have different aspects of fault detection. On the basis of prioritization techniques, functionality of regression testing can improved in minimum time and recourses. Based on test case prioritization using regression testing they calculated the efficiency for each test cases . And according to them this can support to make a better software product. They said that test effectiveness and cost can be calculated for future work [2].

Prateeva Mahali *et al* (2013) present a model-based test case optimization and prioritization technique for regression testing. In this approach they investigate the present methods with respect to reduction of time and cost and also to slow case the efficiency of prioritized test cases. The result indicates that model based test case optimization and prioritization is better effective to solve all these problems at the same time. The work represented in this paper is comparatively small. In the future research, they plan to apply this approach on a relatively large application to review the scalability of the proposed approach [14].

Xiaoboo Han *et al* (2012) they presented an better model-based test prioritization approach, to prioritize the test suite for system retesting in this two types of information about the system model is used. They investigated the existing methods with respect to their effectiveness of early fault detection. Results indicated that algorithm used in this study has better effectiveness of early fault detection. Work presented in this paper was comparatively small. For future research, they told to do experimental studies on larger models and systems to have better understanding of the merits and disadvantages of our approach. They also planned to investigate additional model-based heuristic methods to consider both test case prioritization technique and regression test selection technique to reduce the cost of regression testing [7].

Sanjukta *et al* (2011) in their study the main focus is on various techniques which are based on codes, models and UML diagrams. They used two metrics (APFD and RPD) for determining the effectiveness of prioritization techniques. To understand the concepts of different code based techniques and behavior of components, interactions and also compatibility of components there was good coverage in terms of research. According to them in future we can include more numbers of criteria like during component interactions number of state changes by a test case, and also during the state changes, as whether it is going to access single attribute or multiple attributes from a database schema. They also told about optimization technique like GA to apply over their technique to represent a more effective prioritized test suite [1].

Kavitha V.R *et al* (2010) said that to increases the effectiveness for achieving few performance goals test case prioritization is beat approach and also for scheduling test cases in particular order. They told that the rate of fault detection is also another important goal. Test cases should run in an order because possibility of fault detection should be high means faults should be detected in earlier phases of life cycle model. Here author proposed an algorithm to improve the quality for user satisfaction for TCP from user requirements and also to improve the possibility of fault detection. Model prioritizes the test cases based on these factors: a). customer priority, b).changes in requirement, and c).implementation complexity. This prioritization technique represent the results which show two different sets of industrial projects and also tell about prioritization technique which improves the possibility of earlier fault detection [11].

Leila Naslavsley *et al* (2010) model-based selective approach of regression testing is used in this research and this approach uses relationships between model elements and test cases traversing. This approach adopted UML class and sequence diagrams as modeling perspective. Author analyzed this approach using the regression test evaluation structure, and described a prototype that implements the approach and a case study. A main statement of this method is that abstract test cases to be selected for regression testing and generated with this test generation approach [9].

Arup Abhinna Acharya *et al* (2010) To perform regression testing for real time systems this proposed technique can further be extended to prioritize test cases. In this the authors used model based technique to prioritize the test case. The authors are also working on adding new criterion like frequency of data base access number of state changes in UML state chart diagram etc. Two different modeling diagrams can also be combined to find criterion to generate test cases . Requirement specifications can be another way to prioritize the test cases. Test case prioritization for simultaneous systems is also a very challenging area of research due to its dynamic behavior [15].

Korel *et al* (2009) they simply presented small experimental comparison between model based and code based test case prioritization heuristics. With effectiveness of early fault detection of new modified system these heuristics are evaluated in this study. According to this study model based test case prioritization improved the detection of early faults as compared to code based test case prioritization. Author said that model-based test prioritization is an inexpensive approach as compared to existing code-based test prioritization approach because test suite is inexpensive for execution of the model. And model-based test case prioritization approach is a very sensitive approach to the information provided by the testers or developers whether this information is correct or incorrect. The study presented in this paper is limited to only two test prioritization heuristics. Author said that in future we can perform more extensive research on different methods of model based and code based test prioritization on larger models and systems. Because by this we can understand the limitations and advantages of model based and code based test case prioritization [6].

Praveen Ranjan Srivastava (2008) proposed an algorithm in order to improve regression testing for test case prioritization. With the help of APFD metric analysis is done for prioritized and non prioritized cases. Graphs prove that prioritized case is more effective. According to author in future we can try on test case prioritization over requirement analysis using risk metrics and APFD [13].

Leila Naslavsky (2007) presented the model based regression testing approach using traceability. Author said that to increased use of various models in conjunction modern driven development is leading in software testing with source code. According to author to support activities such as selective regression testing, model based regression testing traceability is required. Most model-based testing automated approaches always focus on the test case generation and execution activities and support to other activities is limited. So, deal with this problem author proposed a solution to create a traceable communications to test-related artifacts using model transformation and use this communications to support model-based selective regression testing approach. Author expected a small-size project which have smaller test suites by one small and one medium size project and it is clear that re-executing all test cases will be more cost effective. On the other hand, author expected that the medium size project should have more test cases and by this the cost selection and re-execution could be less [5].

Srikanth *et al* (2003) said that our results showed that unpredictability impact fault density and requirements complexity. They identified these factors as initial set of PF for PORT. To determine the effectiveness of customer's priority in their prioritization plan they planned to analyze the industrial data further. They believed that PORT can be improve the effectiveness of various testing activities, (1) for TCP to reduces the effort utilized as compared to coverage-based techniques and these techniques are prioritize based on the number of branches and statements covered, (2) the main focus is on functionalities which present the highest cost to the customer, and (3) to improves the rate of detection of severe faults. Earlier is believed to improve perceived software quality for rectifying severe faults [10].

Yanping Chen *et al* presented a specification –based selection approach of regression test with results from a small but real industrial study. And this used technique is object-oriented and risk based. This approach provided various methods to find both targeted and safety tests. For describing various requirements based on customer features and behavior basic model is used. They also used the activity diagram which is a notation of UML [4].

Eliane Collins *et al* (2012) presented study of various strategies for test automation in agile software development Scrum. On team organization and test automation they also observe the authority of agile values. It is feasible to solve problems in team and search for integrated testing tools and manage the management and organization of a software testing and also scrum process. The automation testing is a good resource to perform recurring tasks like document software, reduces cost of project tools, and also task allocation in small parts. To experiment new methods and tools the programmers' attitude is very important. When performing agile test automation strategy in alike environment some concepts learned can be extracted and they can be support other software engineers. According to author in future we can also work with other strategies of test automation for agile methodology [16].

CHAPTER 3
PRESENT WORK

In this chapter, a brief discussion about how the problem was formulated is there. Then, what were the various objectives that were setup during the proposal and what were achieved is presented. Last but not the least the methodology that we followed to work throughout. In section 3.1 a discussion about how the problem was formulated? Next section is 3.2 in which objectives are explained and in section 3.3 the methodology of the work done is explained. Research methodology tells about the flow of our work done with the help of flow chart is explained.

3.1 Problem Formulation

Regression testing is testing in which retesting of system, components and related units is done and it is ensured that modifications made in the system are working correctly and are not effecting other modules of system. Regression testing is testing in which modified system is tested using the old test suites. The procedure is to test modified parts of the system at first tested and then the whole system needs to be retested using old test suits to make sure that modification have not affected the other parts of the previously working system.

Developers are always interested to detect faults as early as possible in the system or software, due to the resource and time constraints. As in regression testing, re-execution of large test suites is there so these constraints are always taken into much consideration. There remains a need to prioritize test cases in order to reduce time and other resources use. So in this dissertation, work is on test case prioritization technique of regression testing and various metrics were be used to calculate the effectiveness of it. Some of these are APBC and APFD. APBC is an abbreviation of Average percentage of block coverage and it shows the code coverage and the second one i.e. APFD is an abbreviation of Average Percentage of Faults Detected and it shows the percentage value of faults detected when the test suite is run to find out the faults. These two would be discussed in more detail in chapter 4.

3.2 Objectives

Regression testing is expensive and time consuming due to a large number of test cases executions. To reduce this expense some techniques can be used which can reduce the time and cost and hence increase efficiency. Researchers are doing work in this field and as a result many new techniques have been developed. One such technique is test case prioritization technique. Through test case prioritization technique, test cases are prioritized in some order of execution according to some criteria so that less number of test case executions can provide better results. But there always remains a risk of not testing some part of system because of the reduction in number of test case executions in order to reduce the expense. So for every such technique there are some metrics on the basis of which their effectiveness can be measured. If effectiveness can be measured then their always remains a scope of improving that effectiveness by some means So here are objectives of this research proposal that were made clear by studying literature review.

The main objectives of the research are:

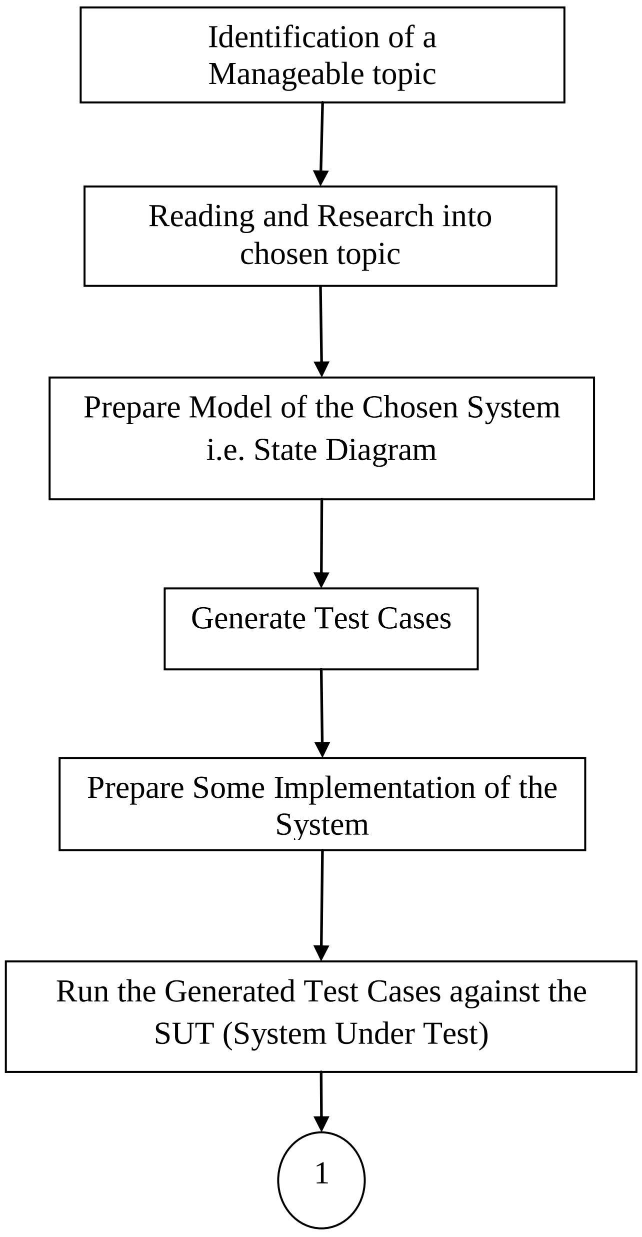
a) To improve the effectiveness of test case prioritization technique for regression testing. As is clear from the literature review that APFD percentage value can be improved more.
b) To analyze the code coverage provided by model based test cases and calculate the value of APBC (Average Percentage of Block Coverage).
c) To prioritize model based test cases by analyzing the fault matrix and a new parameter time and then evaluate the efficiency of test case prioritization technique with APFD metrics.

By improving the effectiveness of test case prioritization technique, we can reduce the cost and time used during the regression testing. And second objective of study is to prioritize test cases with model based dependencies which can give better results as compared to existing heuristic methods. In software development life cycle testing is major and important part. All it is to reduce the use of resources so that overall cost of the system can be reduced.

3.3 Methodology

The step wise step procedure that was followed in order to fulfill the mentioned objectives is as follows. From the very beginning the first step was to identify an original and Manageable topic. The topic chosen was UNDO system because it is not a very big system and it also fulfills the purpose of the study. Then some study was made about the system i.e. how it works and how it could be implemented. The next was to prepare the model of the system.

Different models can represent the behavior of the system and there are different tools available to represent the models. So it was a difficult task but State chart diagrams were chosen for the purpose of this study. Why State machine diagrams is explained in next chapter. Then model of the system was prepared in spec explorer. More about spec explorer is in the chapter 4. Then test cases were generated automatically by the tool. To find out the further findings some mockup of the SUT was prepared and was attached with the test cases to test the system. To see how much code of the implementation i.e. mockup is covered by the automatically generated test cases, analysis about code coverage was done. To be more specific APBC was calculated and analyzed. More on this is in next chapter. Some faults were associated with the system and test cases in different were run against the faults to check whether the faults were found or not.


