Test Path Optimization Algorithm based on UML Activity Diagram

Aye Aye Kyaw, Myat Myat Min
University of Computer Studies, Mandalay
ayeayekyaw2009@gmail.com,myatiimin@gmail.com

Abstract

Software testing is an activity of finding defects during execution time for a program to get non-defect software. Software testing plays a vital role in developing software that is free from bugs and defects. Manual test is a cost and time consuming process although it may find many defects in a software application. If the testing process could be automated, the cost of developing software could be reduced obviously within a minimum amount of time. The most critical part of the testing process is the generation of test paths. The system focus on model based test path generation. The paper presents the time taken based on the simple and swim lanes activity diagrams, and the different concurrent activity diagrams according to the experimental results. The TPOA system is as an efficient test generation technique to get the highest test coverage by minimizing time.

1. Introduction

Software testing is a crucial part of software development to guarantee the verification and validation process of the software. Software testing divided by three main phases: test case generation, test execution and test evaluation. Test case generation is the core of any testing process. There are many different approaches to generate the test case and test data from different models as an emerging type of model based testing (MBT).

Model based testing is testing in which the entire test specification is derived in whole or in part from both the system requirements and a model that describe selected functional aspects of the system under test. MBT derives test cases from software models, not from source code. At the earliest phase of Software Development Life Cycle, no one who is user or developer can see the software product; it is possible only at the final stage of the product development. Any errors/faults found out at the final stage, it spends a lot of cost and time to repair. Model based test path generation approaches identify faults in the implementation at early design phase, reduce the software development time, and inspire developer to improve design quality.

The rest of the paper is organized as follows: The next section presents the processes of model based test path generation. The third section describes about the activity diagram that is one of behavioral UML diagrams. The fourth section discusses about test path optimization algorithm (TPOA). Section 5 shows the experimental results that are evaluated the search efficiency of TPOA on the different systems that differ in size and domain. The paper concludes at section 6.

2. Model Based Test Path Generation

Various authors have used the following architecture for generating the test path as shown in the following steps:

2.1. UML Diagrams

One or more UML diagrams are used as an input. UML diagrams are the most common type of models used to represent the requirements based models. They can be categorized into behavioral, interactional and structural diagrams [1]. Behavioral diagrams include activity, state chart, and use case diagrams as well as the four interaction diagrams (communication, interaction, sequence and timing). Interactional diagrams include communication, interaction overview, sequence, and timing diagrams. Structural diagrams include class, component, deployment, object, composite structure and package diagrams.

2.2. Generation of Dependency Table

UML diagram is used to automatically generate the Dependency Table (DT) with all the activities. These activities include decisions, loops and synchronization along with the entity performing the activity. Activity Dependency Table (ADT) also consists of the input and the expected output values for each activity. Dependency of each node or activity on others is also shown clearly in DT. The repeated activities in the diagram are grouped into one symbol only instead of having several symbols for the same activity [2, 3, 4, and 9].

2.3. Generation of Dependency Graph

The Dependency Table (DT) is accomplished to automatically generate the Dependency Graph (DG). The symbols given for each activity are used to name the nodes in the DG where each node represents an activity in the activity diagram. Since repeated activities are given the same symbol in the DT, only one node is created for them no matter how many times they are used in the activity diagram. This will decrease the search space in the DG. The transitions from one activity to another are represented by edges in the DG. The presence of an edge from a node to another is determined by checking the dependency column in the DT for the current node's symbol. Specifically, if it contains the previous node's symbol then an edge from the previous node to the current one is drawn in the DG [2, 4, and 5].

2.4. Generation of All Test Paths

The generated Activity Dependency Graph (ADG) applied to obtain all the possible test paths. A test path is composed of steps represented by successive symbols/nodes (representing the activities) forming a complete path from the start node in ADG to the end node separated by arrows. Details are then extracted from the ADT and added to each node in the test path to obtain all the final test cases. Each node in the test case is accompanied with its input and expected output. Besides, the whole test case will be accompanied with its initial input and final expected output [2, 3, 4, 5 and 9].

2.5. Optimization of Test Path

The effectiveness of testing process relies on the quality of test cases not in the quantity of test cases which in turn lingers the testing time. The test path will be considered those have a good impact in finding the errors along with fulfilling the specified coverage criteria. That is it is an optimized concept, where the best fit test paths are selected for test execution on Software under Test (SUT) to reveal as many errors as possible from the SUT and to cover the SUT within less time and cost, and rests are ignored. Reduction of test paths can be done in two ways. One is reducing the test path at the time of generating that avoids generation of redundant test path,

while the other one can be seen as an optimization problem, as reducing the test path implies optimization of the test suite based on certain defined optimization criteria [6].

3. Activity Diagram

A process flow diagram is a modernized representation of a flow chart. Even though, the process flow diagrams show the internal processing details, it will not normally show the synchronization operations present in a system. Whereas, in UML, such process flow diagrams with concurrency nature of the processes are represented using activity diagrams. Generally, state chart diagrams and activity diagrams are related to each other. As the state chart diagrams focus on the state of an object during a particular process, an activity focuses on the flow of activities involved in a single process. It shows how the activities are interdependent with each other. An activity diagram is a special case of state chart diagram in which the states are actions.

