



RESEARCH ARTICLE

Automated Model-Based Test Path Generation from UML Diagrams via Graph Coverage Techniques

Parampreet Kaur¹, Gaurav Gupta²

¹Computer Engineering, University College of Engineering, Punjabi University, Patiala, India

²Computer Engineering, University College of Engineering, Punjabi University, Patiala, India

¹ paramnagpal16@gmail.com; ² gaurav_shakti@yahoo.com

Abstract— UML State Chart Diagrams are the basic models used to derive test paths from intermediate graphs generated automatically using graph coverage techniques in addition to the tool support provided by MBT Tool TestOptimal's Basic as well as ProMBT version. The test Paths Generated covers Node Coverage, Edge Coverage, Edge Pair Coverage as well as most importantly Prime Path coverage which is till today not explored much. The algorithm employed is Prefix based combined with Chinese postman Problem Algorithm together. From State charts, first of all Model Coverage Graphs are constructed with help of TestOptimal and then Test Paths are generated one by one. Testing is often incomplete, i.e. cannot cover all possible system behaviours. There are several heuristic means to measure the quality of test suites, e.g. fault detection, mutation analysis, or coverage criteria. These means of quality measurement can also be used to decide when to stop testing. This paper is centred upon coverage criteria. There are many different kinds of coverage criteria, e.g. focused on data flow, control flow, transition sequences, or boundary values. In this paper, we will present new approaches, e.g. to combine coverage criteria and generation of test paths manually as well as automatically using tools based on Chinese postman and prefix based algorithms.

Key Terms: - SUT; TestOptimal; State charts; MCG; STG

I. INTRODUCTION

The work on this paper is focused on functional model-based testing. Functional testing is associated with verifying of the system under test (SUT) with a Software Requirement Specification (SRS). A functional test detects a failure if the observed and the specified behaviour of the SUT do not match. Model-based testing is about using models as specifications. The most intellectually challenging part of testing is the design of test cases. Test cases are usually generated based on program source code. An alternative approach is to generate test cases from specifications developed using formalisms such as UML models. In this approach, test cases are developed during analysis or design stage itself, preferably during the low level design stage. Design specifications are intermediate artefacts between requirement specification and final code. They preserve the essential information from the requirement, and are the basis of code implementation. Test case generation from design specifications has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. It is therefore desirable to generate test cases from the software design or analysis documents, in addition to test case design using the code. UML-based automatic test case generation is a practically important and theoretically challenging topic. Literature survey

indicates, testing based on UML specifications is receiving an increasing attention from researchers in the recent years. In using UML in the software testing process, here we focus primarily on the state chart diagrams which are then modelled into corresponding coverage Model Graphs as well as Test Sequence Graphs instead of simply converting into FSMs. Model-based testing creates tests from abstract models of the software. These models are often described as graphs, and test requirements are defined as sub paths in the graphs. As a step toward creating concrete tests, complete test paths that include the sub paths through the graph are generated. Each test path is then transformed into a test. If we can generate fewer and shorter test paths, the cost of testing can be reduced. The minimum cost test paths problem is finding the test paths that satisfy all test requirements with the minimum cost.

II. LITERATURE SURVEY

AGEDIS (Automated Generation and Execution of Test Suites in Distributed Component based Software) was a three-year research project on the automation of software testing funded by the European Union. Five case studies were conducted. These studies focused on applying model based testing methods and tools to test problems in industrial settings. The studies were conducted at France Telecom, Intrasoft and IBM. The findings showed that models increased the comprehensibility of the system under test and was found to be an efficient way to analyse complex requirements. It was also found that when a requirement changed, adapting the model and regenerating the test cases required less effort compared to updating manually constructed test cases [11].