Identification of a Manageable topic
Reading and Research into chosen topic
Prepare Model of the Chosen System i.e. State Diagram
Generate Test Cases
Prepare Some Implementation of the System
Run the Generated Test Cases against the SUT (System Under Test)
1

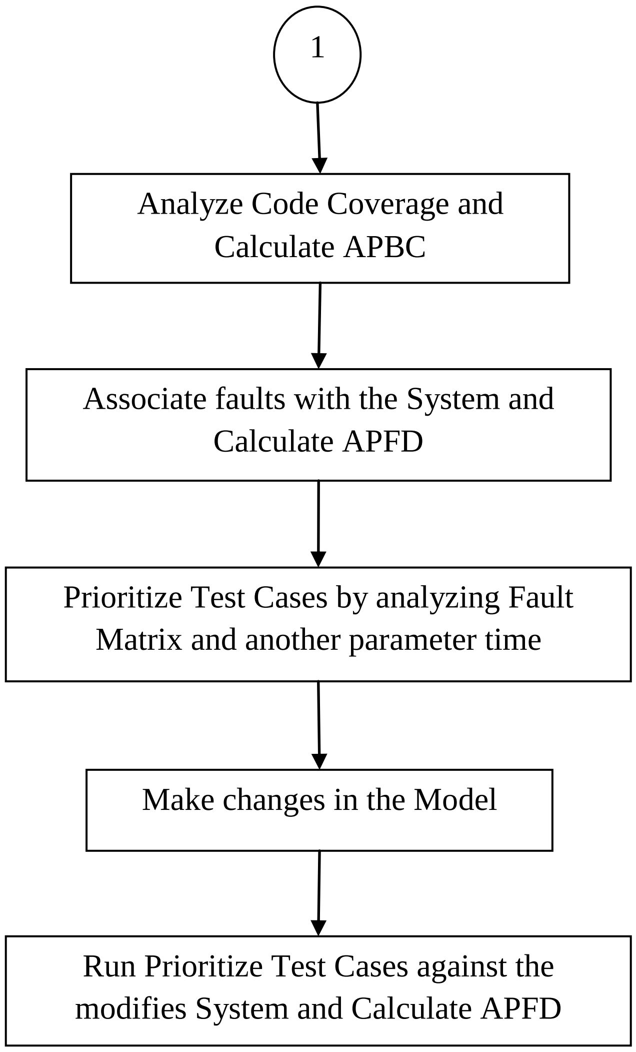

1
Analyze Code Coverage and Calculate APBC
Associate faults with the System and Calculate APFD
Prioritize Test Cases by analyzing Fault Matrix and another parameter time
Make changes in the Model
Run Prioritize Test Cases against the modifies System and Calculate APFD

Figure 3.1: Research Methodology

Then prioritization of test cases was done against the arbitrary order of execution and results were analyzed. More on how all is done is in next chapter.

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Identification of Manageable System

In the software industry there can be simple or most probably there are complex systems. These systems are built by following a process and modeling a system to be developed is an important part of that process. That process is under continuous research so that overall development time can be reduced. This research work is related to that and for the purpose of this research a simple system has been chosen i.e. UNDO system.

A Stack can be used to perform many operations where there is need to pop out the element which was inserted in it at last. One such implementation of the stack can be done to perform the UNDO operation. Now what is UNDO operation?

As is clear from the name that an UNDO operation is where the last operation done is undone first and then the second last operation done is undone and then the third last operation is undone and so on. This operation can be performed with the help of stack. A diagrammatic representation of this system is below:

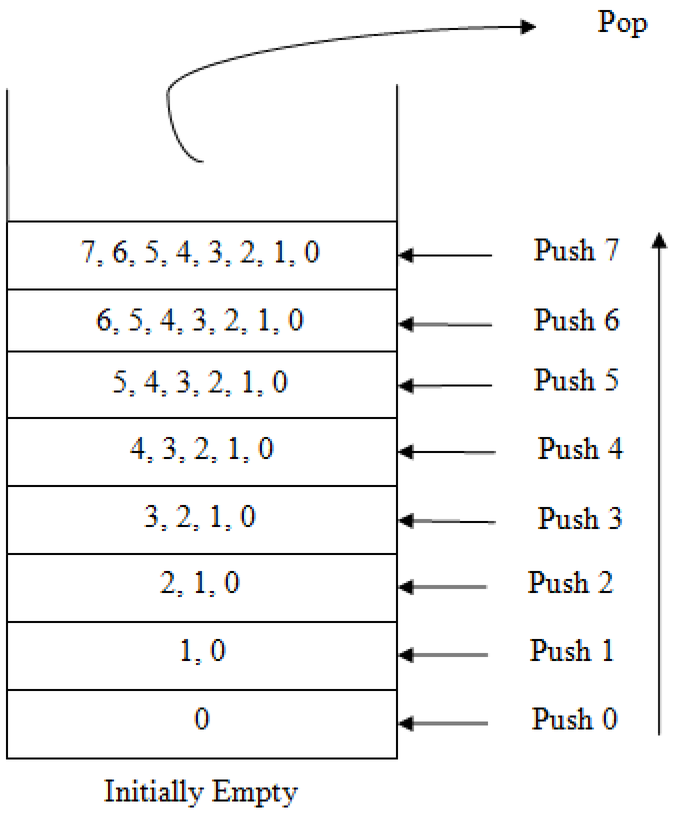


Figure 4.1: Operation of stack

The figure 4.1 shows the operation of stack. As is clear from diagram that initially stack is empty, then seven elements were inserted in the stack in the order 0, 1, 2, 3, 4, 5, 6, 7. When 0 was inserted in the stack, there was one element in the stack i.e. 0. When 1 was inserted in the stack, the stack elements were [1, 0] in the same order. When the element 2 was inserted in the stack, the stack elements were [2, 1, 0] in the same order. In the same order mean that for [2, 1, 0], 2 is at the top. Stack performs two operations i.e. Push and Pop. Push is used to insert an element at top in the stack and Pop is used to pop out the top element from the stack. We mean to say that both the operations are performed on the same end and we say that end Top. This is why UNDO system can be implemented with the help of stack in which the last operation is undone first and then the second last operation is undone second and so on.

4.2 Prepare Model of the chosen System

Here the concern is about Model based testing, where the test cases are generated from the models of the system. There are some tools available which generate abstract test cases and there are others which generate working test cases from the model. The concern here is to automate the testing completely not partially. So Spec Explore which is a model based test case generation tool is used here. Now there are many behavioral representations of the system. A state transition diagram represents the behavior of a system in terms of various states in which a system can be and the transitions that force a system to change from one state to another or to remain in the same state. This can be explained with the help of diagram below:

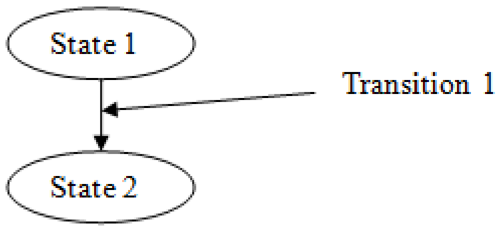


Figure4.2: States and Transitions

4.2.1 Behavioral Diagram

There are nine diagrams used to capture different views of a system in UML. These diagrams can be developed to get the real implementation of the system. Behavioral diagram is one of those diagrams. In behavioral view there are:

- Sequence Diagram
- Collaboration Diagram

- State-chart Diagram
- Activity Diagram

4.2.1.1 State- Chart Diagram

UML state chart which is also known by the name of state machine. In the field of computer science these state machines are the improved understanding of the mathematical concept of finite automata. State machine just like state chart diagram is used to represent the behavior of any system to be developed in the field of computer science. These can represent the behavior of a single object of the overall behavior of the system. The states in state machine diagram are represented by circles and to move from one state to another arrows are used.

4.2.1.2 FSM (Finite State Machine)

FSM have been generally used to model systems in various areas, with Sequential circuits, some types of programs for example lexical analysis, pattern matching and more newly. The requirement of system reliability motivates study into the problem of testing FSM to make sure their accurate functioning and to find out aspects of their performance. A FSM consist a finite number of states and produces outputs on state transitions after getting inputs. In a testing problem we have a machine about which we need some information; we would like to assume this information by provided that a sequence of inputs to the machine and the outputs produced. Because of its practical meaning and theoretical attention, the problem of testing FSM has been studied in different areas and at a range of times.

4.2.2 Representation of State diagram as FSM (Finite State Machine)

For this study, State Chart Diagram of the SUT is used to represent the behavior of the system. And now it is clear that State Chart Diagrams can be represented as Finite State Machines. So a Finite state Machine of the system has been developed which is shown on the figure 4.3. This FSM is representing the behavior of the UNDO system as Stack. As S0 is the initial state i.e. stack is empty. The Checker[] reads a particular state and gives the elements in the stack. Then an action is performed on this state i.e. Push[0]. This operation changes the state of the stack from S0 to S3. S3 is again read by the action Checker[] and it gives the result as Checker[0]. Pop transition will take the system to previous state. As it can be observed in figure 4.3, if pop() is performed on state “S8”, it will take the system to the state “S3”. There is one other operation known as clear(). If this operation performed on any state of the system, the system will return to its initial state.

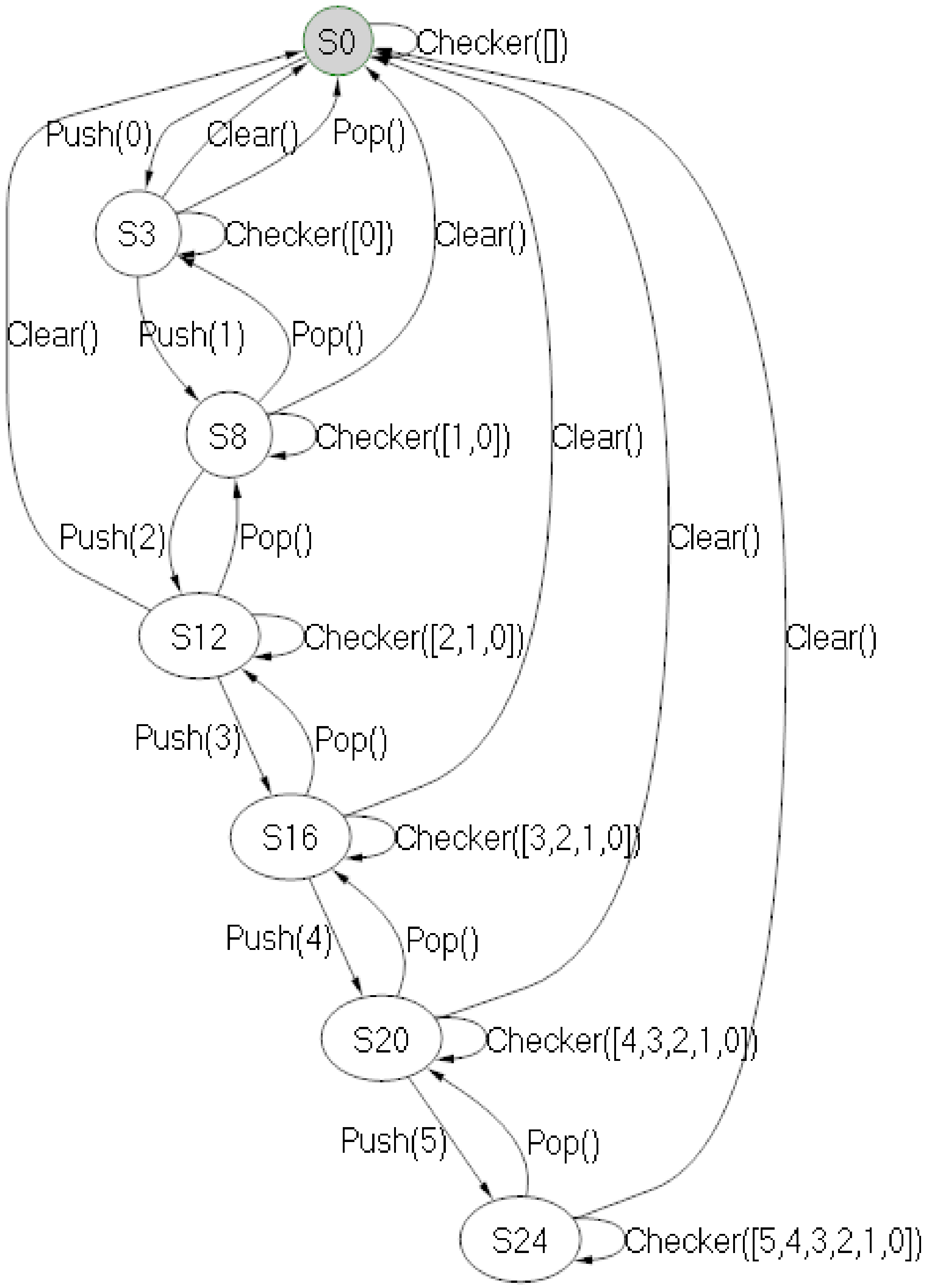


Figure 4.3: Finite State Machine of Undo System

4.2.2.1 Tool Used

Now a day there is a vast category of tools available which are used for model based testing. A study of various tools was made for the purpose of this study. Some of the tools that could be used for this study were some java tools like rational rose, magic draw, model junit, junit, eclipse. Model junit is an extension of junit and it represents the behavior of system in the form of FSM or EFSM. The test cases generated by the model junit are abstract test cases. One should provide the implementation of the methods generated. But to automate the whole process spec explorer as an extension of visual studio was used. Now let’s have a brief about visual studio and spec explorer.

