International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 17 (2018) pp. 13178-13187

Generation of Test Cases from Behavior Model in UML

Priya Kamath B.[1], Narendra V.G.[2]

[1]*Assistant Professor-Senior Scale, Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India.*

[2]*Associate Professor-Senior Scale, Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India. Corresponding author*

Abstract

Software Testing is one among the major phases of Software Development Life Cycle (SDLC). Testing is a process carried out to ensure that the developed system is working as per the given requirements and is error-free [1] Test case generation is a major step in software testing. Software can be tested at any phase of SDLC. Earlier the system is tested, it becomes easier to reveal the faults and there by greatly reducing the cost and time required to develop the system. This paper aims at generating test cases from artefacts developed from design phase of the Software Development Life Cycle. UML Activity Diagram is selected as the design artefact to generate the test cases. The proposed approach generates optimized set of paths from the Activity Diagram. The main idea behind this is the efficiency of the testing process. Generating all possible test cases and then optimizing them is impractical. It will unnecessary create either invalid test cases or redundant test cases and eventually makes the entire process of test case generation inefficient. This approach mainly aims at improving the efficiency of the system by automating the whole process and by generating only the optimized set of test cases

Keywords: Model, Unified Modelling Language (UML), Activity Diagram, XML Metadata Interchange (XMI).

INTRODUCTION

Testing plays a major role in the development of a system. The testing process verifies whether the designed software works as per the user requirements. The testing process ingests nearly 80 percent of the cost of the software development. Testing can be done during any phase of the software development life cycle. The usual practice is to test the system after the implementation phase [1] In case of any defects at this phase, the entire system needs to be corrected to fix the defect. Therefore this process will increase the cost involved for the system development. This paper mainly aims at model based testing. In model based testing [14], design models are used to generate the test cases. With model based testing, it helps us to identify the faults at the early stages of the software development life cycle there by reducing the cost and time involved in the system development. Model is a graphical representation of the system functionality. In this paper, UML Activity Diagram is used as the design model to generate the test cases. UML is a Unified Modelling Language used to model the systems. UML has different notations and symbols used for representing different system functions and components.

The main purpose of this work is to analyze and automatically generate the effective set of test cases from a given activity diagram. A XMI parser is used to process the XML file being obtained from the UML activity diagram and maps it to its corresponding adjacency list. Then the test scenarios are generated by applying test data generation technique or by list traversal using AI techniques [21, 22].

The existing approaches provide a semi-automated way of test case generation from an UML activity diagram. That is, the given diagram is translated to some intermediate representation such as activity dependency graph, data flow graph, control flow graph or the activity dependency table. This intermediate representation is then given as input to the further phases. Even though it seems to be fully automated, the procedure for converting the diagram to its internal representation has not been addressed clearly in any of the existing approaches. The output of the system is optimized set of test suites and test cases. Without optimization the system would have generated all possible test suites out of which there could be even redundant or irrelevant test suites. Some test suites will have higher priority than others. And in case of Fork-Join activities the order of interleaving has to be maintained. Hence by taking care of these concepts, a Fork-Join coverage criterion has been applied to maintain the order of concurrent activities. The redundant paths are optimized using a Weighted Prioritized and Fork-Join coverage technique.

BACKGROUND

Activity Diagram is one of the behavioural diagrams in UML which describes the flow of activities in the system. Each activity can be further subdivided into smaller activities or it can be an atomic activity. The syntax of Activity Diagram is

very similar to that of flowcharts. Activity diagram mainly describes the way in which each functionality is executed in the system. It never explains "what the system does" rather explains "how the system does". Activity Diagram uses variety of symbols to represent different scenarios. Every diagram will have one initial state and can have more than one final states. The start state in an Activity Diagram is denoted using a filled circle. Start state of an activity diagram will have no incoming edges. Activities are denoted using lozenge shape. If-Else scenario can be easily modelled by considering a decision construct which contains branches to represent true/false conditions. Diamond shaped symbol is used for decision making. UML also provides a mechanism to model concurrent activities of the system. When an activity needs to be run concurrently, we use Fork construct to model it. Fork is represented using horizontal thick line. Fork node contains one incoming edge which is split into multiple outgoing edges. The activities which are forked will execute concurrently. All the activities that are forked are joined later using join node. Join node is always used along with a Fork node. For an activity to occur in parallel it has to first fork and later once all the parallel activities are executed, they need to be joined together before starting with the next activity. End state of Activity Diagram is denoted using filled circle with an outer circle. End state will have no outgoing edges [2].

Fig. 1 shows a sample example of UAD consisting of seven activities. The various activities involved are A1, A2, A3, A4, A5, A6 and A7. Every Activity diagram contains only one start state and any number of end states Start state in the Activity Diagram is denoted using filled circle. End state is shown using a filled circle with an outer circle .In the above example, after completion of activity A1, there is a fork node represented using thick horizontal line. The activities following fork node namely A2, A3, A4, A5 and A6 execute concurrently. The order of execution needs to be maintained among forked activities i.e. an activity can execute only when the execution of previous activity is complete. The If-Else construct is modelled using diamond symbol. Every decision node in an Activity diagram will have one incoming edge and can have two or more number of leaving edges. The decision symbol in Fig. 1 has two leaving transitions and each of them has a label referred as the guard expression [11, 15]. In Fig. 1 after executing the activity 'A2', if the condition of the branch node is evaluated to true, then 'A5' will be executed next, else the control moves on to activity 'A4'. Merge nodes are also depicted in the same way as decision nodes but it acts as a source node to combine multiple flows. As per the example, activity A7 can begin only when all activities which were forked previously has completed their execution. A transition which is depicted as directed edge, describes the order of execution or control-flow between two activities. Consider two activities A and B, if there is a transition from A to B it means, Activity A has to be completed before Activity B starts.

