Generation of Test Scenarios Using Activity Diagram

P. Nanda, D. P. Mohapatra and S. K. Swain

Abstract-- **Testing of software is a time-consuming activity which requires a great deal of planning and resources. In scenario-based testing, test scenarios are used for generating test cases, test drivers etc. UML is widely used to describe analysis and design specifications of software development. UML activity diagrams describe the realization of the operation in design phase and also support description of parallel activities and synchronization aspects involved in different activities perfectly. Therefore, test scenarios generated from activity diagrams will achieve test adequacy criteria perfectly. Handling parallel activities represented by fork-join pairs present in activity diagrams is also very difficult. Our approach generates test scenarios from UML activity diagrams, where the design is reused to avoid the cost of test model creation. This approach generates test scenarios from activity diagrams containing fork-join pairs and also handles the complicacy of the fork-join pairs. A prototype tool has been developed to support the approach.**

Index Terms--**Automated testing, Model-based testing, Scenarios, Scenario-based testing, Software testing, Synchronization, test scenarios, UML, UML-based testing, UML activity diagram.**

I. Introduction

SOFTWARE testing plays a crucial role in assuring software quality. One of the most important issues in the software testing research is the generation of the test cases. As the complexity and size of software grow, the time and effort required to do sufficient testing grow. Manual testing is time-consuming and error-prone. It is necessary to develop automatic testing techniques.

Test cases are commonly designed based on program source code. This makes test case generation difficult especially for testing at cluster levels. Test case generation from design documents has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. Another advantage of design-based testing is to test the compliance of the implementation with the design documentation. Test scenarios are used for generating test cases, test drivers etc. Manual generation of test scenarios is time consuming and laborious. Hence either automatic or semi-automatic generation of test scenarios is often desired. Test scenarios can be generated from the design models. UML is achieving a great attention as the industrial de-facto standard for modeling object-oriented software systems, in software testing. Along with the advantages there are also challenges for generating test cases from UML specification. UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe structural or behavioral aspects of systems. Activity diagrams also describe the sequence of activities among the objects involved in the control flow during implementation. It focuses on representing activities. test scenarios generated from activity diagrams will achieve test adequacy criteria perfectly

Scenario-based testing is a software testing activity that uses scenario tests, or simply scenarios, which are based on a hypothetical story to help a person think through a complex problem or system. They can be as simple as a diagram for a testing environment or they could be a description written in prose. These tests are usually different from test cases in that test cases are single steps and scenarios cover a number of steps. Scenarios are also useful to connect to documented software requirements, especially requirements modeled with use cases. Within the Rational Unified Process, a scenario is an instantiation of a use case (take a specific path through the model, assigning specific values to each variable). More complex tests are built up by designing a test that runs through a series of use cases. Scenario testing works best for complex transactions or events, for studying end-to-end delivery of the benefits of the program, for exploring how the program will work in the hands of an experienced user, and for developing more persuasive variations of bugs found using other approaches. Test suites and scenarios can be used in concern for complete system testing.

UML Activity Diagrams are commonly used to model business processes, basic control and data flow in software systems and they require little technical expertise to develop and understand. UML activity diagrams describe the sequential or concurrent control flows of activities. They can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. UML Activity Diagrams are used to model the logic captured by a single use case. The set of activity diagrams represents the overall behavior specified for the application and is the basis for testing the different functionalities and business rules described in the use cases specification.

P Nanda is with National Institute of Technology, Rourkela, 769008 . India . (e-mail: n.pragyan@gmail.com).

Dr. D. P. Mohapatra is with National Institute of Technology, Rourkela, 769008 . India . (e-mail: durga@nitrkl.ac.in).

S. K. Swain is with Kalinga Institute of Industrial Technology, Bhubaneswar, 751024 . India . (e-mail: SwainSantosh@yahoo.co.in).

This type of diagram is ideal for our purposes because we require a method to describe the test case flow. In the test generation phase, an activity's text will become a test step. As is allowed in UML, activities can include or be refined by other activity diagrams (i.e., enhance by refinement). In these cases, the test generation will "flatten" the diagrams. The step containing the activity text will be replaced with the steps generated from the refined or included diagram.