Visual Studio

Visual Studio is developed by Microsoft. It is an integrated development environment (IDE). It is used to build up computer programs for Microsoft Windows, web sites, web applications and web services. Visual Studio uses various Microsoft software development platforms such as Windows Presentation Foundation, Windows API, Windows Store, Windows Forms, and Microsoft Silver light. It can create both local code and managed code.

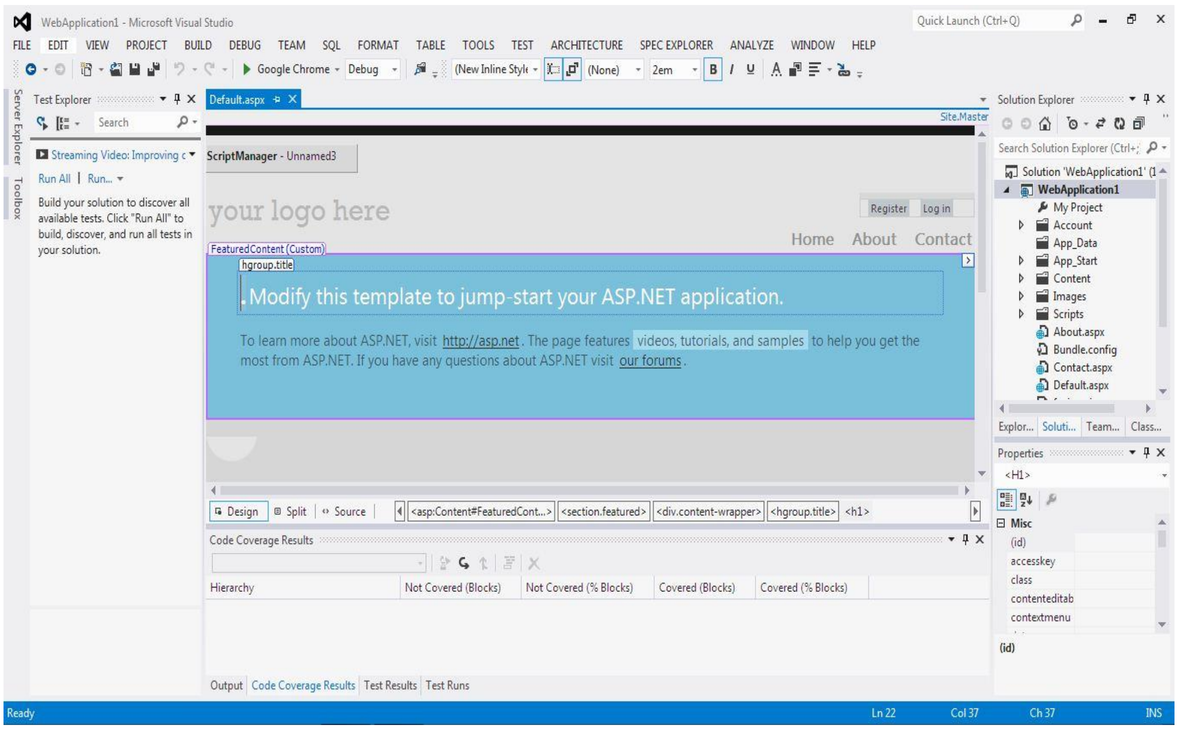
VS include a code editor sustaining IntelliSense and code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Visual Studio wires different programming languages and allows the code editor and debugger to carry any programming language.

Visual Studio has various features like:

- Debugger
- Code editor
- Designer

Spec Explorer

Spec explorer is an extension of visual studio. The main idea behind spec explorer is to train a system's proposed behavior in the form of model program. The model program normally does much less than the execution; it does just sufficient to capture the applicable states of the system and demonstrate the constraints that a correct execution must follow. The objective is to identify from a selected viewpoint what the system must do, what it may do and what it must not do. Explore the probable runs of the requirement-program as a way to thoroughly generate test suites.


WebApplication1 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL FORMAT TABLE TOOLS TEST ARCHITECTURE SPEC EXPLORER ANALYZE WINDOW HELP
Test Explorer
Default.aspx
Solution Explorer
Properties
Code Coverage Results
Output Code Coverage Results Test Results Test Runs
Ready Ln 22 Col 37 Ch 37 INS

Streaming Video: Improving c

Build your solution to discover all available tests. Click "Run All" to build, discover, and run all tests in your solution.

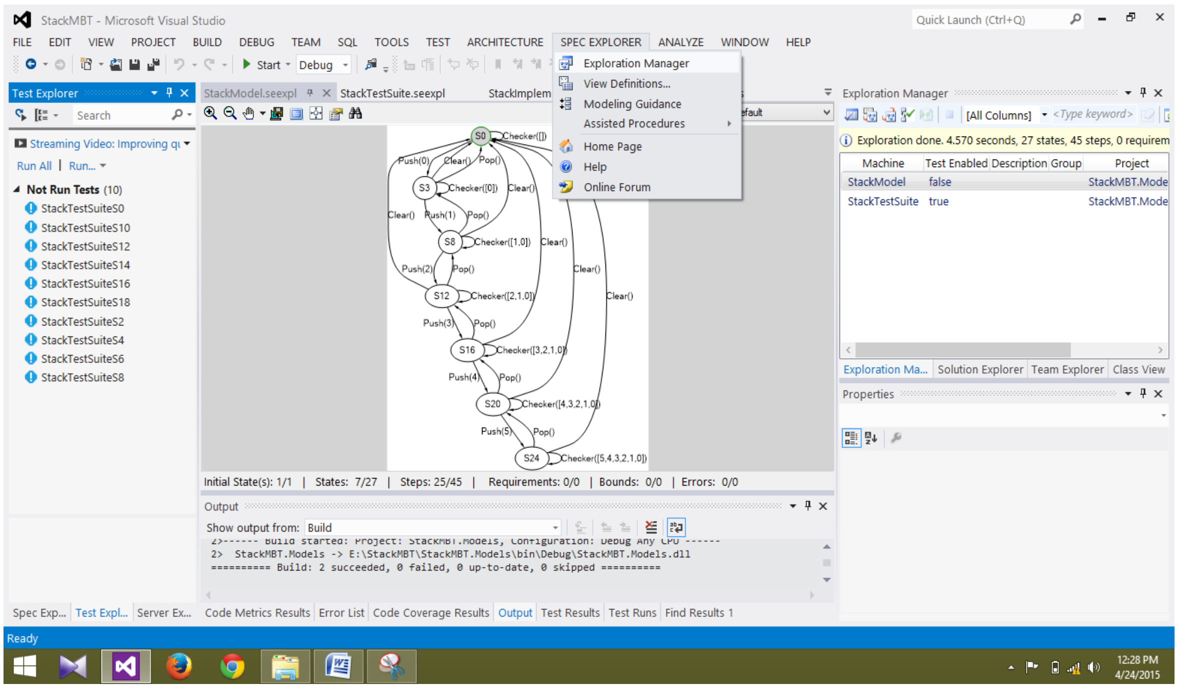
your logo here

Modify this template to jump-start your ASP.NET application.

To learn more about ASP.NET, visit http://asp.net. The page features videos, tutorials, and samples to help you get the most from ASP.NET. If you have any questions about ASP.NET visit our forums.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)

Figure 4.4: Visual Studio framework


StackMBT - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE SPEC EXPLORER ANALYZE WINDOW HELP
Exploration Manager
View Definitions...
Modeling Guidance
Assisted Procedures
Home Page
Help
Online Forum
Test Explorer
StackModel.seexpl
StackTestSuite.seexpl
S0 S3 S8 S12 S16 S20 S24
Push(0) Push(1) Push(2) Push(3) Push(4) Push(5) Pop() Clear()
Checker([]) Checker([0]) Checker([1,0]) Checker([2,1,0]) Checker([3,2,1,0]) Checker([4,3,2,1,0]) Checker([5,4,3,2,1,0])
Spec Exp... Test Expl... Server Ex... Code Metrics Results Error List Code Coverage Results Output Test Results Test Runs Find Results 1
Ready
12:28 PM 4/24/2015

Not Run Tests (10)
StackTestSuiteS0
StackTestSuiteS10
StackTestSuiteS12
StackTestSuiteS14
StackTestSuiteS16
StackTestSuiteS18
StackTestSuiteS2
StackTestSuiteS4
StackTestSuiteS6
StackTestSuiteS8

Exploration done. 4.570 seconds, 27 states, 45 steps, 0 requirem

Machine	Test Enabled	Description	Group	Project
StackModel	false			StackMBT.Mode
StackTestSuite	true			StackMBT.Mode

Initial State(s): 1/1 | States: 7/27 | Steps: 25/45 | Requirements: 0/0 | Bounds: 0/0 | Errors: 0/0

Show output from: Build

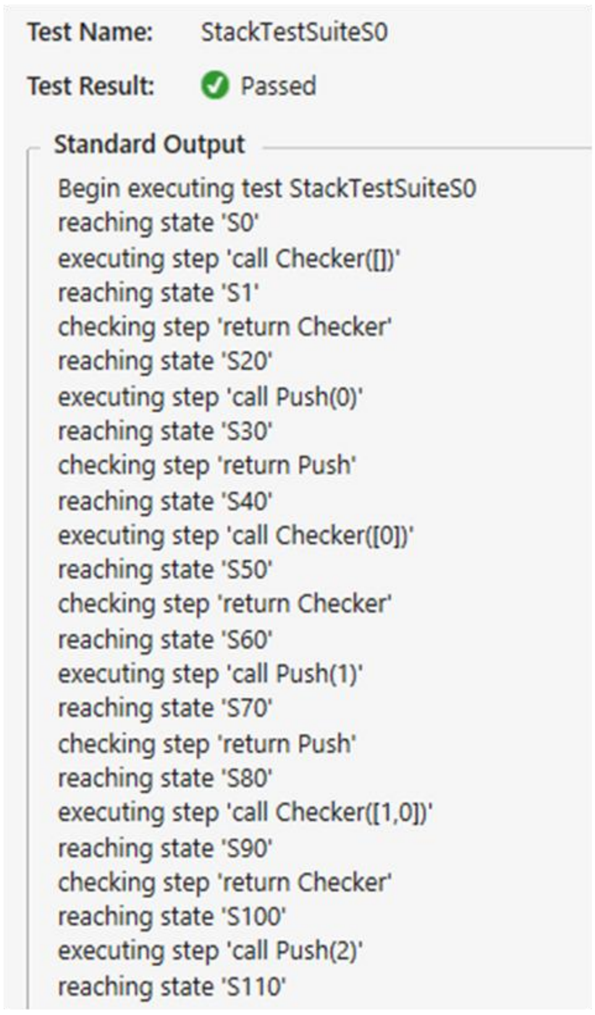
```
2>------ Build started: Project: StackMBT.Models, Configuration: Debug Any CPU ------
2>  StackMBT.Models -> E:\StackMBT\StackMBT.Models\bin\Debug\StackMBT.Models.dll
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 4.5: Spec Explorer

4.3 Generate Test Cases from the FSM

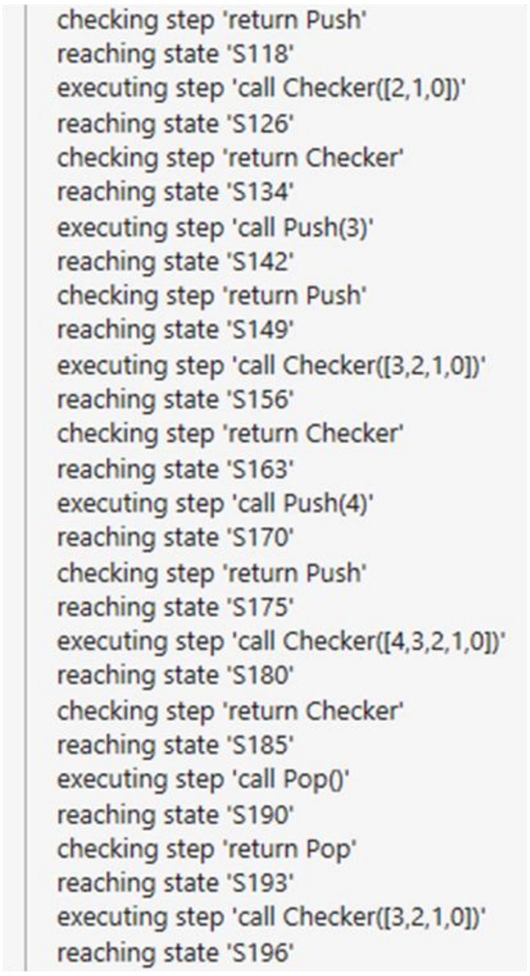
There were ten test cases generated from the FSM model of the system. These test cases were test to pass test cases.

4.3.1 Various States and Transitions Covered by the Generated Test Cases

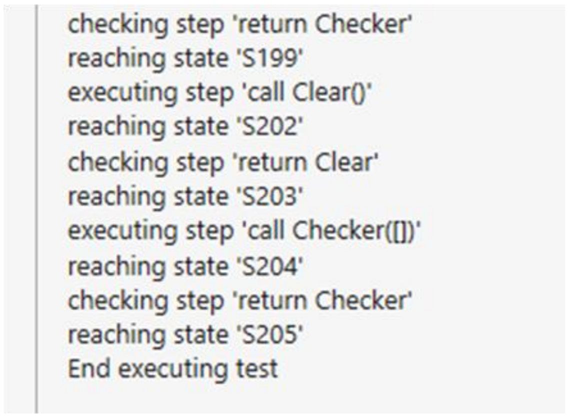