Several researches have productively proposed test case generation for various softwares under different circumstances, such as scenario-based, model based, path-oriented, goal-oriented and genetic approaches. Scenario based techniques test cases are based on concurrent approach of coverage criteria. Model based techniques identify respective test cases for the software with respect to the UML diagrams such as sequence, activity, state-chart, class or object diagram etc. Path-oriented testing is based on static as well as dynamic control flow of the software. Static path testing is done by symbolic execution whereas dynamic path testing is based on the run time test of executing program. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken. Many researchers and doctorates have been working in field of software testing towards generation of test cases. Among the survey results, most of them use modelling language to generate test cases. Since Unified Modelling Language is a standardized general-purpose modelling language in the field of software engineering. UML diagrams represent two different views of a system model static and dynamic. Static (or structural) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams. Dynamic or behavioural view: emphasizes the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

Sanjai [1] presented a method for automatically generating test cases to structural coverage criteria. He showed how, given any software development artifact that can be represented as a finite state model, a model checker can be used to generate complete test cases that provide a predefined coverage of that artifact. He provided a formal framework that is: (a) suitable for defining their test-case generation approach and (b) easily used to capture finite state representations of software artifacts such as program code, software specifications, and requirements models. He showed how common structural coverage criteria can be formalized in their framework and expressed as temporal logic formulae used to challenge a model checker to find test cases. Finally, he demonstrated how a model checker can be used to generate test sequences for modified condition and decision coverage. Their approach is to generate test cases using the model checker as the core engine. A set of properties called trap properties, is generated and the model-checker is asked to verify the properties one by one. These properties are constructed in such a way that they fail for the given system specification.

Aynur [2, 3] defined the following definition in their work: (a) test requirements are specific things that must be satisfied or covered during testing and (b) test specifications are specific descriptions of test cases including test data, often associated with test requirements or criteria. They presented a test data generation method, based on Offut's state-based technique, to prepare and generate a set of data from UML state charts diagram. They proposed to use the TSL language to describe all elements of a test case, like input, output and pre-condition. However, they concentrate on the following elements: (a) precondition values (b) verify values (c) exit command and (d) expected output data. Generally, those elements are directly derived from triggering events and pre-conditions in the state chart diagram. The pre-condition values include all required input data. Any input data, which are required to show the results, are the verify values. The exit commands are depended on the

system or program being tested. The expected output data are created from the after-values of the triggering events and post conditions.

Samuel [6] presented an approach to generate test sequences from the UML sequence diagrams, version 2.0. UML Sequence diagrams are one of the most widely used UML models in the software industry. They found that existing test sequence generation techniques do not encompass certain important features of the UML sequence diagrams, version 2.0. Thus, they proposed an effective method to generate a set of test steps by considering many key features of UML sequence diagram, version 2.0, like loop, alt and break feature.

M.Prasanna et al. (2009) describes that to test a software, test cases generation is best way. The test cases are derived by analyzing the dynamic behaviour of the objects due to internal and external stimuli [5]. Test cases can be generated with the help of UML diagrams. Researchers use model based approach in which genetic algorithms crossover technique is applied on the class diagram and the traversal is done by the depth first search (DFS) algorithm. This tree structure approach coupled with genetic algorithm shows that it is capable to reveal 80% faults in unit level and 88% faults in integration level. They coupled the genetic algorithm with mutation testing to check the effectiveness in the testing process which shows 80.3% of effectiveness. The result shows methodology is useful to generate test cases after the completion of the design phase and error could be detected at an early stage in software development life cycle.

Automatic test case generation using unified modelling language (UML) state diagrams by P. Samuel, R. Mall, A.K. Bothra published on basis of model based test case generation automatically. They explored in their approach, the control and data flow logic available in the UML state diagram to generate test data. The state machine graph is traversed and the conditional predicates on every transition are chosen. Then these conditional predicates are transformed and functional minimization technique is applied to generate test cases. The present test data generation scheme is fully automatic and the generated test cases satisfy transition path coverage criteria. The generated test cases can be used to test class as well as cluster-level state-dependent behaviours [5]. Test data generated using this approach is verified based on the path coverage. The step involved are, the first step is to select a predicate. In this, select a predicate on a transition from a UML state machine diagram. The next step is to transform the selected predicate to a predicate function. In the third step, generate test data corresponding to the transformed predicate function. This Approach can handle change events, time events and transitions with guards, and achieves transition path coverage.