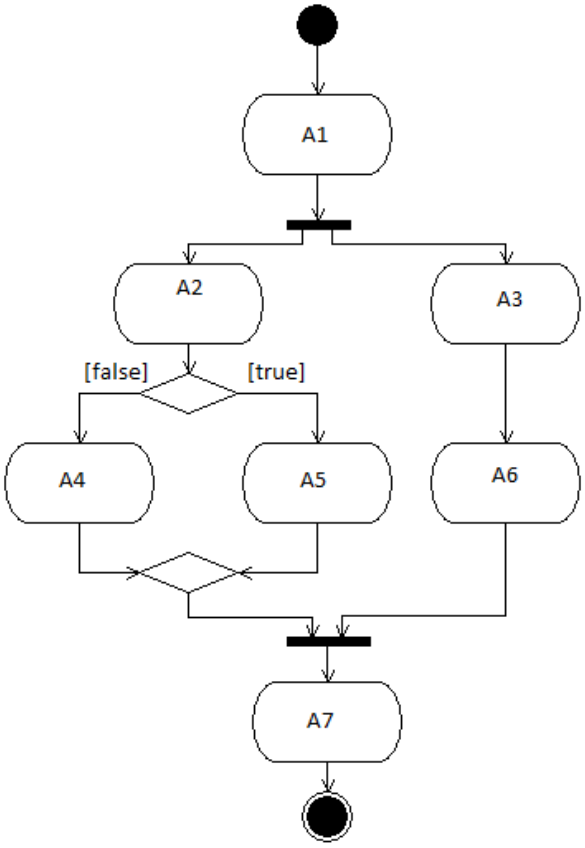


Figure 1. Sample Activity Diagram

LITERATURE SURVEY

Literature survey has been presented in table 1 to highlight the work that has been covered in this field earlier and also give the drawbacks of each work.

Table 1: Survey of previous work

Authors	Description of method	Drawbacks identified
Debasis Kundu and Debasis Samanta [6]	Activity diagram is first converted in to an Activity Graph using which the test cases are generated.	The paper has not clearly addressed the method to handle the fork-join constructs present in activity diagram.
A.Nayak and Debasis Samanta [4]	Based on the classification of control constructs followed by a transformation approach which takes into account any combination of nested structures and transforms an activity diagram into a model called Intermediate Testable Model (ITM). ITM was used to generate test	-

Authors	Description of method	Drawbacks identified
	scenarios	
Mitrabinda Ray et. al. [3]	This approach builds a flow dependence graph from UML activity diagram and applies conditioned slicing on a predicate node of the graph, in order to generate test cases.	Satisfies only path coverage criteria. Does not deal with activity and transition coverage.
Namita Khurana and R.S. Chillar	Generates test cases using sequence and state chart diagrams. The generated test cases are later optimized using genetic algorithm.	Some of the problem related to combination of Sequence and State Chart remain unresolved. Also the proposed approach is not yet automated.
Ajay Kumar Jena et al. [7]	Proposes an approach to generate test cases from UML 2.0 activity diagrams by covering maximum activity nodes in the diagram.	The approach is not automated. So an automated tool can be developed for the proposed approach.
Meiliana et al. [8]	The proposed approach modifies existing DFS algorithm to generate the test cases from design artifacts.	Contains redundant test cases.
Liping Lia et al. [9]	This paper combines software testing with Extenices and suggests an automatic approach for generating decreased set of test cases from UML Activity diagrams partly using Extension Theory.	Does not provide an efficient technique to handle fork and join constructs of an Activity Diagram.
Linzhang et. al [10]	In this approach, UML activity diagram is used to derive the test scenarios. Later, the information required for the generation of test cases is extracted from each test scenario and the possible values of all the input/output parameters could be generated by applying category-partition method.	Fails to handle the fork-join constructs present in activity diagram.
Chandler et al. [11]	The proposed architecture creates a special table called Activity Dependency Table (ADT) for each XML file using which it automatically generates a directed graph called Activity Dependency Graph (ADG) that is used in conjunction with the ADT to extract all the possible final test cases.	Does not handle the coverage criterions and fork join constructs of Activity Diagram.
Sangeeta Sabharwal et al. [18]	Proposes a technique to prioritize test case scenarios by identifying the critical path clusters using genetic algorithm. The test case scenarios are derived from the UML activity diagram and state chart diagram.	Simple fork-join is considered.
Sapna P.G and Hrushikesha Mohanty [19]	Proposes a prioritization technique based on UML activity diagrams. The constructs of an activity diagram are used to prioritize scenarios.	The approach is semi-automated.
Kim et al. [16,17]	Proposed an approach to generate test cases from activity diagrams by exposing all possible external inputs and outputs. It clearly explains the concept of test data generation at run time.	The techniques for handling loops have not been addressed in this paper.
Kanjanee Pechtanun et al.	In this approach, the given activity diagram is first converted into a grammar called as Activity Convert (AC) grammar. The generated test cases are then validated against the coverage analysis.	Redundant test cases are generated.
Yoo-Min Choi and Dong-Jin Lim [25]	The paper mainly aims at producing transition paths considering the dependent transition pairs.	It does not consider other UML diagrams.
Pradeep Kumar Arora and Rajesh Bhatia [24]	The paper aims at automatic generation of test cases from collaboration diagrams and confirms the result using mutation testing.	Not very feasible.
Xinying wang et. al. [26]	This paper presents an approach to prioritize the test scenarios using Genetic Algorithm along with Particle Swarm Optimization.	Not an automated approach.
Haiying Sun et. al[27]	A technique is proposed to generate test cases using mutated activity diagrams with the intention of finding defects.	Test generation method is demonstrated only on code testing.