```
Test Name:    StackTestSuiteS0
Test Result:  Passed

Standard Output
Begin executing test StackTestSuiteS0
reaching state 'S0'
executing step 'call Checker([])'
reaching state 'S1'
checking step 'return Checker'
reaching state 'S20'
executing step 'call Push(0)'
reaching state 'S30'
checking step 'return Push'
reaching state 'S40'
executing step 'call Checker([0])'
reaching state 'S50'
checking step 'return Checker'
reaching state 'S60'
executing step 'call Push(1)'
reaching state 'S70'
checking step 'return Push'
reaching state 'S80'
executing step 'call Checker([1,0])'
reaching state 'S90'
checking step 'return Checker'
reaching state 'S100'
executing step 'call Push(2)'
reaching state 'S110'
```



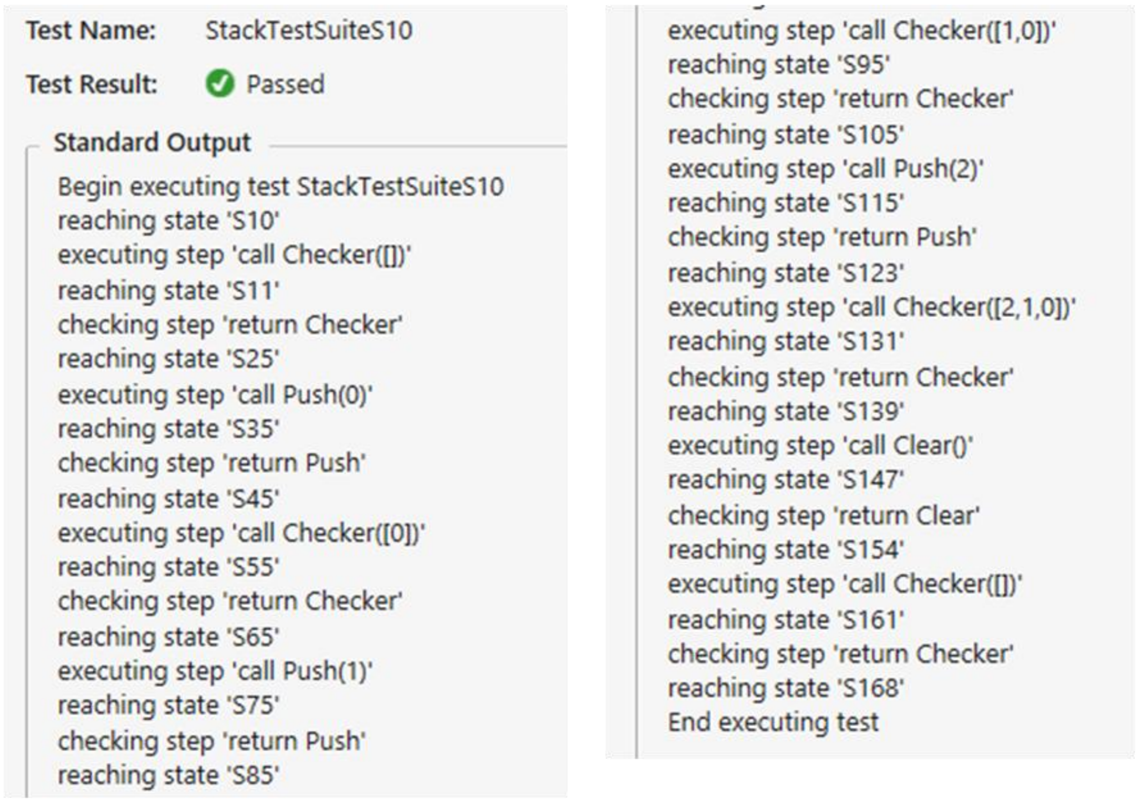
```
checking step 'return Push'
reaching state 'S118'
executing step 'call Checker([2,1,0])'
reaching state 'S126'
checking step 'return Checker'
reaching state 'S134'
executing step 'call Push(3)'
reaching state 'S142'
checking step 'return Push'
reaching state 'S149'
executing step 'call Checker([3,2,1,0])'
reaching state 'S156'
checking step 'return Checker'
reaching state 'S163'
executing step 'call Push(4)'
reaching state 'S170'
checking step 'return Push'
reaching state 'S175'
executing step 'call Checker([4,3,2,1,0])'
reaching state 'S180'
checking step 'return Checker'
reaching state 'S185'
executing step 'call Pop()'
reaching state 'S190'
checking step 'return Pop'
reaching state 'S193'
executing step 'call Checker([3,2,1,0])'
reaching state 'S196'
```



```
checking step 'return Checker'
reaching state 'S199'
executing step 'call Clear()'
reaching state 'S202'
checking step 'return Clear'
reaching state 'S203'
executing step 'call Checker([])'
reaching state 'S204'
checking step 'return Checker'
reaching state 'S205'
End executing test
```

Figure 4.6: Standard output of test case StackTestSuiteS0

Figure 4.6 shows the standard output generated by the test case named "StackTestSuitS0". It can be seen from this figure that the test case is beginning its execution by reaching at state “S0”. Then an events triggers which is a call to a function “checker([])”. Now after this function is performed, there occurs a transition which takes the system from state “S0” to “S1”. Next is what is returned? i.e. checker which will just read the elements in the stack. Now the System reaches the state “S20”. Then a “push(0)” operation would be performed which will take the system from state “S20” to “S30”. Next the Push would be returned and the system will transit from state “S30” to “S40”. In this way system will transit from one state to another depicting the behavior of the system. These state names are not in the sequence because these were generated automatically by the tool.



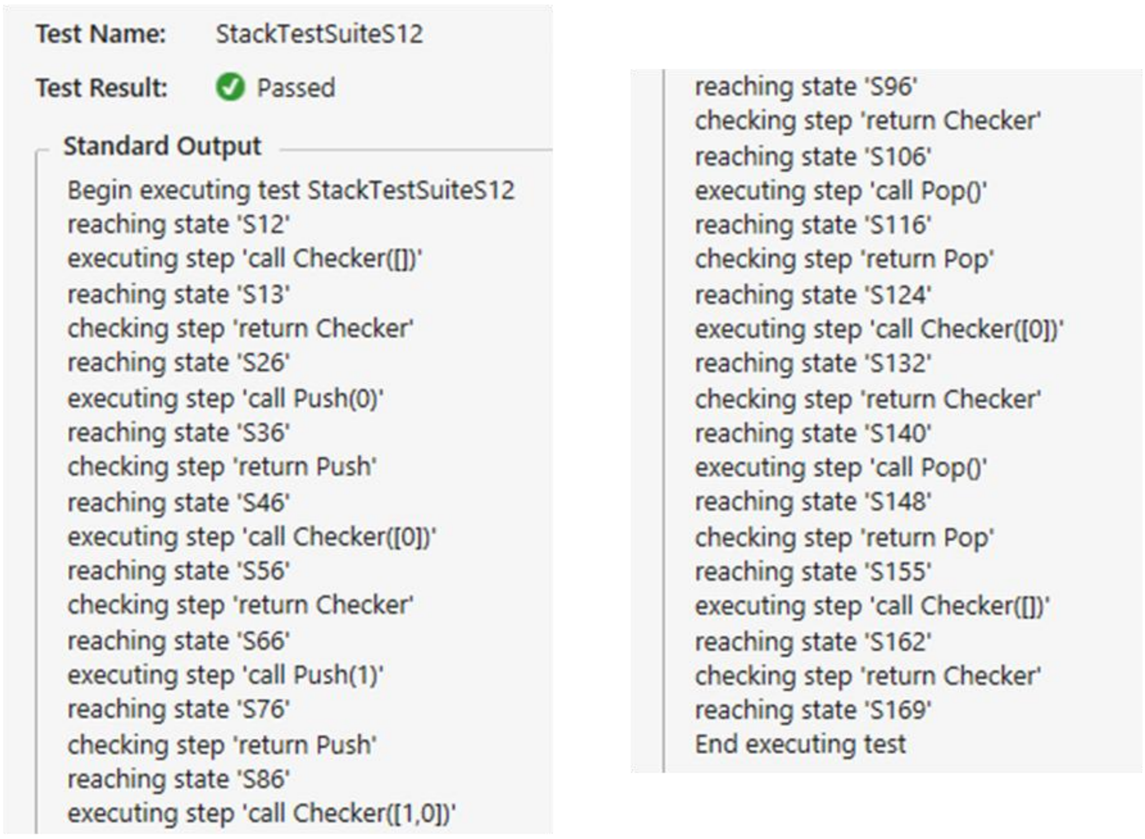
```
Test Name:     StackTestSuiteS10
Test Result:   Passed

Standard Output
Begin executing test StackTestSuiteS10
reaching state 'S10'
executing step 'call Checker([])'
reaching state 'S11'
checking step 'return Checker'
reaching state 'S25'
executing step 'call Push(0)'
reaching state 'S35'
checking step 'return Push'
reaching state 'S45'
executing step 'call Checker([0])'
reaching state 'S55'
checking step 'return Checker'
reaching state 'S65'
executing step 'call Push(1)'
reaching state 'S75'
checking step 'return Push'
reaching state 'S85'
```

```
executing step 'call Checker([1,0])'
reaching state 'S95'
checking step 'return Checker'
reaching state 'S105'
executing step 'call Push(2)'
reaching state 'S115'
checking step 'return Push'
reaching state 'S123'
executing step 'call Checker([2,1,0])'
reaching state 'S131'
checking step 'return Checker'
reaching state 'S139'
executing step 'call Clear()'
reaching state 'S147'
checking step 'return Clear'
reaching state 'S154'
executing step 'call Checker([])'
reaching state 'S161'
checking step 'return Checker'
reaching state 'S168'
End executing test
```

Figure 4.7: Standard output of test cases StackTestSuitesS10

Figure 4.7 shows the various states and transitions covered by the test case named “StackTestSuitS10”. This test case is starting its execution from the state “S10” then reaching “S11” by the transition “call Checker([])”. It transits from state “S11” to “S25” through “return checker” and so on. By starting its execution from state “S10”, this test case ends its execution by reaching the state “S168”.



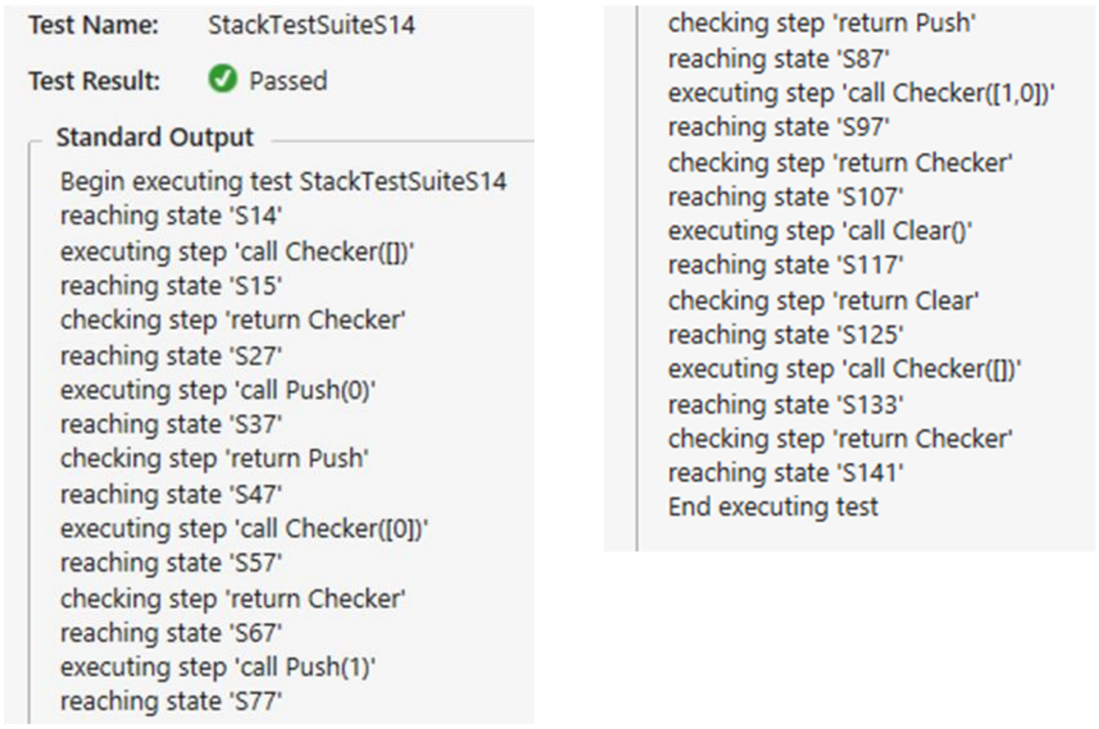
```
Test Name:    StackTestSuiteS12
Test Result:  Passed