Dirk Seifert [4] presented in Test Case Generation from UML State Machines that test cases include not only stimuli to trigger the SUT, they also include possible correct observations to automatically evaluate the test case execution. In comparison to classical Harel State charts, state machines behave in asynchronous manner, which makes automatic test case generation a challenge. The TEAGER Tool Suite implements the automatic generation, execution and evaluation of test cases and proves the applicability of test approach. It is possible to select relevant and interesting inputs for a test case and to calculate the possible correct observations for given inputs. They allow to automatically evaluating test executions which is a difficult and time consuming task. Applied approximation makes the generation process practical, whereas it is possible to control this process depending on the time and computation power to invest.

Automated-Generating Test Case Using UML State chart Diagrams by Supaporn Kansomkeat and Wanchai Rivepiboon experimented on the automatic testing technique to solve partially the testing process. This technique can automatically generate and select test cases from UML state chart diagrams. Firstly, transform this diagram into intermediate diagram, called Testing Flow Graph (TFG), explicitly identify flows of UML state chart diagrams and raise them for testing. Secondly, from TFG generate test case using the testing criteria that is the coverage of the state and transition of diagrams. Finally, the evaluation is performed using mutation analysis to assess the fault revealing power of test cases [7]. Specification based testing uses information derived from a specification to assist testing as well as to develop program. Testing activities consist of designing test cases that are a sequence of inputs, executing the program with test cases, and examining the results produced by this execution. Automatically generate test cases from UML specification with the aid of the Rational Software Corporation's Rational Rose tool. Test cases are measured the effectiveness on the basis of their fault detection abilities. Results of simple test experiments are high effectiveness of the generated test cases. However, extensive experiments are needed to have more confidence of the testing technique and to compare it with other techniques in term of cost and effectiveness.

Ranjit Swain, Vikas Panthi, Durga Prasad (2012) in their paper presented different techniques to generate the test cases to test the software. The functional minimization technique is also used to generate the test cases. In this technique first predicate is selected and then transformed to create test cases. The functional minimization

technique is used for finding the minimum of predicate function. In this approach the test cases are generated step by step. Here the object diagram that is used for generating the test cases is state machine diagram. This approach covers state coverage, transition pair coverage, action coverage. The numbers of test cases are minimized that achieve transition path coverage by testing the borders determine by simple prediction. It is found that test cases are generated from the object diagram by minimizing the cost and time. It can also handle transitions with guards and achieves transition path coverage [8].

Stephan Weissleder [10] proposed that UML state machines are widely used as test models in model based testing. Coverage criteria are applied to them, e.g. to measure a test suite's coverage of the state machine or to steer automatic test suite generation based on the state machine. The model elements to cover as described by the applied coverage criterion depend on the structure of the state machine. Model transformations can be used to change this structure. He presented semantic preserving state machine transformations that are used to influence the result of the applied coverage criteria. The contribution is that almost every feasible coverage criterion that is applied to the transformed state machine can have at least the same effect as any other feasible, possibly stronger coverage criterion that is applied to the original state machine. Simulated satisfaction as a corresponding relation between coverage criteria is introduced. He provided formal definitions for coverage criteria and used them to prove the correctness of the model transformations that substantiate the simulated satisfaction relations. The results are especially important for model-based test generation tools, which are often limited to satisfy a restricted set of coverage criteria.

