International Journal of Innovations in Engineering and Technology (IJIET)
http://dx.doi.org/10.21172/ijiet.82.020

Generation of Test Cases for Object Oriented Software using UML State Machine Diagram

Roopashri Shetty
Department of Computer Science and Engineering
Manipal Institute of Technology ,Manipal University , Manipal, Karnataka, India

Geetha M
Department of Computer Science and Engineering
Manipal Institute of Technology ,Manipal University , Manipal, Karnataka, India

Archana Praveen Kumar
Department of Computer Science and Engineering
Manipal Institute of Technology ,Manipal University , Manipal, Karnataka, India

Abstract- Software testing is one of the most important phases of software development life cycle. Software testing is being used by all the software developers to test the overall working of their software. Testing is one of the most important means to validate the correctness of systems. A UML state machine diagram is a model which represents the dynamic behaviour of the system. The aim of this paper is to fully automate the testing process of software by generating the test cases in the design phase where, the state machine diagram is modelled using standard Unified Modelling Language (UML) tool called StarUML which exports the state machine diagram to its internal XMI representation. This XMI file is then pre-processed to extract the relevant data and store the data into an immediate data structure. After traversing the data structure, a set of test path will be generated and for each test path, test case will be generated.

Keywords – software testing, state machine diagram, test cases

I. INTRODUCTION

Software testing is performed to verify the working of the software developed according to the requirements specified i.e. to identify the correctness, completeness and the quality of the software developed. It includes a set of activities conducted with the intent of finding errors in software so that the errors could be corrected before the product is released to the end users. In other words, software testing is a process which checks whether the actual results match the expected results and ensures that the software system is defect free.[10] The cost of testing are put at 50% of the overall project costs.[10] There are many efforts to decrease the cost of testing, e.g. by introducing automation. Modelbased testing is about using models as specifications. Model based testing allows deriving test suites automatically from system under test (SUT). : First the test model is usually quite small, easy to maintain and easy to understand. Second, the use of test models allows traceability from requirements to test cases. Third model-based testing can be used for testing after system development as well as test first approaches, i.e. early creation of formal test models help in finding faults and inconsistencies within the requirements. Fourth, test models can be used to automatically generate small or huge test suites that satisfy corresponding coverage criterion. The work is focused on automatic test case generation with UML state machine diagram and the coverage criteria applied to them.

A state machine diagram is a simple illustration of representing dynamic behaviour of the system. It shows different states of a system and the order in which they get executed [3]. The State Machine Diagram represents objects of a single class and tracks the different states of its objects through the system. UML State machine diagram shows different states of a system and the order in which these activities get executed.

Test case is a document normally prepared by the tester with the sequence of steps to test the behaviour of feature of the application. Test Case document consists of Test case ID, Test Case Name, Environment, Expected Results, Actual Results and Pass/Fail. [3] Test cases describe tests that need to be run on the program to verify that the

program runs as expected. Test cases are usually generated from the requirement or the code while the design is seldom concerned.

Hence, in the proposed approach, the process of test case generation for a state diagram has been fully automated by performing the method of converting the model into its internal representation i.e. XMI file. The proposed method also explains the algorithm which parses the XMI file and stores into its internal data structure. All the test path are generated and for each test path, a test case is generated. In this regard, the contributions of the proposed approach can be stated as a framework for complete automation of test case generation

The rest of the paper is organized as follows. Proposed methodology are explained in section II. Result Analysis are presented in section III. Concluding remarks are given in section IV.

II. PROPOSED METHODOLOGY

The proposed approach takes one of the design artifact i.e. UML State machine diagram as the input and produces set of efficient test cases as its final output.
StarUML is the UML tool is preferred for the proposed approach because it supports exporting the diagram to XMI/XML representation, simplifying the further processing.