Standard Output
Begin executing test StackTestSuiteS12
reaching state 'S12'
executing step 'call Checker([])'
reaching state 'S13'
checking step 'return Checker'
reaching state 'S26'
executing step 'call Push(0)'
reaching state 'S36'
checking step 'return Push'
reaching state 'S46'
executing step 'call Checker([0])'
reaching state 'S56'
checking step 'return Checker'
reaching state 'S66'
executing step 'call Push(1)'
reaching state 'S76'
checking step 'return Push'
reaching state 'S86'
executing step 'call Checker([1,0])'
```

```
reaching state 'S96'
checking step 'return Checker'
reaching state 'S106'
executing step 'call Pop()'
reaching state 'S116'
checking step 'return Pop'
reaching state 'S124'
executing step 'call Checker([0])'
reaching state 'S132'
checking step 'return Checker'
reaching state 'S140'
executing step 'call Pop()'
reaching state 'S148'
checking step 'return Pop'
reaching state 'S155'
executing step 'call Checker([])'
reaching state 'S162'
checking step 'return Checker'
reaching state 'S169'
End executing test
```

Figure 4.8: Standard output of test cases StacktestsuitesS12

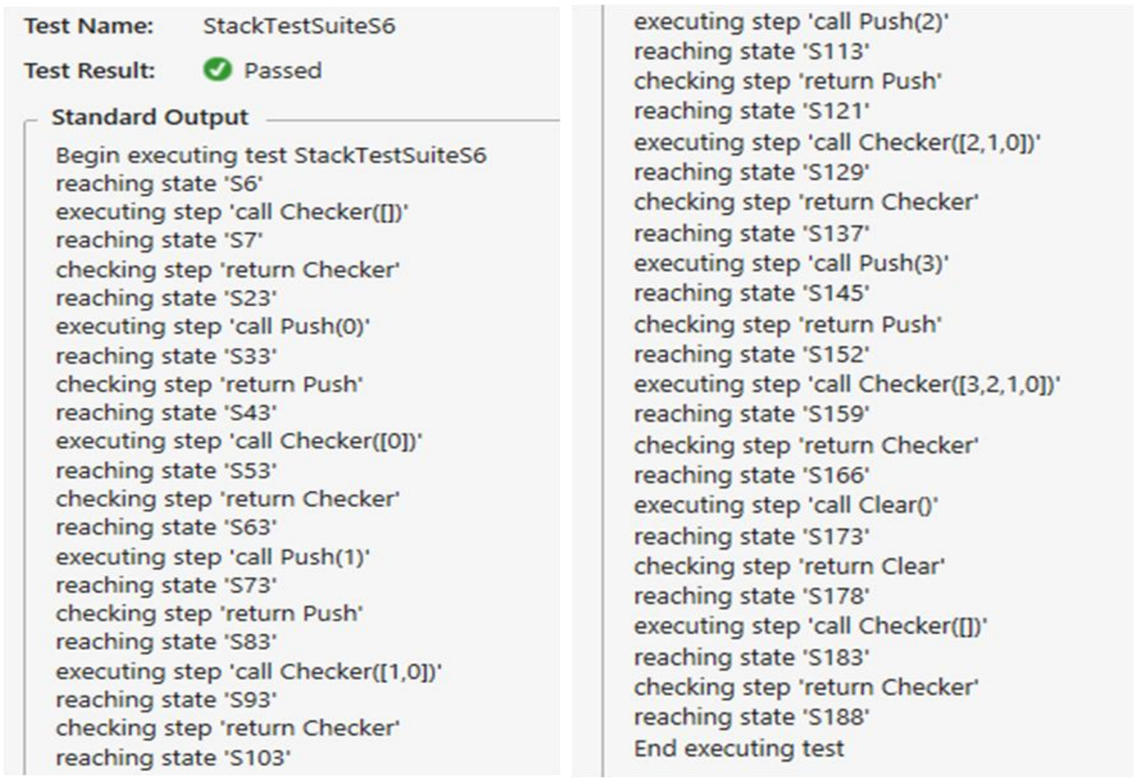
Figure 4.8 shows the execution of test case named “StackTestSuiteS12”. As is clear from the figure, this test case starts its execution from the state “S12” and ends its execution by reaching at the state “S169”. From state “S12” system moves to state “S13” through transition “call Checker([])”. By looking at some arbitrary state, the behavior of other function can be analyzed. E.g.: let us start at state “S66”. When the system was at state “S66”, a “call Push(1)” moves the system from this state to “S76” and when the “checker()” will perform, it will return “Checker([1,0])” i.e. the elements in the stack are 1 and 0 and the stack is at level 1 as per our assumption. In this way the system goes through various states and transitions and various paths are observed by the automatically generated test cases.


```
Test Name:    StackTestSuiteS14
Test Result:  Passed

Standard Output
Begin executing test StackTestSuiteS14
reaching state 'S14'
executing step 'call Checker([])'
reaching state 'S15'
checking step 'return Checker'
reaching state 'S27'
executing step 'call Push(0)'
reaching state 'S37'
checking step 'return Push'
reaching state 'S47'
executing step 'call Checker([0])'
reaching state 'S57'
checking step 'return Checker'
reaching state 'S67'
executing step 'call Push(1)'
reaching state 'S77'
checking step 'return Push'
reaching state 'S87'
executing step 'call Checker([1,0])'
reaching state 'S97'
checking step 'return Checker'
reaching state 'S107'
executing step 'call Clear()'
reaching state 'S117'
checking step 'return Clear'
reaching state 'S125'
executing step 'call Checker([])'
reaching state 'S133'
checking step 'return Checker'
reaching state 'S141'
End executing test
```

Figure 4.9: Standard output of test cases StackTestSuiteS14

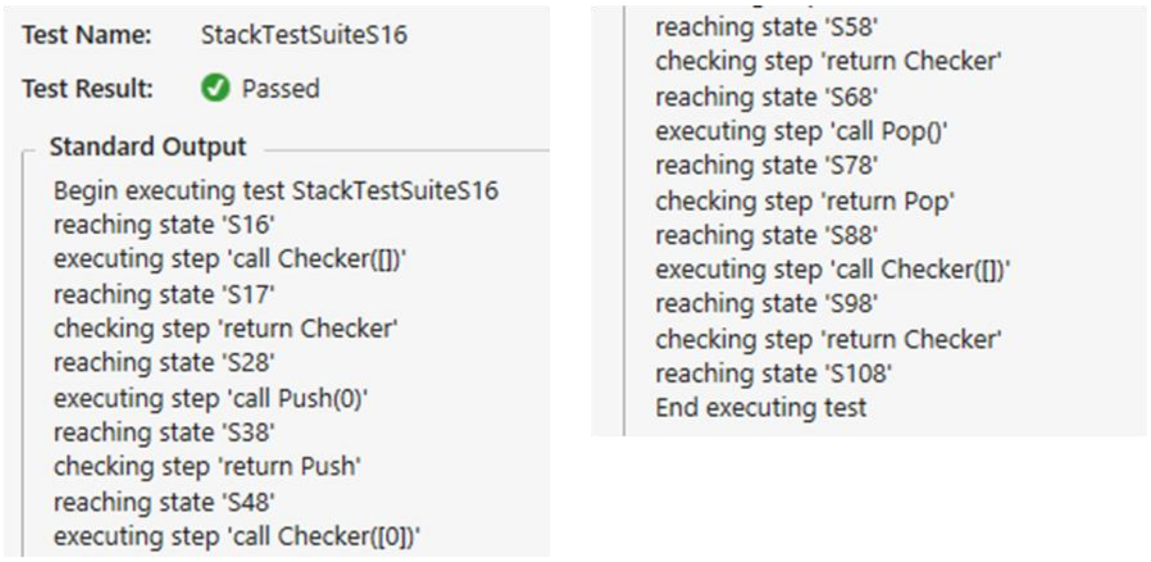
Figure 4.9 shows the various states and transitions covered by the test case named "StackTestSuiteS14". The execution starts at state "S14" and ends at the state "S141".


```
Test Name:    StackTestSuiteS6
Test Result:  Passed

Standard Output
Begin executing test StackTestSuiteS6
reaching state 'S6'
executing step 'call Checker([])'
reaching state 'S7'
checking step 'return Checker'
reaching state 'S23'
executing step 'call Push(0)'
reaching state 'S33'
checking step 'return Push'
reaching state 'S43'
executing step 'call Checker([0])'
reaching state 'S53'
checking step 'return Checker'
reaching state 'S63'
executing step 'call Push(1)'
reaching state 'S73'
checking step 'return Push'
reaching state 'S83'
executing step 'call Checker([1,0])'
reaching state 'S93'
checking step 'return Checker'
reaching state 'S103'
executing step 'call Push(2)'
reaching state 'S113'
checking step 'return Push'
reaching state 'S121'
executing step 'call Checker([2,1,0])'
reaching state 'S129'
checking step 'return Checker'
reaching state 'S137'
executing step 'call Push(3)'
reaching state 'S145'
checking step 'return Push'
reaching state 'S152'
executing step 'call Checker([3,2,1,0])'
reaching state 'S159'
checking step 'return Checker'
reaching state 'S166'
executing step 'call Clear()'
reaching state 'S173'
checking step 'return Clear'
reaching state 'S178'
executing step 'call Checker([])'
reaching state 'S183'
checking step 'return Checker'
reaching state 'S188'
End executing test
```

Figure 4.10: Standard output of test cases StackTestSuiteS6

Figure 4.10 shows the standard output generated by the test case named "StackTestSuiteS6". The test case starts its execution from the state "S6" and ends its execution by reaching at the state "S188".

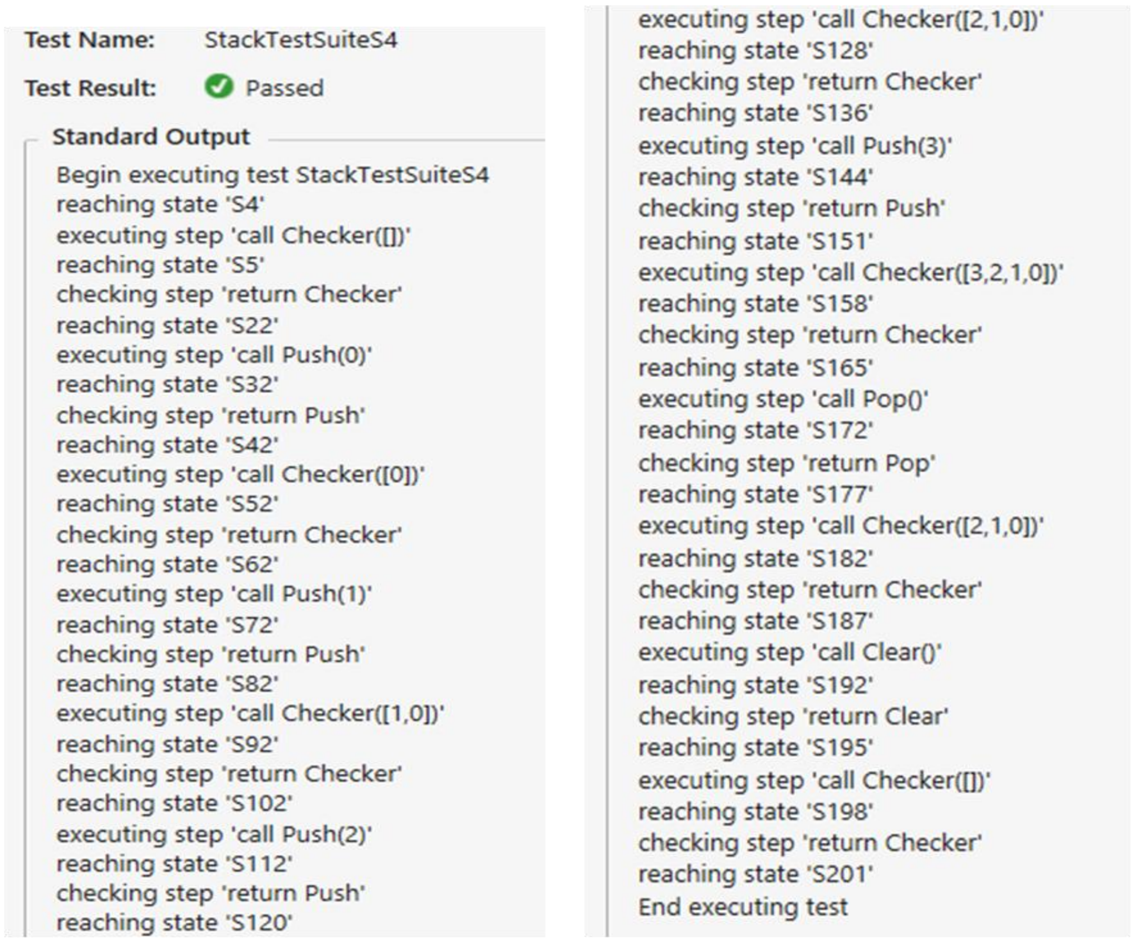