Utting and Legeard [11] presented four test generation approaches in model-based testing: 1) Generation of test input data from a domain model, 2) Generation of test cases from an environment model, 3) Generation of test cases with oracles from a behavior model, and 4) Generation of test scripts from abstract tests. The first three approaches are all based on test generation from a model. The domain model describes the domains of input data that can be given to the system and the environment model the expected environment, such as operation frequencies, of the SUT. Both models can be used to generate input for the SUT, but either does not include the expected output of the system, thus requiring manual work for verification. The third model, behavioural model, includes oracle information about the expected behavior of the system and can thus be used to detect any irregularities in the output of the SUT automatically. The fourth approach does not include models as such, but rather test cases that are described in a high-level of abstraction, without the low-level implementation details. Basically a script defined with high-level keywords could be categorized as the fourth approach.

III. RESEARCH CRITERIA

Figure shows taxonomy for model-based testing that is taken from [9]. It describes the typical aspects of model-based test generation and test execution. Its focus is on the various kinds of models and test generation techniques. We use this taxonomy to position our thesis: The subject of our test models is the SUT. The test models are separate from the development models. Furthermore, the used test models are deterministic, untimed, and discrete. The paradigm of the test model is transition-based, i.e. with a guided depth-first graph search algorithm. Satisfying structural model coverage is used to steer the test generation. The search technology is based on a combination of graph search algorithm and symbolic backward execution. The test execution is mainly offline although online.

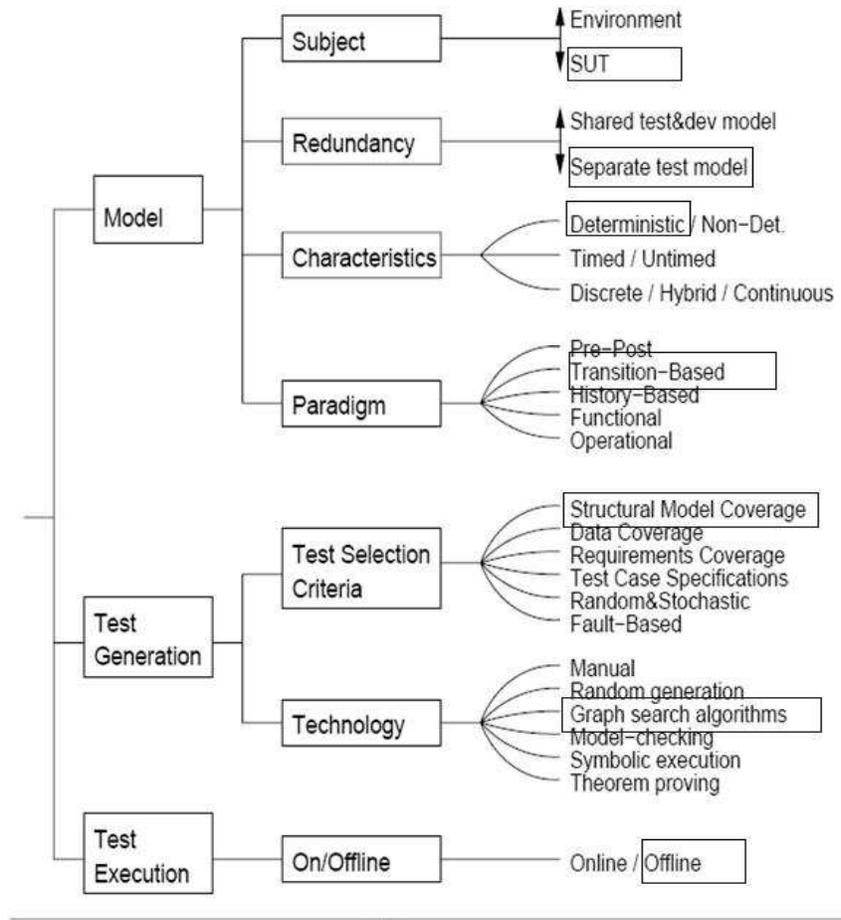


Fig: Flowchart Taxonomy according to Utting, Pretschner, Legeard [9]