METHODOLOGY

Fig. 2, shows an overview of the proposed methodology. In the first step, activity diagram is designed using standard UML tool called StarUML. StarUML provides a workspace for the user to model different views of the system using Unified Modelling Language. It is not possible to perform any operations using this diagram, until and unless we have its internal representation. Hence the Activity diagram needs to be pre-processed. To achieve this, in the next step the diagram is exported to its corresponding XMI representation using StarUML. UML Activity diagram in its XML format consists of various XML tags and its attributes which are essential to produce the desired result. XMI file contains details about each of the activity and transitions in the activity diagram. Once the relevant details are obtained, the next step is to store this information into intermediate data structure (i.e. Adjacency list) for further processing. Adjacency list is traversed using AI techniques to obtain the optimized set of test scenarios. In the last step, test cases are generated from each of the scenario.

Several UML tools are available to model UML diagrams. Some of the widely used tools are MagicDraw, RationalRose, StarUML [12] etc. MagicDraw is a good suite for UML modelling. The tool manages a true model representation which it stores as one large XML flat file. The main reason for not using MagicDraw in the proposed approach is, the XML file generated by the tool for the diagram is very large and takes longer time for processing. RationalRose [13] is an object-oriented Unified Modelling Language (UML) software design tool intended for visual modelling and component construction of enterprise-level software applications. Rational Rose allows designers to take advantage of iterative development because the new application can be created in stages with the output of one iteration becoming the input to the next. The RationalRose tool doesn't support option for exporting the diagram to its XMI/XML representation. Hence further processing of the diagram cannot be done using RationalRose. Both the tools mentioned above are proprietary software. StarUML is an open source tool which overcomes the above mentioned limitations and hence is preferred for the proposed approach.

Proposed methodology consists of following steps:

Step 1: Input to the system is an Activity diagram, which is modelled using StarUML tool. The Activity diagram considered for this project supports any number of connected activities, branching constructs and set of concurrent activities. By default, the diagram modelled using StarUML is saved in its .mdl format.

Step 2: Activity diagram in its .mdl format is converted to its XMI representation. XMI is a standard used to store UML models.

Step 3: The XMI file obtained in Step 2 is parsed to extract the required data. At this stage an algorithm is designed to parse the XMI file. Algorithm 1 explains the method used to extract the data from XMI file.

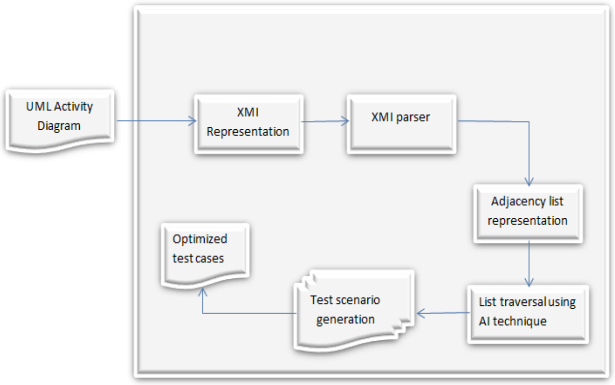


Figure 2. Overview of the proposed methodology

Algorithm 1 XMI Parser

```
procedure parseXMIStates( XMI file)
  Load the XML document
  if swim lanes are present then
     Check for the swim lanes in the diagram.
     Create a new list to hold all the swimlanes
     Add each activity to the swimlane list.
else
Add elements of compositeState.subvertex tag to sta
for each element in sta.elements
    Retrieve elements name, id and kind.
      if incoming != NULL then
           Split the incoming edges based on ' '
             else if outgoing != NULL then
                 Split the outgoing edges based on ' '
              end if
       end if
   end for
  end procedure
```

Step 4: The XMI generated in Step 3 contains information about States and Transitions of Activity Diagram. The above algorithm makes use of two separate functions namely, parseXMLStates and parseXMLTransitions to parse states and its respective transitions. If the Activity diagram is modelled using Swimlanes, then the swimlane details will be enclosed within ActivityGraph.Partition element. The Activity state information is present within the CompositeState.subvertex element. Hence the file is traversed to get the descendants of CompositeState.subvertex element. In the first step, for each of the elements in CompositeState.subvertex, a state object is created, where all the attributes of state are initialized. For each of the element extract its id, name and kind. In the next step, the incoming transition is checked. If there is more than

one incoming transition in the Activity diagram then it is separated by a space in the XMI file which is later split based on space and stored in an array. The same procedure is then repeated for outgoing transitions. More than one outgoing transitions are split based on space and is stored in another separate array. In the third step, all the states of Activity diagram are added to the stateList. Once all the states of the Activity diagram have been parsed, next step is to parse the transitions of the diagram. Transition details are contained in UML:StateMachine.transition.