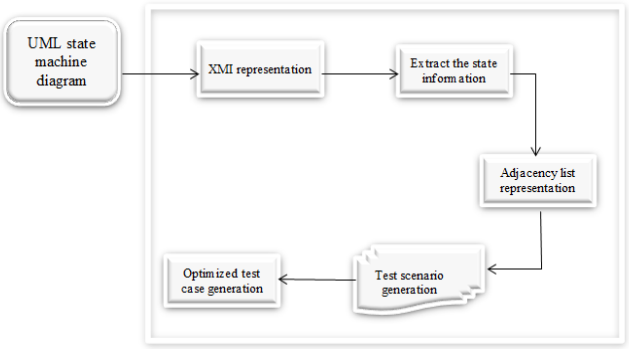

Figure 1: Proposed methodology for test case generation using a state diagram

Figure 1 gives the overview of the proposed methodology. In this, UML state machine is internally converted to XMI representation from which the state information is extracted and stored in the adjacency matrix. From the adjacency matrix, test paths and generated. On applying loop coverage criteria for the test path, the set of test cases are generated
Steps followed for the proposed methodology are:

Step 1: Input to the system is a UML State Machine Diagram, which is modelled using StarUML. A simple State Machine diagram is considered with few states and a looping construct. The diagram modelled using StarUML is saved in its default .mdl format.

Step 2: The state machine diagram in its .mdl format is converted to its corresponding XMI representation. The XMI file contains all the information about the state diagram. It contains the different states of the state diagram and all the attributes corresponding to a state.

Step 3: Pre-process/Parse the XMI file to extract the required data. At this stage a mini parser is designed using C# to parse the XMI file. Parsing is require because the XMI file generated will hold some redundant information about the different UML diagrams including the state diagram and it take more time to traverse the entire XMI file every time to generate the test cases. Hence it is easier to parse the file and extract only the relevant required information into an immediate data structure.

Pseudocode for parsing the XMI file is as follows:
If the state is a Psuedostate,

Then kind=initial
Compare the attributes of Psuedostate

Extract the name, id, kind, outgoing, and type attribute values and Store the attribute values.
Store the index of the state visited.

If the state is a Simplestate,
Then compare the attributes of Simple State
Extract and store the name, id, incoming, outgoing and type attribute values of the simple state
Keep track of the states visited by using the index for each state.

If the state is a Finalstate,
Then kind=final
Compare the attributes of Finalstate
Extract the name, id, kind, incoming, and type attribute vales and Store the attribute values.
Store the index of the state visited.

After extracting all the state information, bind this information to a gridview
Step 4: store the information into an immediate date structure i.e. adjacency matrix. Adjacency matrix is a way to define which states are adjacent to a given state. In an adjacency matrix, the neighbours of each state may be listed efficiently.
Pseudocode for the same is as follows:

```
For each State s1 in sList,Where s1.outgoing is not equal to null
    For each string outg in s1.outgoing
        For each State s2 in sList  Where s2.incoming is not equal to null
 For each string inc in s2.incoming
     If inc is equal to outg
                Connect the nodes s1 and s2 with   transition labelled outg.
                  Add it to the adjacency matrix as
AdjacencyMatrix[s1.index, s2.index] = transition;
```

Repeat the above for every pair of states to completely store the nodes in the adjacency list.

Step 5: Traverse the data structure by using test sequence generation techniques to generate the set of test scenarios. Since the information is stored in an adjacency list, traversing is easier. In order to generate all the possible test paths, test sequence generation technique is used. In test sequence generation, starting from the first node, traverse remaining nodes using DFS concept, in order to form the test sequence.

The steps used for the traversal are:

Step 1: Find the state whose kind is initial. This state will be the current state.
"Initial" is the name of the start node for every path
Step 2: Get all the child states of the current node and store it in a list called getstate.
Consider a state (nextstate) from the getstate list.
Separate the name value of that particular state.
Append this name to another list called path separated by "_".
Path = path + "_" + nextState.name;
Repeat this step2 for all the states till the nextstate.kind="final".
Step 3: Display all the paths in a gridview.

Step 6: Generate the test cases for each of these scenarios. Test scenarios are nothing but the test pattern used for evaluating the actual output of the system. Test scenarios will give the expected output of the system. During testing and validating the system, the actual output of the system is compared with the expected output of the system. If there is difference, then accordingly the system design has to be changed

III. RESULT ANALYSIS

The software is tested against the following case study "withdraw money from an ATM".