UML activity diagrams describe the realization of the operation in design phase perfectly. Activity diagrams also describe the sequence of activities among the objects involved in the control flow during implementation. It focuses on representing activities. Activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

The modeling elements consist of nodes and edges. The model includes action states, activity states, decisions, swim lanes, forks, joins, objects, signal senders and receivers. The edges represent the occurring sequence of activities, objects involving the activity, including control flows, message flows and signal flows. *Swim lanes* enable the activity diagram to group activities based on who is performing them. Swim lanes subdivide activities based on the responsibilities of some components. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. *Activity states* represent the performance of a step within the workflow. Activity states and action states are denoted with round cornered boxes. *Transitions* show what activity state follows after another. This type of transition is sometimes referred to as a completion transition, since it differs from a transition in that it does not require an explicit trigger event; it is triggered by the completion of the activity the activity state represents. Transitions are represented by arrows. *Decisions* are those for which a set of guard conditions are defined. These guard conditions control which transition of a set of alternative transitions that follows once the activity has been completed. You may also use the decision icon to show where the threads merge again. Decisions and guard conditions allow you to show alternative threads in the workflow of a business use case. Decisions are shown as diamonds with one incoming arrow and multiple exit arrows each labeled with a Boolean expression to be satisfied to choose the branch. *Synchronization bars*, are used to show parallel sub flows. Synchronization bars allow us to show concurrent threads in the workflow of a business use case. These are also known as *fork* and *join* pairs. Forks or joins are shown by multiple arrows entering or leaving a synchronization bar.

Figure 1 shows a UML activity diagram for an operation of withdrawing money from an ATM. In this diagram a_0 and a_9 are the initial and final states respectively. The states a_1 to a_8 are the activity states. The arrows t_0 to t_{15} are the transitions showing the flow between the activity states. Decisions are shown using diamonds and synchronization bars are used to show the fork and join pairs. In this example, we have considered a nested fork-join pair. The swim lanes are represented as ATM and BANK. These swim lanes show which activities are performed by whom.