IV. TEST PATH GENERATION ALGORITHM

The description of the abstract test case generation algorithm, whose purpose is the creation of an abstract test case with abstract information about inputs, is presented here. The algorithm starts at a certain point in the test model. From that point, the algorithm iterates backward in the state machine to the initial configuration with a guided depth-first graph search process and creates a corresponding path. While moving backward, the algorithm collects all conditions and keeps them in a consistent set of dataflow information.

```

TestCase createTestCase(te : TraceExtension)
{
    n = target node of the last transition of te;
    TestCase tc = searchBackwardsFromNode(n, te);
    if(tc is a valid test case)
    {
        return tc;
    }
    else {
        return null;
    }
}

TestCase searchBackwardsFromNode(n : Node, te : TraceExtension) {
    if(n is initial node and all expressions are satisfied)
    { // valid
        return test case that contains the current path information;
    }
}
    
```

```

}
TestCase tc = null;
if(n has a transition t that is part of te)
{
tc = traverseTransition(t, te);
if(tc != null)
return tc;
}
Else
{
for each incoming transition t of n {
tc = traverseTransition(t, te);
if(tc != null)
return tc;
}
}
return null;
}

```

V. TOOL USED: TESTOPTIMAL

TestOptimal is an integrated next-generation test design and test automation toolset powered by Model-Based Testing (MBT). Unlike QTP and TestComplete, TestOptimal helps to bring agility and efficiency to the testing process and shorten the testing cycle. TestOptimal BasicMBT, ProMBT, Enterprise, RuntimeMBT are a suite of model-based test automation tools for functional testing and load/performance testing. TestOptimal combines Model- Based Testing (MBT) and Data-Driven Testing (DDT) to provide a powerful test case generation and test automation tool. MBT enables to find defects earlier in the development cycle and respond to changes quickly and efficiently. Tracks requirement coverage and visualize test cases in various graphs. Choose one of many algorithms to generate test sequences for desired test coverage. Re-purpose same models and automation scripts for load and performance testing. TestOptimal can help reduce development cycle, achieve unprecedented test coverage and improve response to changes while gaining higher confidence in your software delivery. TestOptimal is a web based client server tool that tests desktop and multitier enterprise applications. A TestOptimal model is an FSM, created interactively while analyzing the web site being tested. It can also be imported in GraphML9, XMI10 and GraphXML11 formats. TestOptimal provides model validation, simulation and debugging support. It provides an XML based scripting language called mScript to connect adapter/driver of the model to the SUT. A tester can test do scenario testing using mCase. TestOptimal provides multiple algorithms to generate test cases and supports online and offline testing. It can be used for stress, load and regression testing. can add function logic to run generated test cases.

1) Test Generation

From this model, we then generate the series of sequences i.e. transition traversals which will walk through different web pages and cover all transitions in the model. To do this, we select the Optimal Sequencer in Model Property. From above model, to generate the test sequences click on  to display the test sequence in Traverse Graph.

2) Test Automation and Execution

Test sequence is just a series of steps .We can certainly follow this sequence by manually clicking the web pages to test. With TestOptimal, one can automate this with a set of simple mScript i.e. XML based scripting. With the model and mScript, one can execute the model and test the application by clicking on the run button and watch the web pages being clicked away automatically.