For withdrawing money from an ATM, the prerequisite is the user should have a valid ATM card. When the user goes near the ATM machine, he will switch on the system. Once the system is turned on, it will be in the idle state since no even has been generated. When the user inserts the ATM card, system will be active and it will check the validity of the card. If the card is not inserted properly then it will go back to idle state waiting for the user to reinsert the card. If the card is a valid one, then it will ask the user to enter the pin. After the user enters the pin, it will check for the validity of the pin. If the pin is incorrect, it will go the idle state prompting the user that the pin entered is wrong. If the pin is correct then it goes to the request state where the user can request for the withdrawal by specifying the amount to be withdrawn. After entering the amount to be withdrawn, it enters the processing state where it will check the account for the specified amount. If the amount is available, then the user can collect the cash. Otherwise the state will be changed to idle state prompting the user that no sufficient balance in the account. Once the user collects the cash, a final state will be reached.

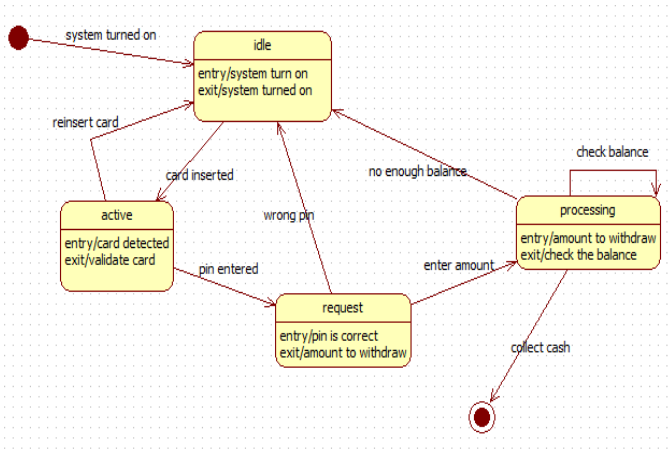

Fig 2: state diagram for "money withdrawal in ATM"

The state machine diagram modelled using starUML which is then exported internally to XMI file. A part of XMI file for this case study is given in Figure 3

<?xml version = "1.0" encoding = "UTF-8"?>
<XMI xmi.version = "1.1" xmlns: UML= "href://org.omg/UML/1.3" timestamp = "Tue May 21 15:6:40 2013">
<XMI.header>
<UML:CompositeState xmi.id="UMLCompositeState.5" name="TOP" visibility="public" isSpecification="false" stateMachine="UMLStateMachine.4" isConcurrent="false">
<UML:CompositeState.subvertex>
<UML:SimpleState xmi.id="UMLCompositeState.6" name= "idle" visibility="public" isSpecification ="false " container="UMLCompositeState.5" outgoing ="UMLTransition.20" incoming= "UMLTransition.21 UMLTransition.23 UMLTransition.25 UMLTransition.27">
<UML:Pseudostate xmi.id="UMLPseudostate.9" name="Initial1" visibility="public" isSpecification ="false" container="UMLCompositeState.5" outgoing="UMLTransition.25" kind="initial"/>
<UML:Transition xmi.id="UMLTransition.20" name="card inserted" visibility="public" isSpecification="false" stateMachine= "UMLStateMachine.4" source= "UMLCompositeState.6" target= "UMLCompositeState.10"/>

<UML:Transition xmi.id="UMLTransition.24" name= "collect cash" visibility="public" isSpecification= "false" stateMachine="UMLStateMachine.4" source="UMLCompositeState.13" target="UMLFinalState.16"/>
</XMI.header>

Fig 3: A part of XMI file for money withdrawl

XMI file shown in Figure 3 is parsed to get the set of all states and the state information as well as the set of transitions involved in the state diagram. Result of parsing the XMI file is shown in Fig 4

Fig 4: Result of parsing the XMI file for money withdrawal

The states and the transitions of the state machine diagram are represented in the form of an adjacency matrix so as to traverse the matrix and find all the test scenarios applying loop coverage criteria. Test scenarios are nothing but the paths traced.