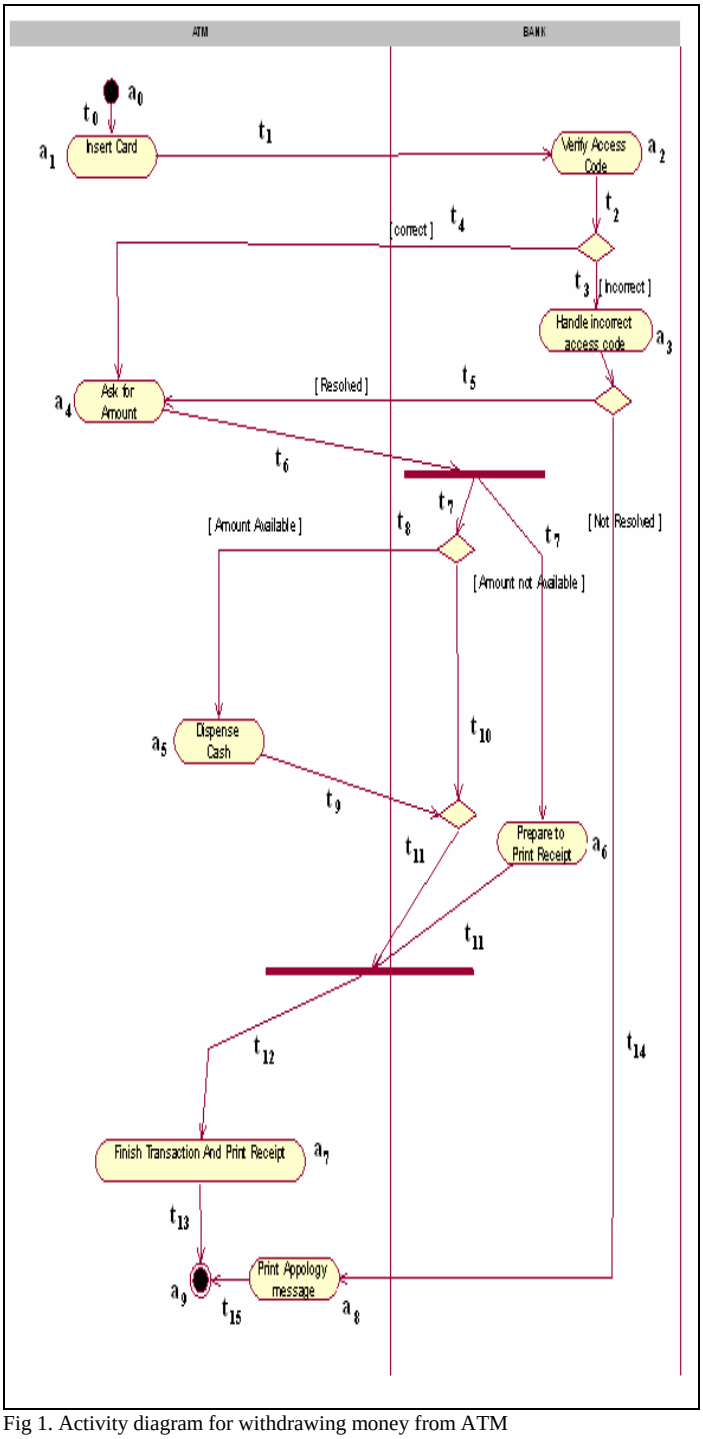

Fig 1. Activity diagram for withdrawing money from ATM

II. Generating Test Scenarios from Activity Diagram

Our approach parses the activity diagram and generates the test scenarios which satisfy the path coverage criteria. In order to traverse all the executing paths present in the activity diagram that satisfies the requirement specifications, the activity diagram is transformed into flowcharts and then it is traversed to achieve path coverage criteria. As activity diagrams represent the implementation of an operation like the flow chart of code implementation and an executing path is a possible execution trace of a thread of a program, the executing paths are derived directly from the activity diagrams. We have considered path coverage in our approach,

since it has the highest priority among all the coverage criteria for testing. Our approach also handles the complicacy of nested fork joins. Moreover, in our approach we have used a priority criterion that checks whether the target activity state of a transition is a fork or an activity state. If the target of the transition is a fork, then the fork has higher priority over the activity state. So it should be considered first and then only the other path is considered.

As a result of this priority criterion the complicated nested fork-join pair is handled properly in our approach. In the following, we describe the steps required for test scenario generation.

A. Steps for test scenario generation

***Step 1*:** First we construct the activity diagram for the given problem. Then in order to generate test scenarios from activity diagram, our approach traverses the activity diagram using depth first search (DFS) method.

***Step 2*:** Then, to traverse the activity diagram from initial state to final state, our approach visits all the current activity states and the corresponding transitions released from the current activity state.

***Step 3*:** Next, a record of the trace of a run of the activity diagram is maintained by recording the visiting trace of the current activity states and transitions. The current activity state is recorded using a stack and its number of occurrence (in the stack) is recorded by setting a flag array.

***Step 4*:** When the current state is not empty, one of the possible transitions is fired and its occurrence is set in the flag array and the transition is then deleted from the possible transition lists.

***Step 5*:** Then the transition and its corresponding guard condition are pushed into the stack. The current activity state is then incremented and set to point the next possible activity state.

***Step 6*:** Each loop present in the activity diagram is executed at most once covering the corresponding activity states and transitions. A loop is bypassed in the sequence if it is already considered earlier. After entering a fork-join pair, our approach checks whether the priority criterion is considered or not. Then, the next current activity state is visited according to this priority criterion.

***Step 7*:** This process continues till the current activity state is empty i.e. no more transitions are present. In other words, this process is continued untill a full path is completed.

***Step 8*:** The test scenario is then read out from the bottom of the stack.

***Step 9*:** Then our approach checks the activity diagram for the last visited current activity state where an unvisited transition is present and repeats the whole process from that current activity state.

***Step 10*:** The process continues until all the activity states and the corresponding transitions present in the activity diagram occur at least once in the set of test scenarios.

B. Result

By implementing the above steps, we have generated a set of test scenarios for the problem represented in the activity diagram shown in fig. 1.

One of the test scenarios is as follows:

TS: (a_0) t_0 (a_1) t_1 (a_2) t2 [incorrect] t_3 (a_3) [resolved] t_5 (a_4) t_6t_7 [amount available] t_8 (a_5) t_9 t_{11} t_7 (a_6) t_{11} t_{12} (a_7) t_{13} (a_9).

This result is shown in fig. 2.