Step 5: Data retrieved in step 4 is populated into a data structure for further processing. For convenience, Adjacency matrix is of size NXN is used; where 'N' refers to number of activities in an Activity diagram. The entry A [i,j] in Adjacency matrix is set to transition name when there is a transition from Activity i to Activity j. The main reason for choosing adjacency list as a data structure is- each node in the list can be easily traversed to generate the set of test suites. Or every element in the above mentioned tag, a transition object is created. If the transition has any guard expression then the guard expressions attributes needs to be extracted. Once all the transitions of Activity diagram has been parsed, the transitions are added to transList.

Consider the Activity diagram in Fig. 3. The adjacency list representation for the same is shown in Fig. 4

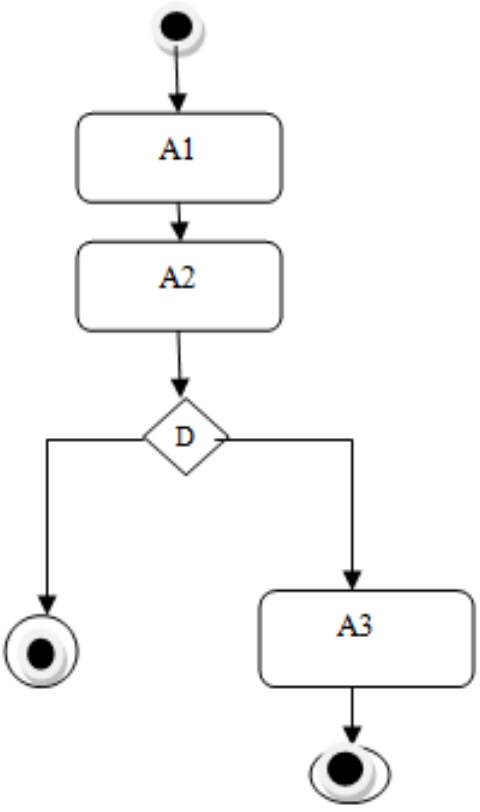


Figure 3. Example Activity Diagram

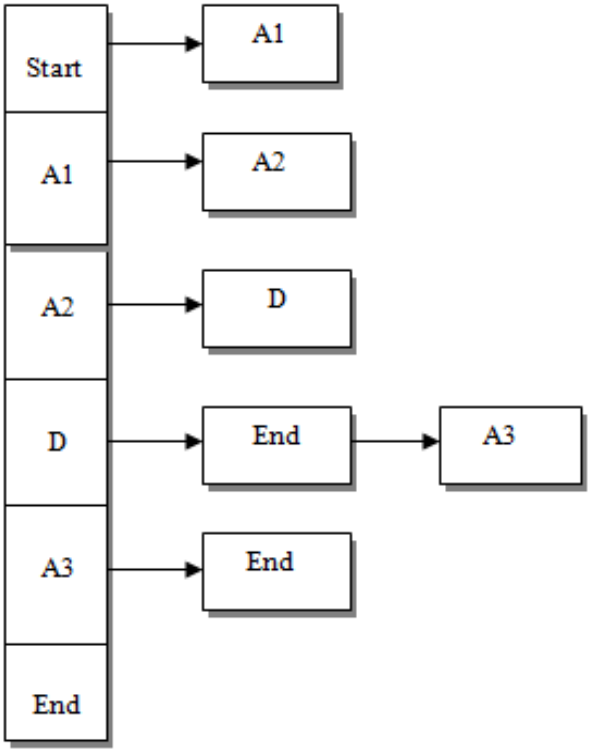


Figure 4. Adjacency list representation

Step 5: Once the data has been populated in Adjacency list, the list is traversed using AI technique called as Best First Search method (BFS). The BFS method is not used as it is; instead it is modified as per the need of the project. Best-first search in its most general form is a simple heuristic search algorithm. "Heuristic" here refers to a general problem-solving rule or set of rules that do not guarantee the best solution or even any solution, but serves as a useful guide for problem-solving. Best-first search is a graph-based search algorithm meaning that the search space can be represented as a series of nodes connected by paths.

The main objective here is to traverse the list and generate all possible paths from a Start activity node to a Final Activity node. The modified heuristic approach makes use of two lists namely Open List and Closed List. Open list is a list which contains the path without the Final node in it. Closed list is a list which contains the path with Final state. Every time a node is visited, it is first en-queued to the open path list. The last node in the list is then checked to see it it's a final node. If it is a final node, then the whole path will be dequeued and will be added to Closed Path list. In other words, every closed path has been an open path in its lifetime. The same is explained using a pseudocode in Algorithm 2.

Algorithm 2: List_Traverse

Enqueue the start node in the openpaths Queue.