Test scenarios generated for the above case study is shown in Table 1

Table 1: Test scenarios for the case study money withdrawal for ATM

Final Paths:

Item
Initial1_idle_active_request_processing_FinalState1
Initial1_idle_active_request_processing_processing_FinalState1
Initial1_idle_active_idle_active_request_processing_FinalState1
Initial1_idle_active_idle_active_request_processing_processing_FinalState1
Initial1_idle_active_request_idle_active_request_processing_FinalState1
Initial1_idle_active_request_idle_active_request_processing_processing_FinalState1
Initial1_idle_active_request_processing_idle_active_request_processing_FinalState1

The test cases are generated from the test scenarios. For the case study withdraw money from an ATM, there exists seven test cases mapped to seven test scenarios. Few sample Test cases for the above test scenarios are:

State Name	Test Input	Expected Output	Comments
Initial1			
idle	system turn on/	system turned on/	
active	card detected/	validate card/	
request	pin is correct/	amount to withdraw/	
processing	amount to withdraw/	check the balance/	
FinalState1			

Fig 5: Test case for the scenario "Initial1_idle_ active_ request _processing_FinalState1"

State Name	Test Input	Expected Output	Comments
Initial1			
idle	system turn on/	system turned on/	
active	card detected/	validate card/	
request	pin is correct/	amount to withdraw/	
processing	amount to withdraw/	check the balance/	
processing	amount to withdraw/	check the balance/	
FinalState1			

Fig 6: Test case for the scenario "Initial1_idle _active_request _processing_processing_FinalState1 "

IV.CONCLUSION

Automatic code generation from UML state machine diagram is constructed during the design process. Model based testing allows the test cases to be available early in the software development life cycle thereby making test planning more effective, saving time and resources.

In this paper, first, a state machine model of system under test is built. The XMI file is derived from state machine diagram. Then, the XMI file is parsed and all required information is extracted from the XMI file. Then, the test scenarios are generated by applying test sequence generation technique and loop coverage criteria. Lastly, a set of test cases are obtained for each test scenario. These test cases are used to compare the actual output of the system of the system with the expected output of the test case in the design phase itself. If there is any discrepancy in the actual output of the system, then the system design can be changed as and on required. In this way, this project introduces efficient test generation technique to optimize test coverage by minimizing time and cost.

REFERENCES

[1] "Test Case Generation Based on State and Activity Models" by Santosh Kumar Swaina Durga Prasad Mohapatrab Rajib Mallc

[2] Manuj Aggarwal , Sangeeta Sabharwal -Test case generation from state machine diagram-A survey- third international conference on ICCCT, November 2012.

[3] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. et al. Generating test data from state-based specifications- Software Test Verification and Reliability, 13:25– 53, 2003.

[4] J. Offutt and A. Abdurazik. - Generating tests from UML specifications, In Proceedings of the 2nd International Conference on UML, Lecture Notes in Computer Science, volume 1723, pages 416– 429, Fort Collins, TX, January 2001. Springer-Verlag GmbH.

[5] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams In Proceedings of SAICSIT, ACM, pages 296–300, 2003

[6] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from UML state diagrams. Software Testing Verification and Reliability, 146(4):187– 192, 1999.

[7] Stefania Gnesi, Diego Latella, and Mieke Massink: Formal test-case generation for UML statecharts. In ICECCS '04: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age, pages 75–84, Washington, DC, USA, 2004. IEEE Computer Society

[8] D. Deng, P. C.-Y. Sheu, T. Wang, A. K. Onoma. Model-based Testing and Maintenance. In: Proceedings of the IEEE Sixth International Symposium on Multimedia Software Engineering (ISMSE'04).

[9] Ammann, Paul E.; Black, Paul E.; Majurski, William: Using Model Checking to Generate Tests from Specifications. In: ICFEM'98: Proceedings of the Second IEEE InternationalConference on Formal Engineering Methods, p. 46. IEEE ComputerSociety, Washington, DC, USA, 1998. ISBN 0-8186-9198-0.

[10] Ammann, Paul; Offutt, Jeff: Introduction to Software Testing. Cambridge University Press, New York, NY, USA, 2008. ISBN 9780521880381.