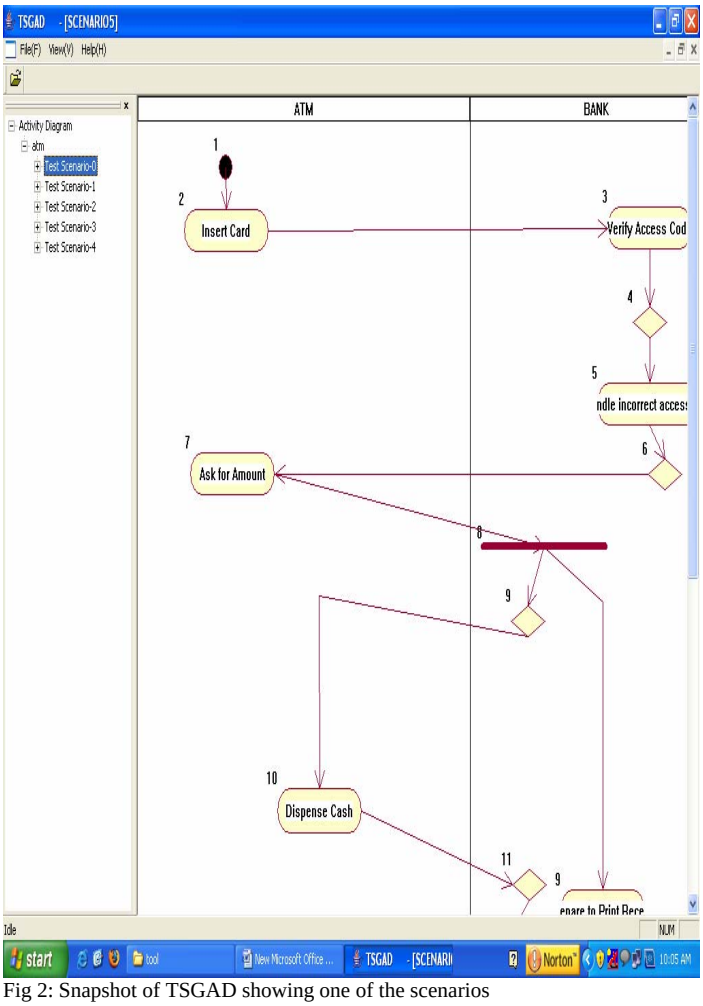

Fig 2: Snapshot of TSGAD showing one of the scenarios

C. TSGAD: A prototype tool

To support the above proposed scheme, we have developed a tool named Test Scenario Generator for Activity Diagram (TSGAD), which generates test scenarios from activity diagrams. The activity diagram can be developed using any UML tool such as Rational Rose or Magic Draw etc. We have used Rational Rose to construct the activity diagram. Then it can be exported as a XMI file. The GUI was developed using the swing component of Java. The GUI screen along with a sample test scenario is shown in Fig. 2. The UML model parser present in TSGAD is abided by the XMI specification of OMG. It imports activity diagrams easily and analyzes the Rational Rose MDL file with the help of Rose Extensibility Interface. Next, the model parser extracts the activity states and the corresponding transition information and stores them in an intermediate data structure such as *tree* which can be accessed later by the scheme for generating test scenarios. The tool analyzes the semantics of the result of the model parser, and derives test scenarios using the proposed scheme that satisfies the path coverage criteria.

III. Related works

Canevet et al. [5] present a method of analyzing the newly revised UML2.0 activity diagrams. Their analysis method builds on a formal interpretation of these diagrams with respect to the UML2.0 standard. Their analysis approach is exercised on a substantial example of modeling a multiplayer distributed role-playing game. Mingsong et al. [6] used UML activity diagrams as design specifications, and present an automatic test case generation approach. The approach first randomly generates abundant test cases for a JAVA program under testing. Then, by running the program with the generated test cases, we can get the corresponding program execution traces. Last, by comparing these traces with the given activity diagram according to the specific coverage criteria, we can get a reduced test case set which meets the test adequacy criteria. The approach is also used to check the consistency between the program execution traces and the behavior of UML activity diagrams. Linzhang et al. [7] proposed an approach to generate test cases directly from UML activity diagram using gray-box method, where the design is reused to avoid the cost of test model creation. In this approach, test scenarios are directly derived from the activity diagram modeling an operation. Then all the information for test case generation, i.e. input/output sequence and parameters, the constraint conditions and expected object method sequence, is extracted from each test scenario. At last, the possible values of all the input/output parameters could be generated by applying category-partition method, and test suite could be systematically generated to find the inconsistency between the implementation and the design. Chandler et al. [8] introduced an approach that will capture, store and output usage scenarios derived automatically from UML activity diagrams. In this paper they presented an approach, dubbed AD2US, which automatically extracts USs from ADs; thereby extending the time available for other activities such as test-case generation or the verification of consistency between ADs, use cases and usage scenarios. They have defined UCs by textually describing them using a template to ensure that all possible scenarios and usage interactions are defined. As per our knowledge although lots of works has already been done using activity diagram yet no one has discussed the fork-join complicacy in detail. In our approach we have considered the fork-join complicacy.

IV. Future work and conclusion

Testing of software is a time-consuming activity requiring a great deal of planning and resources. For successful automated testing, support must come from both processes and tools. Our approach generates test scenarios directly from UML activity diagram, where the design is reused. One of the advantages of our approach is that, reuse of the design model for generating test scenarios reduces the cost of building test models or transforming models. By using our approach defects in the design model can be detected during the analysis of the model itself. So, the defects can be removed as early as possible, thus reducing the cost of defect removal. Another advantage of our approach is that it handles the complicacy of nested fork-join pair which is more often overlooked by other approaches.