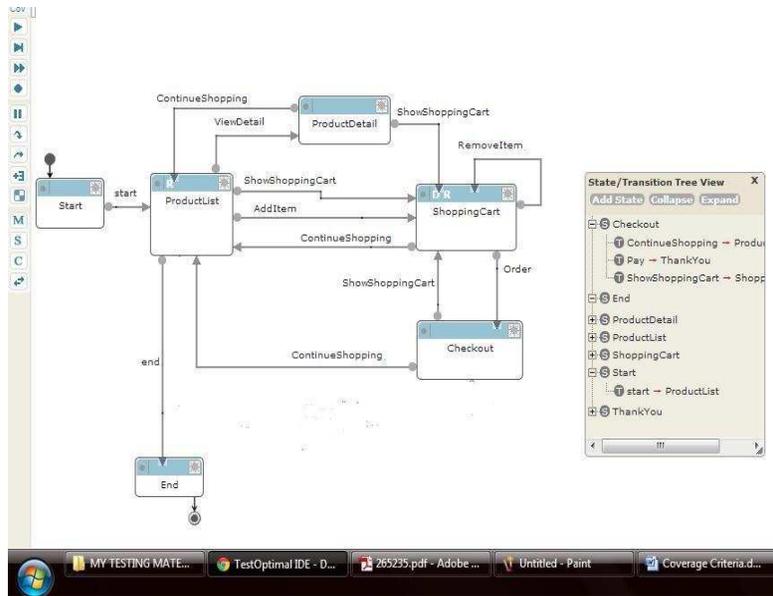


Fig 25: Statechart of Webstore – online shopping site.



Fig 26: Test Sequence Graph (TSG)

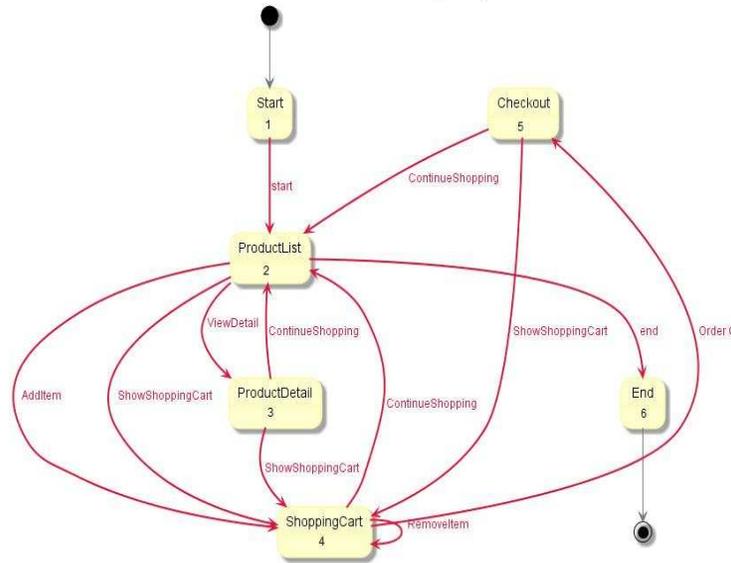


Fig 32 : Model Execution Coverage Graph (MECG)

According to the algorithm applied, 5 test Paths are generated for edge coverage:

Test Paths	Generated TPs
Tp-1	1,2,6
Tp-2	1,2,4,5,2,4,2,6
Tp-3	1,2,4,5,4,5,2,6
Tp-4	1,2,3,4,5,4,2,3,4,2,6
Tp-5	1,2,4,5,2,3,4,2,6

But by using Tool that uses Prefix Graph algorithm for edge-pair coverage, following test paths have been generated.

6 test paths are needed for Edge-Pair Coverage using Prefix Graph

Test Paths	Test Requirements that are toured by test paths directly
[1,2,3,4,2,4,2,4,5,4,2,6]	[1,2,3], [2,3,4], [3,4,2], [4,2,4], [4,2,4], [4,2,6], [2,4,2], [2,4,5], [2,4,2], [2,4,5], [4,5,4], [5,4,2]
[1,2,4,5,2,4,2,6]	[1,2,4], [1,2,4], [4,2,6], [2,4,2], [2,4,5], [2,4,2], [2,4,5], [4,5,2], [5,2,4], [5,2,4]
[1,2,4,5,4,5,2,6]	[1,2,4], [1,2,4], [2,4,5], [2,4,5], [4,5,2], [4,5,4], [5,2,6], [5,4,5]
[1,2,6]	[1,2,6]
[1,2,3,4,5,4,2,3,4,2,6]	[1,2,3], [2,3,4], [3,4,2], [3,4,5], [4,2,3], [4,2,6], [4,5,4], [5,4,2]
[1,2,4,5,2,3,4,2,6]	[1,2,4], [1,2,4], [2,3,4], [3,4,2], [4,2,6], [2,4,5], [2,4,5], [4,5,2], [5,2,3]