Repeat

Dequeue a path from the openpaths queue

until openpaths queue is not empty

for each connected node n

newpath← oldpath+n

if n is a final state then

enqueue the newpath to closedpaths queue

else

enqueue the newpath to openpaths queue

end if

end for

Path traversal in case of concurrent activities enclosed within fork-join pair. Path traversal in case of fork-join pair is a complex task. The activities within a fork-join pair can execute in an interleaved manner but, only condition is - an activity cannot start its execution until and unless the activity before it has completed its execution. As the number of activities within a fork-join pair increases; it results in path explosion and hence, is infeasible to consider all paths due to limited capability of resource and time. Most of the existing techniques generate all possible paths and then select the correct paths from the set of generated paths thereby eliminating the redundant and irrelevant paths. This will simply waste lot of time by generating irrelevant paths. Hence an attempt is made to reduce to the time involved in generating the possible paths. This concept is illustrated by considering few examples of activity diagram with fork-join constructs.

Step 6: Test case optimization -Test adequacy criteria are nothing but an essential and important predicate, which shows the adequacy. In other words it is a stopping condition to stop the testing process, when all the defined criteria have been met. Path coverage, transition coverage and activity coverage are few test adequacy criteria, used for the work.

Algorithm 3 Coverage Calculation

for each path in Final set of paths

Create a bew path object p

Count total no. of states in the every path

Count transitions in each path

Assign weight for each of the path

Weight=Weight + (inCounter*outCounter)

ACoverage = $\frac{noOfStatesCovered}{totalStates} * 100$

Tcoverage= $\frac{noOfTransitionsCovered}{totalTransitions} * 100$

end For

Algorithm 4 Fork-Join traversal

if node.kind == fork then

Extract the corresponding join node

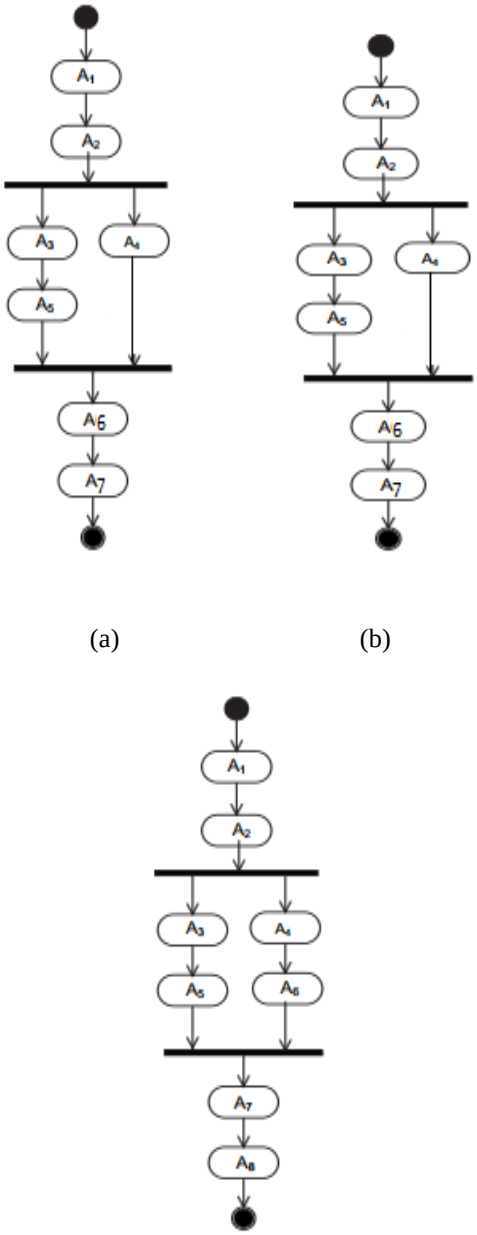
FJTraversal (Start, End, Path)

Generate all permutations

Remove the irrelevant permutations

end if

Algorithm 4 is applied to different activity diagram examples given in Fig. 5



(a) (b)

(c)

Figure 5 (a) Two activities (b) Three activities (c) Four activities

The algorithm 4 is applied to various activity diagrams given under Fig. 5 and following paths are generated.

Start→ A1→A2→A3→A4→A5→A6→End
Start→ A1→A2→A4→A3→A5→A6→End

Start→ A1→A2→A3→A4→A5→A6→A7→End
Start→ A1→A2→A3→A5→A4→A6→A7→End
Start→ A1→A2→A4→A3→A5→A6→A7→End

Start→ A1→A2→A3→A4→A5→A6→ A7→A8→End
Start→ A1→A2→A4→A3→A5→A6→ A7→A8→End
Start→ A1→A2→A3→A5→A4→A6→ A7→A8→End
Start→ A1→A2→A4→A6→A3→A5→ A7→A8→End
Start→ A1→A2→A3→A4→A6→A5→ A7→A8→End

RESULT ANALYSIS

The proposed methodology is explained with a sample case study of ATM management system. The Automated Teller Machine will serve one customer at a time. A customer needs to first insert an ATM card and enter the four digit PIN code. Once the PIN is entered, the system will verify if the entered PIN is valid or invalid. If it is valid one, then it will proceed to the next step. If the entered PIN is an invalid one, then it will allow the user to enter the PIN again. For the sake of simplicity, the number of trails on an invalid input is set to one. (i.e. if the entered input is invalid, user will be given only once chance to re-enter the data). The ATM Management System supports. Once the user entered PIN is valid, in the next step system will ask the user to enter the choice of option. User can select one among the following choices:

- Withdrawal
- Balance Inquiry
- Mini Statement of the previous transactions

When the user clicks on Withdrawal Option, the system will prompt the user to enter the amount to be withdrawn. Once the user enters, the amount entered will be verified to check if it is less than 500. If yes, the system will give a message saying insufficient balance and then asks the user to re-enter amount. Later, the amount value will be deducted from users account and cash will be dispensed to the user along with the receipt. When the user clicks on Balance Enquiry, the system will ask the user to confirm if he needs a printed receipt or no. If the user selects yes, then the system will output the balance details on a receipt and exits. If user selects no, then the system will output balance details on the screen and then exits. Mini Statement feature helps the user to know his past ten transaction details prior to current transaction. The Use case diagram for above mentioned case study is shown in Fig. 6 and Activity diagram for *Withdraw Money* use case is shown in Fig. 7.