```
Test Name:    StackTestSuiteS16
Test Result:  ✓ Passed

Standard Output
Begin executing test StackTestSuiteS16
reaching state 'S16'
executing step 'call Checker([])'
reaching state 'S17'
checking step 'return Checker'
reaching state 'S28'
executing step 'call Push(0)'
reaching state 'S38'
checking step 'return Push'
reaching state 'S48'
executing step 'call Checker([0])'
reaching state 'S58'
checking step 'return Checker'
reaching state 'S68'
executing step 'call Pop()'
reaching state 'S78'
checking step 'return Pop'
reaching state 'S88'
executing step 'call Checker([])'
reaching state 'S98'
checking step 'return Checker'
reaching state 'S108'
End executing test
```

Figure 4.11: Standard output of test cases StacktestsuitesS16

Figure 4.11 shows the standard output generated by the test case named "StackTestSuiteS16". The test case starts its execution from the state "S16" and ends its execution by reaching at the state "S108".

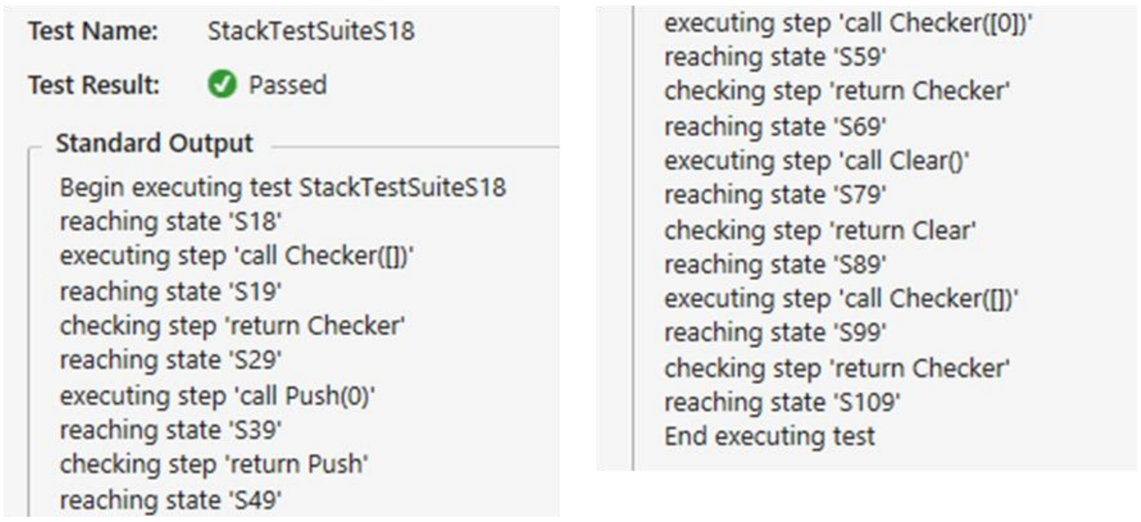


```
Test Name:     StackTestSuiteS4
Test Result:   ✓ Passed

Standard Output
Begin executing test StackTestSuiteS4
reaching state 'S4'
executing step 'call Checker([])'
reaching state 'S5'
checking step 'return Checker'
reaching state 'S22'
executing step 'call Push(0)'
reaching state 'S32'
checking step 'return Push'
reaching state 'S42'
executing step 'call Checker([0])'
reaching state 'S52'
checking step 'return Checker'
reaching state 'S62'
executing step 'call Push(1)'
reaching state 'S72'
checking step 'return Push'
reaching state 'S82'
executing step 'call Checker([1,0])'
reaching state 'S92'
checking step 'return Checker'
reaching state 'S102'
executing step 'call Push(2)'
reaching state 'S112'
checking step 'return Push'
reaching state 'S120'
executing step 'call Checker([2,1,0])'
reaching state 'S128'
checking step 'return Checker'
reaching state 'S136'
executing step 'call Push(3)'
reaching state 'S144'
checking step 'return Push'
reaching state 'S151'
executing step 'call Checker([3,2,1,0])'
reaching state 'S158'
checking step 'return Checker'
reaching state 'S165'
executing step 'call Pop()'
reaching state 'S172'
checking step 'return Pop'
reaching state 'S177'
executing step 'call Checker([2,1,0])'
reaching state 'S182'
checking step 'return Checker'
reaching state 'S187'
executing step 'call Clear()'
reaching state 'S192'
checking step 'return Clear'
reaching state 'S195'
executing step 'call Checker([])'
reaching state 'S198'
checking step 'return Checker'
reaching state 'S201'
End executing test
```

Figure 4.12: Standard output of test cases StacktestsuitesS4

Figure 4.12 shows the standard output generated by the test case named "StackTestSuiteS4". The test case starts its execution from the state "S4" and ends its execution by reaching at the state "S201".

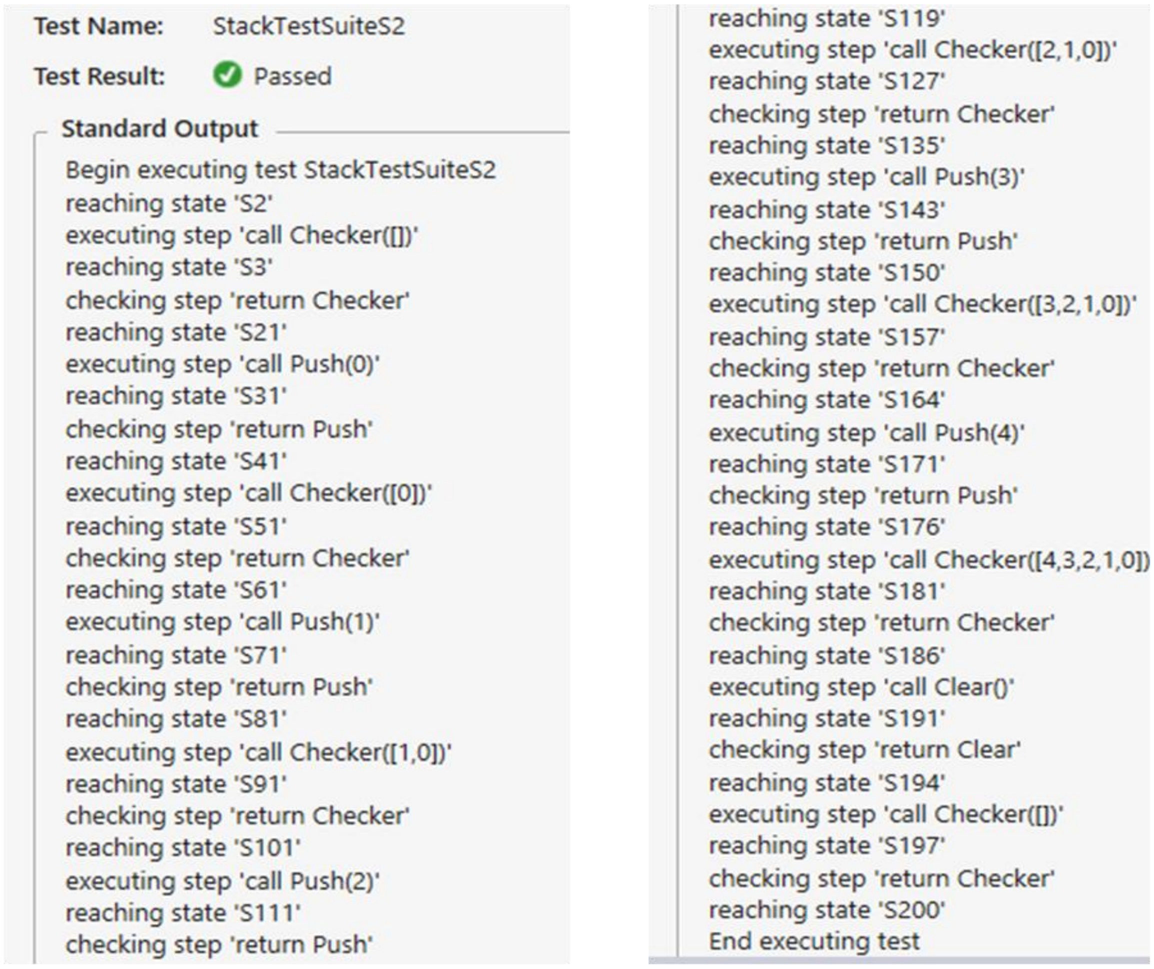


```
Test Name:     StackTestSuiteS18
Test Result:   ✓ Passed

Standard Output
Begin executing test StackTestSuiteS18
reaching state 'S18'
executing step 'call Checker([])'
reaching state 'S19'
checking step 'return Checker'
reaching state 'S29'
executing step 'call Push(0)'
reaching state 'S39'
checking step 'return Push'
reaching state 'S49'
executing step 'call Checker([0])'
reaching state 'S59'
checking step 'return Checker'
reaching state 'S69'
executing step 'call Clear()'
reaching state 'S79'
checking step 'return Clear'
reaching state 'S89'
executing step 'call Checker([])'
reaching state 'S99'
checking step 'return Checker'
reaching state 'S109'
End executing test
```

Figure 4.13: Standard output of test cases StacktestsuitesS18

Figure 4.13 shows the standard output generated by the test case named "StackTestSuiteS18". The test case starts its execution from the state "S18" and ends its execution by reaching at the state "S109".

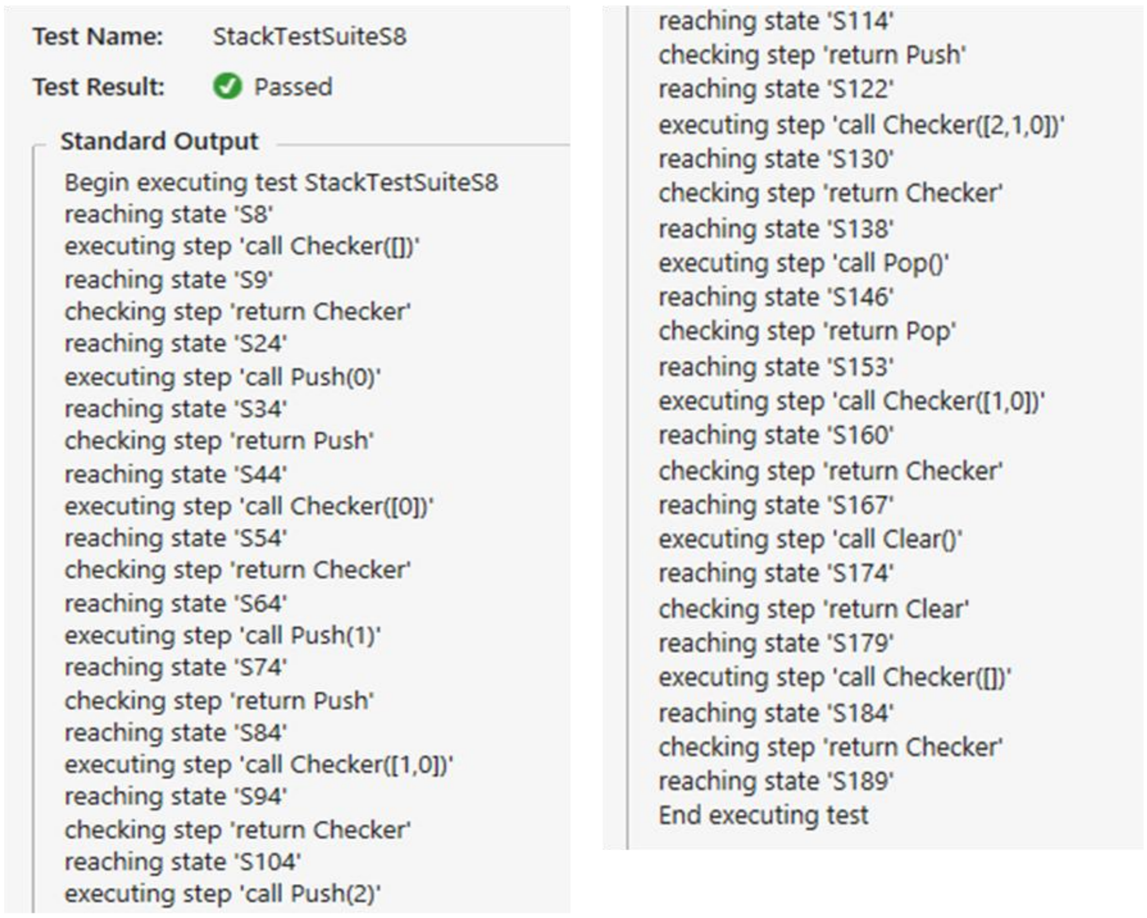


```
Test Name:    StackTestSuiteS2
Test Result:  Passed

Standard Output
Begin executing test StackTestSuiteS2
reaching state 'S2'
executing step 'call Checker([])'
reaching state 'S3'
checking step 'return Checker'
reaching state 'S21'
executing step 'call Push(0)'
reaching state 'S31'
checking step 'return Push'
reaching state 'S41'
executing step 'call Checker([0])'
reaching state 'S51'
checking step 'return Checker'
reaching state 'S61'
executing step 'call Push(1)'
reaching state 'S71'
checking step 'return Push'
reaching state 'S81'
executing step 'call Checker([1,0])'
reaching state 'S91'
checking step 'return Checker'
reaching state 'S101'
executing step 'call Push(2)'
reaching state 'S111'
checking step 'return Push'
reaching state 'S119'
executing step 'call Checker([2,1,0])'
reaching state 'S127'
checking step 'return Checker'
reaching state 'S135'
executing step 'call Push(3)'
reaching state 'S143'
checking step 'return Push'
reaching state 'S150'
executing step 'call Checker([3,2,1,0])'
reaching state 'S157'
checking step 'return Checker'
reaching state 'S164'
executing step 'call Push(4)'
reaching state 'S171'
checking step 'return Push'
reaching state 'S176'
executing step 'call Checker([4,3,2,1,0])
reaching state 'S181'
checking step 'return Checker'
reaching state 'S186'
executing step 'call Clear()'
reaching state 'S191'
checking step 'return Clear'
reaching state 'S194'
executing step 'call Checker([])'
reaching state 'S197'
checking step 'return Checker'
reaching state 'S200'
End executing test
```

Figure 4.14: Standard output of test cases StacktestsuiteS2

Figure 4.14 shows the standard output generated by the test case named "StackTestSuiteS2". The test case starts its execution from the state "S2" and ends its execution by reaching at the state "S200".