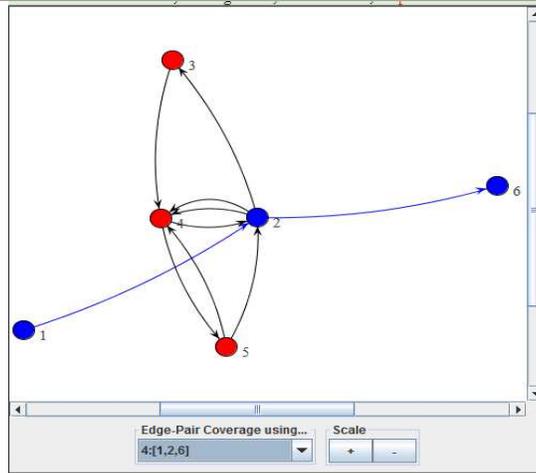


Fig: Edge-Pair Coverage using Prefix Graph method.

Prime-Path coverage: Generating Test Paths

Test Paths	Prime-Path coverage Test Paths
Tp-1	1,2,4,2,3,4,2,6
Tp-2	1,2,3,4,5,2,6
Tp-3	1,2,4,5,2,3,4,5,2,6
Tp-4	1,2,4,5,2,3,4,2,6
Tp-5	1,2,4,2,4,2,6
Tp-6	1,2,3,4,5,2,3,4,2,6
Tp-7	1,2,6

13 Test paths are needed for Prime Path Coverage

Test Paths	Test Requirements that are toured by test paths directly
[1,2,3,4,5,2,3,4,2,6]	[3,4,5,2,3], [2,3,4,5,2], [1,2,3,4,5], [4,5,2,3,4], [3,4,2,6], [2,3,4,2]
[1,2,3,4,5,2,6]	[2,3,4,5,2], [1,2,3,4,5], [3,4,5,2,6]
[1,2,4,5,2,3,4,5,2,6]	[2,3,4,5,2], [3,4,5,2,6], [5,2,3,4,5], [4,5,2,3,4], [2,4,5,2], [1,2,4,5]
[1,2,4,5,2,3,4,2,6]	[4,5,2,3,4], [3,4,2,6], [2,4,5,2], [1,2,4,5], [2,3,4,2]
[1,2,3,4,2,3,4,2,6]	[3,4,2,6], [3,4,2,3], [4,2,3,4], [2,3,4,2]
[1,2,4,2,3,4,2,6]	[3,4,2,6], [4,2,3,4], [2,3,4,2], [2,4,2]
[1,2,4,5,2,4,5,2,6]	[2,4,5,2], [1,2,4,5], [5,2,4,5], [4,5,2,4]
[1,2,4,5,4,2,3,4,2,6]	[3,4,2,6], [4,2,3,4], [1,2,4,5], [2,3,4,2], [5,4,2,3], [4,5,4]
[1,2,4,5,4,2,6]	[1,2,4,5], [5,4,2,6], [4,5,4]
[1,2,4,5,2,4,2,6]	[2,4,5,2], [1,2,4,5], [4,5,2,4], [2,4,2]
[1,2,4,2,4,2,6]	[4,2,4], [2,4,2]
[1,2,6]	[1,2,6]
[1,2,4,5,4,5,2,6]	[1,2,4,5], [5,4,5], [4,5,4]

Thus Automatic Generation leads to more test coverage paths thus resulting in efficient and effective testing strategy. The above automatic Generation of Test Paths are based on prefix graph algorithms which cover both edge pair as well as prime path coverage criteria and derive their concept using Chinese postman algorithm.