Use Case diagram is one of the behavioral diagrams in UML. It is used to represent various users of the system and its various functionalities. In this ATM management system, there are two users namely Customer and Admin. Customer can withdraw money, enquire the balance or get the previous transaction details. Admin is the one who updates the ATM machine by periodically filling cash into it. The proposed methodology is tested on two functionalities namely Withdraw Money and Enquire Balance of the system. As a first step, Activity diagram is modelled for these two functionalities.

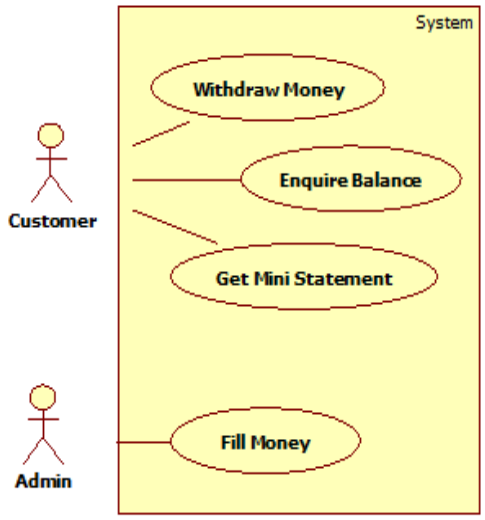


Figure 6. Use Case Diagram for ATM case study

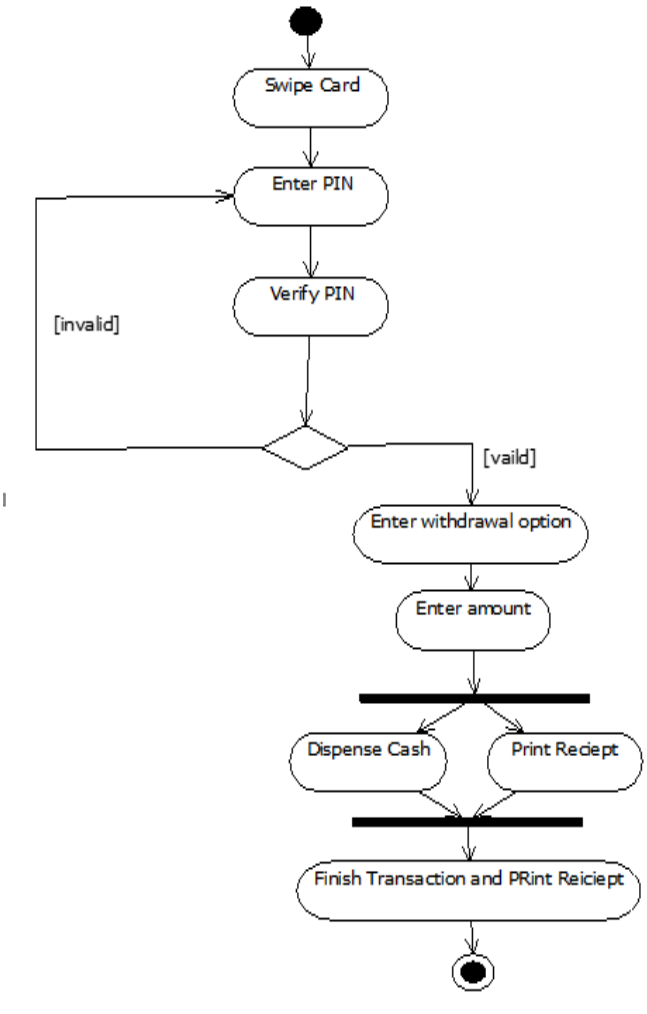


Figure 7. Activity Diagram for *Withdraw Money* use case.