```
Test Name:    StackTestSuiteS8
Test Result:  ✓ Passed

Standard Output
Begin executing test StackTestSuiteS8
reaching state 'S8'
executing step 'call Checker([])'
reaching state 'S9'
checking step 'return Checker'
reaching state 'S24'
executing step 'call Push(0)'
reaching state 'S34'
checking step 'return Push'
reaching state 'S44'
executing step 'call Checker([0])'
reaching state 'S54'
checking step 'return Checker'
reaching state 'S64'
executing step 'call Push(1)'
reaching state 'S74'
checking step 'return Push'
reaching state 'S84'
executing step 'call Checker([1,0])'
reaching state 'S94'
checking step 'return Checker'
reaching state 'S104'
executing step 'call Push(2)'
reaching state 'S114'
checking step 'return Push'
reaching state 'S122'
executing step 'call Checker([2,1,0])'
reaching state 'S130'
checking step 'return Checker'
reaching state 'S138'
executing step 'call Pop()'
reaching state 'S146'
checking step 'return Pop'
reaching state 'S153'
executing step 'call Checker([1,0])'
reaching state 'S160'
checking step 'return Checker'
reaching state 'S167'
executing step 'call Clear()'
reaching state 'S174'
checking step 'return Clear'
reaching state 'S179'
executing step 'call Checker([])'
reaching state 'S184'
checking step 'return Checker'
reaching state 'S189'
End executing test
```

Figure 4.15: Standard output of test cases StacktestsuitesS8

Figure 4.15 shows the standard output generated by the test case named "StackTestSuiteS8". The test case starts its execution from the state "S8" and ends its execution by reaching at the state "S189".

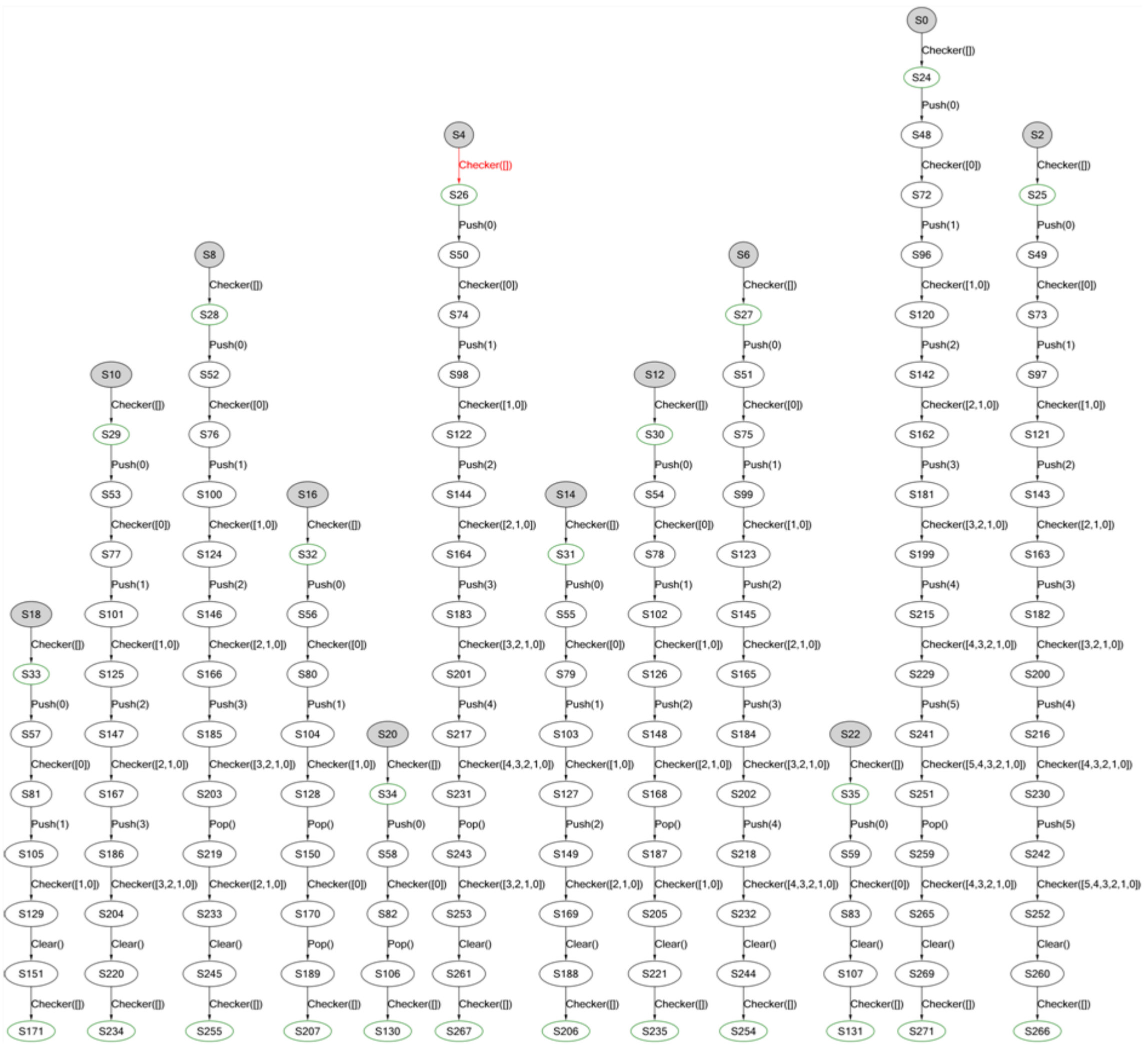


Figure 4.16: Representation of Stack Test Suite

Figure 4.16 shows the pictorial representation of all the generated test cases. To explore any transition or any state, that one can be clicked in the spec explorer and analyzed.

4.4 Prepare some mockups of the SUT (System under Test)

We know that stack is also known as LIFO (last in first out). It is represented as collection of elements. This Stack has various operations:

- Push
  - Push means insert an element in the stack.
- Pop
  - Pop means remove the last elements from the stack.
- Clear
  - It clears the stack.
- Checker
  - The Checker[] reads a particular state and gives the elements in the stack.

4.5 Run the Generated Test Cases against the SUT

The test cases which were generated by the spec explorer from the FSM of the system now need to be run against the SUT. The overall process is depicted in the Figure ----- .After generating test cases from the system, a test suit of ten test cases was obtained. Now these test cases were provided a reference of the SUT. The SUT was a small implementation of the behavior of system. Now the SUT is against the generated test suit. After this all four faults were inserted in the system one by one and the test suit was run against the faults. The results were obtained as a fault matrix. This fault matrix was consisted of test cases at one end and faults at the other end. The generated test cases could be run on the fly and could be verified to see whether they are failing or passing.

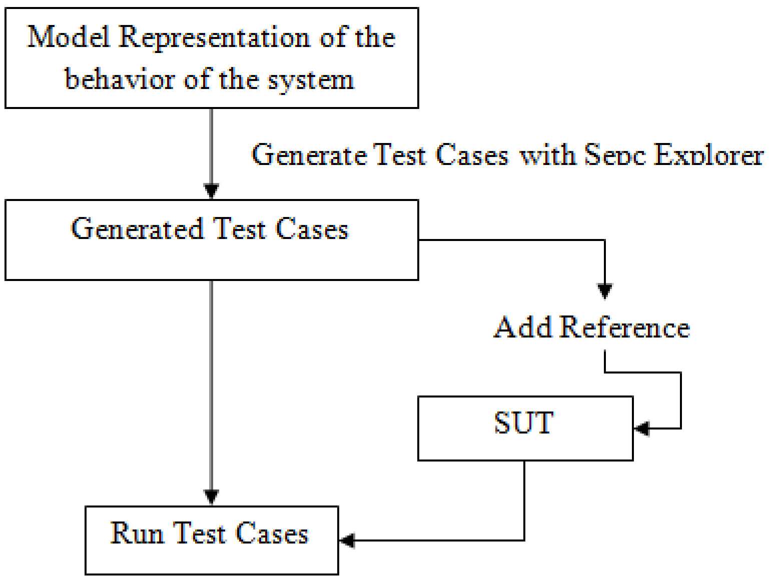

Model Representation of the behavior of the system
Generate Test Cases with Sepc Explorer
Generated Test Cases
Add Reference
SUT
Run Test Cases

Figure 4.17: Testing SUT against generated test cases

The fault matrix that was generated is depicted in the table 4.2. More on this is in the section 4.7

4.6 Analyze code Coverage and Calculate APBC

APBC stands for average percentage block coverage. This metric also represented as average percentage branch coverage. APBC metric measures the rate at which a prioritized test suite covers the various blocks and branches.

Table 4.1 Block Coverage by Model Based Test Cases

Block Coverage by Model Based Test Cases				
Test Case	**Block Not Covered**	**Blocks Covered**	**Blocks Covered (%)**	**Blocks Not Covered (%)**
STS0	15	31	67%	33%
STS10	17	29	63%	37%
STS12	17	29	63%	37%
STS14	17	29	63%	37%
STS16	19	27	59%	41%
STS18	19	27	59%	41%
STS2	17	29	63%	37%
STS4	15	31	67%	33%
STS6	17	29	63%	37%
STS8	15	31	67%	33%
		APBC	**63%**	

Table 4.1 shows the block coverage i.e. the percentage of code blocks covered by the generated test cases. In the spec explorer the coverage is shown for both the generated test suit code and implementation of the SUT. But here the concern is about the SUT, so the percentage of code blocks covered by each test case is shown here. From this table APBC was calculated as follows:

Block Coverage by a test case = Number of blocks covered by a test case/ Total number of blocks

e.g.: Block Coverage for test case STS0

= (31 / (31+15))*100

= (31/ 46)*100

= 67%

Then APBC is calculated as:

APBC= Sum of block coverage by each test case/ Total number of test cases.

= ((67+63+63+63+59+59+63+67+63+67) / 10)*100

= 63%

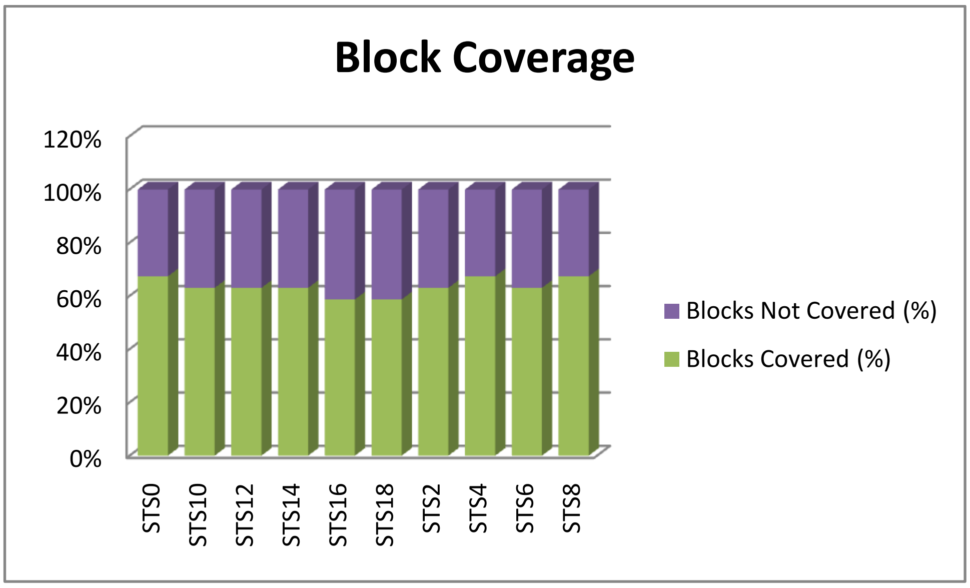


Figure 4.18: Block coverage

Figure 4.18 shows the percentage of blocks covered and percentage of blocks not covered by each test case.

4.7 Associate Faults with the System and Calculate APFD

Now the next step is to associate some faults with the implementation and calculate the APFD i.e. Average percentage of Faults detected. For this, some Restrictions were imposed on the stack to increase the number of faults that can be associated with the system make testing more interesting. The restrictions that were imposed on the system are as follows:

1. The value at a particular level in the stack will be equal to the level number e.g.
Value at level 0 will be 0; value at level 1 will be 1 and so on.

After imposing the above restriction, now there is need to associate some faults with the system implementation so that we can check that whether the test cases are able to detect the faults that can be there in the system or not. There were four faults associated with the system implementation. There four faults are as follows:

Faults:

1. Perform POP on Empty stack initially.
2. After clearing whole stack, Perform POP operation.
3. Perform more than one consecutive POP operation.
4. Perform more than one consecutive Push operation.

After associating all these faults one by one with the system implementation and running all the test cases in some arbitrary order, the fault matrix was obtained, which is as follows:

Table 4.2: Number of Faults Detected By each Model Based Test Case and APFD Calculation

Number of Faults Detected By each Model Based Test Case and APFD Calculation										
Faults	**Test Cases**									
	STS0	STS10	STS12	STS14	STS16	STS18	STS2	STS4	STS6	STS8
1	1	1	1	1	1	1	1	1	1	1
2	1	1	0	1	0	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
Total Number of faults found	4	4	3	4	3	4	4	4	4	4
				APFD	95%					

This fault matrix shows that whether the fault was detected by a particular test case or not. For the purpose of representation, if a fault was detected by a test case then "1" was marked in the particular cell and if it was not detected then "0" was marked in the particular cell. Table 4.2 shows which test case has detected which fault. E.g.: "STS0" detected all four faults that is why "1" is marked in each cell below the test case name and "STS12" had not detected fault "2", so "0" was marked in the particular cell under test case "STS12". This is how fault matrix was obtained. Now this fault matrix can be used to calculate the APFD value of a test suit run.

To calculate APFD value for a test run one should find out the first test case that is detecting the particular fault in a particular run. For this an arbitrary order of test cases execution was set (the order in which test cases were generated and stored by the tool) and then that test suit was run against the each fault.

In this case where test cases were generated from FSM of the system and where four faults were taken into consideration, test case "STS0" was able to find out all the faults. So the faults matrix obtained by running test cases in order STS0, STS10, STS12, STS14, STS16, STS18, STS2, STS4, STS6, STS8 is shown in the Table 4.2.

Now, to calculate APFD value for this run, the formula is

APFD = 1 – ((first test case that found fault 1 + first test case that found fault 2 + … + first test case that found the nth fault) / (n * m)) + (1 / (2 * n))

Here n = total number of test cases in test suit

And m = total number of faults associated

As per this equation the APFD value for non prioritized test cases is as follows:

= 1 - ((1+1+1+1)/ (10*4)) + (1/ (2*10))

= 1 - (4/40) + (1/20)

= 1 - 0.10 + 0.05

= 1.05 – 0.10

= .95

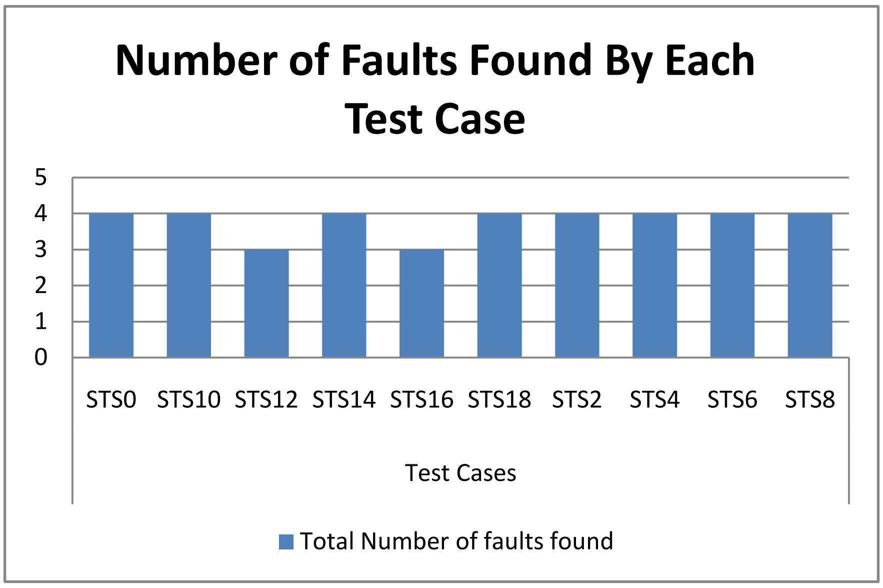


Figure 4.19: Number of faults found by each test case

Figure 4.19 just shows a graphical representation of the number of faults detected by each test case if it is executed in the same order as depicted in the table 4.2.

4.8 Prioritize Test Cases by Analyzing Fault Matrix and another Parameter Execution Time