Our future work involves developing a testing approach that supports developers in their task of creating automated functional test drivers for object-oriented software on a compressed schedule. The objective here is to support the derivation of functional system test requirements, which will be transformed into test cases, test oracles, and test drivers once we obtain the detailed design information. The ultimate goal will be to address testability, coverage criteria and automation issues, in order to fully support system testing activities.

V. References

Periodicals:

[1] C. Meng, L. Xuan-dong, Z. Guo-liang, "Formal Analysis on UML Real-time Activity Diagram", *Chinese Journal of Computers*, vol. 3,2004.

[2] L. Min, J. Maozhong, L. Chao, "Design of Testing Scenario Generation Based on UML Activity Diagram", *The Engineering and Application of Computer*, Vol. 12,pp.122 124, 2001.

Books:

[3] R. S. Pressman, Software Engineering- A Practitioners Approach, 4th ed, New York: McGraw-Hill, 1964, pp. 644-673.

[4]

Technical Reports:

[5] U2P. Unified Modeling Language: Superstructure version 2.0,April 2003. Available from http://www.omg.org/uml/as ad/03-04-01.

[6]

Papers from Conference Proceedings (Published):

[7] C. Canevet, S. Gilmore, J. Hillston, L. Kloul and P. Stevens, "Analyzing UML 2.0 activity diagrams in the software performance engineering process", in *Proc of WOSP'04* January 14-16, 2004, Redwood City, California.

[8] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic Test Case Generation for UML Activity Diagrams", *AST'06*, May 23, 2006, Shanghai, China.

[9] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng "Generating Test Cases from UML Activity Diagram based on Gray-Box Method", Proc. APSEC'04, 2004.

[10] R. Chandler, C. P. Lam, H. Li, "AD2US: An Automated Approach to Generating Usage Scenarios from UML Activity Diagrams", *Proc. of 12th APSEC'05, 2005* .

[11] H. Li, C. P. Lam, "Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams", *Proc. TESTCOM 2005, LNCS 3502, Montreal, 2005* .

[12] D. Xu, C. P. Lam, H. Li, "Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams", *Proc. of 12th APSEC'05, IEEE Computer Society Press, Taipei, 2005* .

[13] W. T. Tsai, A. Saimi, L. Yu, R. Paul, "Scenario-based Object-Oriented Testing Framework", *Proceedings of the Third International Conference On Quality Software (QSIC'03).*

[14] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, J. Kazmeier, "Automation of GUI Testing Using a Model-driven Approach", *AST'06, May 23, 2006, Shanghai, China.*

[15] Hartmann, J., Imoberdof, C., Meisenger, M., "UML-Based Integration Testing", in *ISSTA 2000 conference proceeding, Portland, Oregon, 22-25 August 2000, pp. 60-70*

[16] Y.Wu, M. Chen and J. Offutt, "UML-based Integration Testing for Component-based Software", *The 2nd International Conference on COTS-Based Software Systems (ICCBSS). pages 251-260, Ottawa, Canada, February 2003.*

VI. BIOGRAPHIES

Pragyan Nanda was born on May 8, 1981. She graduated from the Jagannath Institue of Technology, Cuttack, and persuing her M.Tech at National Institute of Technology, Rourkela, India. Her employment experience includes Lecturer in Mahavir Institute of Technology, Bhubaneswar. Her special fields of interest included software engineering, UML-based testing, Object Oriented Testing.

Durga Prasad Mohapatra studied his M.Tech at National Institute of Technology, Rourkela, India. He has received his Ph. D from Indian Institute of Technology, Kharagpur., India His employment experience includes being Assistant Professor at National Institute of Technology, Rourkela. His special fields of interest include Software Engineering, Discrete Mathematical Structure, slicing Object-Oriented Programming and distributed computing.

Santosh Kumar Swain was born on : 14th May, 1967. He graduated from O. U. A. T , Bhubaneswar, he studied his M.Tech at Utkal University , Bhubaneswar. He is Pursuing PhD in Utkal University Bhubneswar. His employment experience includes Asst. Prof. & Coordinator M. Tech, Department of Computer Sc. & Engineering, Kalinga Institute of Industrial Technology, Bhubaneswar. His special fields of interest includes software engineering, UML-based testing, Object Oriented Testing, slicing.