index	type	id	name	kind	inc	inCounter	outCounter	outg
0	PseudoState	UMLPseudostate.5	Initial1	initial		0	1	UMLTransition.36
1	ActionState	UMLActionState.6	Swipe Card		UMLTransition.36	1	1	UMLTransition.37
2	ActionState	UMLActionState.9	Enter PIN		UMLTransition.37 UMLTransition.41	2	1	UMLTransition.38
3	ActionState	UMLActionState.12	Verify PIN		UMLTransition.38	1	1	UMLTransition.39
4	PseudoState	UMLPseudostate.15	Decision1	branch	UMLTransition.39	1	2	UMLTransition.40 UMLTransition.41
5	ActionState	UMLActionState.16	Enter Withdrawal Option		UMLTransition.40	1	1	UMLTransition.42
6	ActionState	UMLActionState.19	Enter Amount		UMLTransition.42	1	1	UMLTransition.43
7	PseudoState	UMLPseudostate.22	Synchronization1	fork	UMLTransition.43	1	2	UMLTransition.44 UMLTransition.45
8	ActionState	UMLActionState.23	Dispense Cash		UMLTransition.44	1	1	UMLTransition.47
9	ActionState	UMLActionState.26	Deduct amount from Account		UMLTransition.45	1	1	UMLTransition.46
10	ActionState	UMLActionState.29	Prepare to print reiept		UMLTransition.46	1	1	UMLTransition.48
11	PseudoState	UMLPseudostate.32	Synchronization2	join	UMLTransition.47 UMLTransition.48	2	1	UMLTransition.49
12	ActionState	UMLActionState.33	Display message		UMLTransition.49	1	1	UMLTransition.50
13	FinalState	UMLFinalState.35	FinalState1	final	UMLTransition.50	1	0	

Figure 8. Snapshot of the output showing various activities and attributes of Withdraw Money Activity Diagram

Fig. 8 shows various activities and its details obtained after parsing the XMI file. The first column in the table indicates the index of each activity. It is a unique number given to each activity, so that it becomes easy to access the activity whenever required. Second column gives the type of each node of an Activity diagram. As we have already seen earlier, there can be 4 kind of Activity node namely action state, pseudo state and final state. Third column gives the xmi id of each state. Fourth column of grid view gives name of each state of Activity Diagram. Next comes the kind of the node which can be either initial, branch, fork, join or final. This attribute is very much essential to traverse the path in case of fork-join pair or looping construct. Inc and Outg field in the grid view specify the incoming and outgoing transition ids of each state. Each transition is labelled as UMLTransition followed by a unique number. At last, there is a field called inCounter and outCounter, which keeps track of number of incoming and outgoing transitions. In total there are 14 states in the given AD starting from 0 to 13.

completePath	weight	ACoverage	noOfStatesCovered	totalStates	TCoverage	noOfTransCovered	totalTrans
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Dispense Cash_Deduct amount from Account_Prepare to print reiept_Synchronization2_Display message_FinalState1	16	100	14	14	93.33	14	15
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Deduct amount from Account_Dispense Cash_Prepare to print reiept_Synchronization2_Display message_FinalState1	16	100	14	14	93.33	14	15
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Deduct amount from Account_Prepare to print reiept_Dispense Cash_Synchronization2_Display message_FinalState1	16	100	14	14	93.33	14	15
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Dispense Cash_Deduct amount from Account_Prepare to print reiept_Synchronization2_Display message_FinalState1	21	100	14	14	100	15	15
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Deduct amount from Account_Dispense Cash_Prepare to print reiept_Synchronization2_Display message_FinalState1	21	100	14	14	100	15	15
Initial1_Swipe Card_Enter PIN_Verify PIN_Decision1_Enter PIN_Verify PIN_Decision1_Enter Withdrawal Option_Enter Amount_Synchronization1_Deduct amount from Account_Prepare to print reiept_Dispense Cash_Synchronization2_Display message_FinalState1	21	100	14	14	100	15	15

Figure 9. Snapshot of the output showing various Test Paths for *Withdraw Money* Activity Diagram

Fig. 9 shows all possible test paths generated from the input activity diagram. The first three paths consider that the valid PIN is entered and permutes all three activities present with in fork-join pair. In the next three paths, it considers the entry of invalid PIN code which will cause the loop to be traversed again. Thus in total six paths are generated. For each of the paths its weight, activity coverage and transition coverage values are displayed. Weight of one node is given by: Number of incoming edges for a node X number of outgoing edges for a node. For a path, weight is sum of weights of nodes in that path. For the sake of simplicity, the activity coverage in case of loop is considered to be 100%

For each of the test paths generated, a separate test case is generated. Since there are six different paths, we get six test cases shown in the snapshots shown below. Each test case consists of the state name in that path and test input and Expected Output for each of the states. The test cases generated here are not so accurate since starUML doesn't support object diagram XMI generation. Hence the object diagram couldn't be integrated with the activity diagram output.

The test case generated for one of the path of Withdraw Money AD is shown in Fig. 10

State Name	Test Input	Expected Output
Initial1		
Swipe Card	Card	Card details
Enter PIN	PIN code	Entered PIN code
Verify PIN	EntryAction1	Vaild/Invalid PIN
Decision1		
Enter Withdrawal Option	Choice	Enter amount screen
Enter Amount	Amount value	a:Amount
Synchronization1		
Dispense Cash	a:amount	Cash
Deduct amount from Account	Amount value	Amount value is deducted from Account table
Prepare to print reiept	Transaction details	r:Reciept
Synchronization2		
Display message	transaction complete	Display successful message
FinalState1		

Figure 10. Sanpshot of ne Sample Test case for Withdraw Money AD

CONCLUSION

Software Testing is a time consuming and costly process in software development life cycle. Automation of this phase may lead to overcome the above problems and also reduces the human effort in other ways it also helps in detecting the human intended errors and logical errors as well. Automation of testing will not be that much productive in terms of time consuming and cost, if we have to wait till the end of the SDLC stage i.e. if we follow the white box testing methodology of testing. If any errors will be detected in this stage, we have to go for that part of the code and design document as well. We have to follow up strict verification of both code and design document from beginning to short out the error. So only one solution to this problem is to, start the testing process from early stage of SDLC i.e. from requirement specification stage through design phase up to the last phase.