Now, prioritization can be done by taking into consideration the fault matrix. As can be seen in the fault matrix in Table 4.2 that there are two test cases named “STS12” and “STS16” which are not detecting the fault 2, so these two test cases can be moved to the last position giving the order of execution as STS0, STS10, STS14, STS18, STS2, STS4, STS6, STS8, STS12, STS16 as shown in the Table 4.3. Even it will not affect the APFD value calculated because it is based on the fact of finding the first test case that is detecting a particular fault.

Table 4.3: Number of Faults Detected By each Model Based Test Case and APFD Calculation after prioritization based on Fault matrix

Faults	**Test Cases**									
	STS0	STS10	STS14	STS18	STS2	STS4	STS6	STS8	STS12	STS16
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	0	0
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
Total Number of faults found	4	4	4	4	4	4	4	4	3	3
			APFD	0.95						

As it can be seen that there is no effect on the value of APFD after prioritizing test cases based on the values obtained in the fault matrix but it is making the testing results more reliable. Because there was not any fault left after least execution of test suit as it may be the case that for some arbitrary order of test case execution, “STS12” and “STS16” may execute at the very first and will not detect the fault 2. So prioritization in this manner is giving reliability.

Another factor that was taken into consideration for prioritizing these test cases was “Execution time” of each test case to find out a particular fault.

During the process of finding the faults by executing the generated test cases, the execution time of each test case was calculated as is shown in the Table 4.4 as follows:

Table 4.4: Execution Time Taken by Each Test case to find each fault wherever applicable

Faults	**Test Cases**									
	STS0	STS10	STS12	STS14	STS16	STS18	STS2	STS4	STS6	STS8
1	100	7	6	5	3	3	10	10	8	8
2	61	6	NA	5	NA	3	11	10	8	8
3	61	6	6	5	3	3	11	10	9	8
4	56	6	6	4	4	3	10	10	9	8
Total Time Taken By Each Test Case	278	25	NA	19	NA	12	42	40	34	32

Table 4.4 shows the time taken to find out the fault by a test case wherever applicable. As there are ten test cases and four faults. If test suite STS0, STS10, STS12, STS14, STS16, STS18, STS2, STS4, STS6, STS8 is executed, no doubt that it will give APFD value = .95 as is calculated in the table 4.2, but if we take into consideration the execution time also then according to table 4.4, for .95 APFD value the execution time is 278 ms (milliseconds taken into consideration for this small system) which is maximum in all the executions. It is a small system so 278 ms may not affect the overall cost of the system which also relies on the time taken for testing and system and finding all the faults, but for large system if only APFD value is taken into consideration then cost may increase.

Here two test cases which were not able to detect the fault 2 were represented with the NA.

Now if both APFD based prioritization and Execution time are taken into consideration then the values obtained would be as follows:

Table 4.5: Prioritization of test cases on APFD and execution time Basis

Faults	**Test Cases**									
	STS18	STS14	STS10	STS8	STS6	STS4	STS2	STS0	STS16	STS12
1	3	5	7	8	8	10	10	100	3	6
2	3	5	6	8	8	10	11	61	NA	NA
3	3	5	6	8	9	10	11	61	3	6
4	3	4	6	8	9	10	10	56	4	6
Total Time Taken By Each Test Case	12	19	25	32	34	40	42	278	NA	NA

It is clear from the table 4.5 that if we take into consideration execution time along with APFD value then the overall cost of the testing can be minimized. As in this case are faults were detected with an APFD value of .95 but in that case the execution time was largest among all i.e. 278 m s. But same APFD value can be achieved with much reduced time i.e. 12 m s.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE

Software testing is the process of evaluating software to detect differences between given input and expected output. And regression testing is done whenever any change is made in the system. This change could be any error removal or any update. In regression testing, after any modification that modified part and the whole system is tested again to check that the modified part has not made any defect in the working system.

For this the whole system is tested again. So it is costly due to large size of test suites in system retesting. There will be large amount of time consume and more computing resources would be used. To reduce the cost and time some techniques can be applied to reduce the test suit which is to be applied to retest the system. It could be done by applying one of technique that is test case prioritization.

Test case prioritization technique is that technique which schedule test cases in an execution order according to some criterion. By this technique the maximum number of faults can be found in less time and cost and so effectiveness will also improve. So, we conclude that we improved the effectiveness of test case prioritization technique for regression testing and prioritize test case by generating them from model of the system. Prioritization was made by analyzing APFD value for non-prioritized test cases and also another factor i.e. execution time. Analysis was made and our results showed better results than the earlier research made in this field.

As a part of future work, this work was done on a small system and a small unit of time i.e. millisecond was taken into consideration along with four faults and ten test cases. For this research not to be system biased, it can be done on large system and other behavioral representations of the system like sequence diagrams.

REFERENCES

[1]. Sanjukta Mohanty, Arup Abhinna Acharya, Durga Prasad Mohapatra "A Survey On Model Based Test Case Prioritization" International Journal of computer science and information technologies, vol. 2(3), (2011) ISSN: 0975-9646, ISSN: 1042-1047.

[2]. L.UmaRani, R.Pradeepa "Prioritization Based Test Case Generation Using Regression Testing" international journal of computer application, (2013), ISSN: 0975-8887.

[3]. Chandana Bharati, Shradha verma, " Analysis Of Different Regression Testing Approaches", international journal of advanced research in computer and communication engineering, vol.2, issue 5, May 2013, ISSN(print): 2319-5940, ISSN(online): 2278-1021.

[4]. Yanping Chen, Robert L.Probert, D. paul Sims "Specification-Based Regression Test Selection With Risk Analysis".

[5]. Leila Naslavsky "Using Traceability To Support Model Based Regression Testing", ASE'07, November 5-9, 2007, Atlana, Georgia, USA.

[6]. Bogdan Korel, George , "Experimental Comparison Of Code-Based And Model-Based Test Prioritization", IEEE International conference on software testing verification and validation workshop, DOI 10.1109/ICSTW.2009.45

[7]. Xiaobo Han ,Hongwei zeng, Honghao gao, "A Heuristic Model-Based Test Prioritization Method For Regression Testing", IEEE International symposium on computer , consumer and control, DOI 10.1109/IS3C.2012.226.

[8]. Ms. Sujata, Nancy Dhamija, " Test Case Prioritization Using Model-Based Test dependencies: A Survey ", International journal of information and computation technologies, ISSN:0974-2239 VOLUME 4, NUMBER 10(2014).

[9]. Leila Naslavsky, Hadar ziv, Debra j. Richardson, "Model-Based Selective Regression Testing With Traceability", Third IEEE International conference on software testing verification and validation, DOI 10.1109/ICST.2010.61.

[10]. Hema Srikanth, Laurie Williams, "Requirement-Based Test Case Prioritization".

[11]. Kavitha V..R, "Requirement-Based Test Case Prioritization",

[12].http://www.softwaretestinghelp.com/regression-testing-tools-and-methods/

[13] Praveen Ranjan Srivastva, "Test case Prioritization" , Journal of theoretical and applied information technology 2005-2008 JATIT.

[14]. Prateeva Mahali," MODEL BASED TEST CASE PRIORITIZATION USING UML ACTIVITY DIAGRAM AND EVOLUTIONARY ALGORITHM", International Journal of Computer Science and Informatics, ISSN (PRINT): 2231 –5292, Volume-3, Issue-2, 2013 .

[15]. Arup Abhinna Acharya," Model Based Test Case Prioritization for Testing Component Dependency in CBSD Using UML Sequence Diagram", (IJACSA) International Journal of Advanced Computer Science and Applications,
Vol. 1, No. 6, December 2010.

[16]. Eliane Collins "Strategies for Agile Software Testing Automation: An Industrial Experience ", 2012 IEEE 36th International Conference on Computer Software and Applications Workshops.

[17]. Bogdan Korel, “Model-Based TestPrioritization Heuristic Methods And Their Evaluation”, 2007 AMOST’07 , LONDON, UK.

[18252Fany%252F%3B1099%3B744

[18].http://www.tutorialspoint.com/software_testing/

APPENDIX

ABBREVIATIONS

APFD	Average Percentage Fault Detection
PORT	Prioritization of Regression Testing
PFs	Prioritization Factors
TCP	Test Case Prioritization
MLP	Most Likely Position
EFSM	Extended Finite State Machine
APBC	Average Percentage of Block Coverage
GA	Genetic Algorithm