VI. CONCLUSIONS AND FUTURE WORK

Models are an excellent way to represent and understand system behavior, and they provide an easy way to update tests to keep pace with applications that are constantly changing and evolving. Testing an application can be viewed as traversing a path through the graph of the model. Graph theory techniques therefore allow us to use the behavioural information stored in models to generate new and useful tests. Because graph theory techniques deal directly with the model so new traversals can be automatically generated when the model changes. Tests can be constantly changing on the same model. Different types of traversals can meet different needs of testers. The traversal techniques are general and can be re-used on different models. Model-based testing is a black-box technique that offers many advantages over traditional testing: Firstly, Constructing the behavioural models can begin early in the development cycle. Secondly, Modelling exposes ambiguities in the specification and design of the software. The model embodies behavioural information that can be re-used in future testing, even when the specifications change. Moreover the model is easier to update than a suite of individual tests. And, most importantly, a model furnishes information that can be coupled with graph theory techniques to generate many different test scenarios automatically.

Testing benefits from the fact that the real system is brought to execution. Thus, the interaction of the real hardware and the real software can be evaluated. It aims in falsification, i. e. to show inconsistencies between the specification and developed system. Testing is applicable at different levels of abstraction and at different stages of the development. With our approach UML state machines can be used in the quality assurance to serve as a specification for the desired reactive behaviour of the system. It is possible to select relevant and interesting inputs for a test case and to calculate the possible correct observations for given inputs. They allow to automatically evaluating test executions which is in general a difficult and time consuming task. Applied approximation makes the generation process practical.

Our technique achieves much important coverage like state coverage, transition coverage, transition pair coverage, Prime path coverage. It can handle transitions with guards and achieves transition path coverage. Here the number of test cases is minimized and they achieve transition path coverage by testing the boundaries. Moreover, our planning is to include other diagrams of UML to generate test cases. In future, we will look into how the test cases can be optimized and how other UML diagrams can be combined and used to generate test cases and achieve higher coverage.

REFERENCES

- [1] Sanjai Rayadurgam and Mats P. E. Heimdahl, "Test-Sequence Generation from Formal Requirement Models", Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering (HASE'01), 2001.
- [2] Jeff Offutt, Shaoying Liu, Aynur Abdurazik and Paul Ammann, "Generating Test Data from State-based Specifications", ISE Department, George Mason University, USA, 2003.
- [3] Aynur Abdurazik and Jeff Offutt, "Generating Test Cases from UML Specifications", 1999.
- [4] Dirk Seifert, "Test Case Generation from UML State Machines", inria-00268864, version 2 - 23 Apr 2008.
- [5] M. Prasanna, K.R.Chandran, "Automatic Test Case Generation for UML Object Diagrams Using Genetic Algorithm", Int. J. Advance. Soft comput. Appl., vol.1, no. 1, July 2009, pp. 19-32.
- [6] Philip Samuel and Anju Teresa Joseph, "Test Sequence Generation from UML Sequence Diagrams", Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2008.
- [7] Supaporn Kansomkeat and Sanchai Rivepiboon, "Automated- Generating Test Case Using UML Statechart Diagrams ",SAICSIT 2003.
- [8] Ranjit Swain, Vikas Panthi, Prafulla Kumar Behera, Durga Prasad Mahapatra, "Automatic Test Case Generation Based on State Machine Diagram", International Journal of Computer Information Systems, vol.4, no.2, 2012, pp. 99-124.
- [9] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Working Papers 2006. Department of Computer Science, The University of Waikato (New Zealand), April 2006.
- [10] Stephan Weissleder, Simulated Satisfaction of Coverage Criteria on UML State Machines, Third International Conference on Software Testing, Verification and Validation, 2010.
- [11] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.