REFERENCES

[1] C. Jorgensen, Software Testing: a Craftsman's approach. No. Ed. 2, CRC PRESS, 2002.

[2] O.M.Group, "Unifiedmodelinglanguagespecification."http://www.omg.org/.

[3] Mitrabinda Ray et al., "Test case design using conditioned slicing of activity diagram, "International Journal of Recent Trends in Engineering, Vol. 1, No. 2, May,2009.

[4] A. Nayak and S. Debasis, "Synthesis of test scenarios using UML activity diagrams, software and systems modelling,"vol.10, Springer Berlin/Heidelberg, pp. 63-89, 2011.

[5] N. Khurana and R. S. Chillar, "Test Case Generation And Optimization using UML models

and Genetic Algorithm,"vol. 57, Elsevier Procedia Computer Science 57 (2015) 996 – 1004, 2015.

[6] Debasish Kundu and Debasis Samanta: "A Novel Approach to Generate Test Cases from UML Activity Diagrams", In: Journal of Object technology. Vol. 8, No. 3, May-June (2009).

[7] J. Ajay Kumaret al., "A novel approach for test case generation from UML activity diagram,"vol. 57, Elsevier Procedia Computer Science 57(2015) 996 – 1004, 2014.

[8] Meiliana et al.,"Automated test case generation from UML activity diagram and sequence diagram using depth first search algorithm,"vol.57, Elsevier Procedia Computer Science 57 (2015) 996 – 1004, 2014.

[9] L. Liping et al., "Extenics-based test case generation for UML ACTIVITY DIAGRAM,"vol. 57, Elsevier Procedia Computer Science 57 (2015) 996– 1004, 2014.

[10] W. Linzhang et al., "Generating test cases from UML activity diagram based on Gray-Box method," in Proceedings of the 11thAsia-PacificSoftware Engineering Conference, pp. 284–291, 2004.

[11] R. Chandler et al., "An automated approach to generating usage scenarios from UML activity diagrams," in Proceedings of 12th Asia Pacific Software Engineering Conference, pp. 9–16, 2005.

[12] MKLab, "GTS: GNU StarUML Modelling tool." http://staruml.io/.

[13] IBM,"RationalRose https://www.ibm.com/software/rational/uml/products/

[14] 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, (2004).

[15] P. Nanda, D. P. Mohapatra and S. K. Swain, "Generation of Test Scenarios Using Activity Diagram", In: Proceedings of SPIT-IEEE Colloquium and International Conference, Mumbai, India.

[16] Kim H., Kang S., Baik J. and Ko I., "Test cases generation from UML Activity diagrams", In: Proceedings of 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 556–561 (2007).

[17] Mingsong, C., Xiaokang, Q. and Xuandong, L.: "Automatic test case generation for UML Activity diagrams", In: Proceedings of the International workshop on Automation of software test, pp. 2–8. Shanghai, China (2006).

[18] Sabharwal, Ritu Sibal and Chayanika S. "Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams", IJCSI International Journal of Computer Science Issu es,Vol. 8, Issue 3,May2011.

[19] Sapna P.G., Hrushikesha Mohanty, "Prioritization of Scenarios based on UML Activity Diagrams", First International Conference on Computational Intelligence, Communication Systems and Networks, 978-0-7695-3743-6/09, (2009).

[20] Pakinam N. Boghdady, Nagwa L. Badr, Mohamed A. Hashim, Mohamed F. Tolbam, "An Enhanced Test Case Generation Technique Based on Activity Diagrams", In: Proceedings of SPIT-IEEE Colloquium and International Conference, Mumbai, India.

[21] Timothy J. Grose, Gary C. Doney, Stephen A. Brodsky, Mastering XMI: Java Programming with XMI, XML, and UML, John Wiley & Sons, (2002).

[22] E.J.Jones, Robert J. Oberg," XML Programming Using C# and .NET", 2006.

[23] Hettab, A., Kerkouche, E. & Chaoui, A. (2015). A Graph Transformation Approach for Automatic Test Cases Generation from UML Activity Diagrams. In J. Y. Chen, M. J. Zaki, T. Kahveci, S. Salem & M. Koyutürk (eds.), C3S2E (p. /pp. 88-97), : ACM. ISBN: 978-1-4503-3419-8

[24] Pradeep Kumar Arora and Rajesh Bhatia, "Agent-Based Regression Test Case Generation using Class Diagram, Use cases and Activity Diagram", Elsvier Procedia Computer Science, Volume 125, 2018, Pages 747-753.

[25] Yoo-Min Choi and Dong-Jin Lim, "Automatic feasible transition path generation from UML state chart diagrams using grouping genetic algorithms", Elsevier Information and Software Technology Volume 94, February 2018, Pages 38-58.

[26] Xinying Wang, Xiajun Jiang and Huibin Shi, "Prioritization of Test Scenarios using Hybrid Genetic Algorithm Based on UML Activity Diagram", 978-1-4799-8353-7/15/$31.00 ©2015 IEEE.

[27] Haiying Sun et. al, "Improving Defect Detection Ability of Derived Test Cases Based on Mutated UML Activity Diagrams", 2016 IEEE 40th Annual Computer Software and Applications Conference.