An important characteristic of activity diagrams is their ability to show dependency between activities. The activity diagrams can be of two types:

- Simple activity diagram: This diagram simply resembles the process flow with concurrent processes representation. It will not show the actors or the classes which are responsible for each of the processes.
- Swimlane activity diagram: This diagram simply resembles the process flow with concurrency operations, but also shows the actors or the classes involved in the process. Swimlane is a way to group activities performed by the same actor on an activity diagram or to group a set of activities in a single th Swimlanes also indicate who/what is perfo the activities. In this type, the activity di are divided into object swimlanes that determine which object or actor is responsible for which activity. A single transition comes out of each activity, connecting it to the next activity [7].

3.1. Element of Activity Diagram

An activity diagram has two kinds of modeling elements: Activity nodes and Activity edges.

Activity edges: Edges represent flow of control through the activity. It connects the individual components of activity diagrams.

Activity nodes: There are two main kinds of nodes in activity diagrams:

- Action nodes (AN): Action nodes consume all input data/control tokens when they are ready to generate new tokens and send them to output activity edges. Action, an individual step within an activity, can possess input and output information. The output of one action can be the input of a subsequent action within an activity.
- Control nodes (CN): Control nodes route tokens through the graph. The control nodes include constructs to start the diagram, to terminate the diagram, to choose between alternative flows (decision/ merge), to split or merge the flow for concurrent processing (fork / join). Control node is an activity node used to coordinate the flows between other nodes. Control nodes are Initial node, Flow final node, Activity final node, Decision node, Merge node, Fork node, Join node [2].

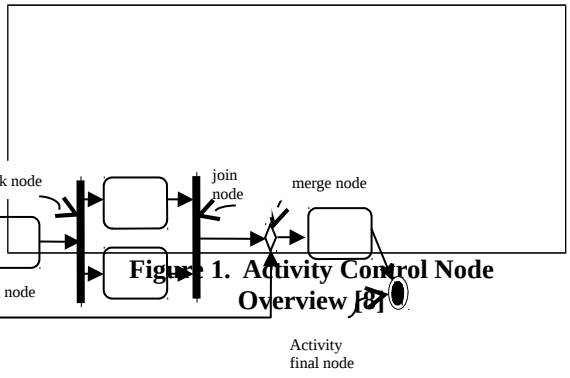

Figure 1. Activity Control Node Overview [8]

3.2. Concurrent Activity Diagram

Activity diagrams are different from flow diagrams in the fact that activity diagrams express parallel behavior which flow diagrams cannot express. Activity diagrams can be classified into two types based on concurrency, non-concurrent activity diagrams and concurrent activity diagrams.

A fork and join pair in an activity diagram are used to process activities in parallel. Four categories of fork and join pairs are defined by [10] namely atomic, simple, nested, and branched fork and join pairs. These categories are further simplified basing on branches present in between fork and join pair. These fork and join pairs are categorized as simple and nested. Simple fork and join contains set activities that can be executed in parallel. Simple with decision and merge are further classified into simple fork and join with loops, simple fork and join with branching. Simple fork join pairs activity diagrams are shown in Figure 2.

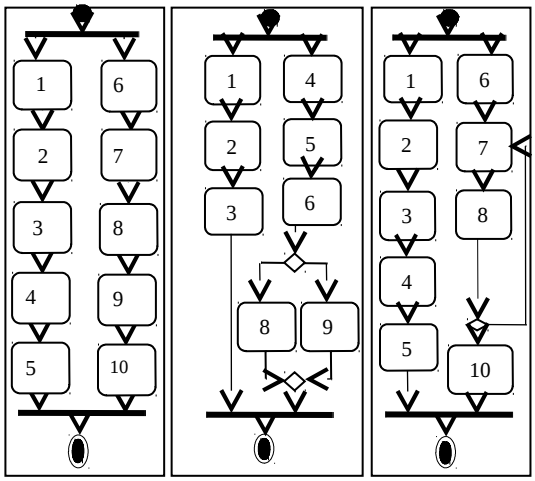

Figure 2. Activity Diagrams with Simple Fork and Join Pairs

Nested fork and join pair contains another set of fork and join pair with set activities that can be executed in parallel. Nested fork and join with decision and merge are classified into nested fork and join with loops, with branching. Figure 3 presents the nested fork and join pairs activity diagrams.

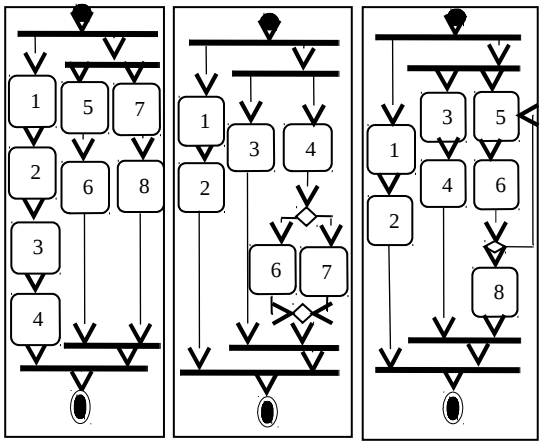

Figure 3. Activity Diagrams with Nested Fork and Join Pairs

4. Test Path Optimization Algorithm

The proposed system uses the activity diagram as an input for the automated algorithm of generating test paths.

Input: XMI file for Activity Diagram.
Output: all possible test paths and the best test path.
begin
k:=0; j:=0; countJoinIn:=0; countForkOut:=0;
countDecision:=0;
totalDecision:=total decision node of the AD;
InitialEdge:= the edge from the initial node;
TP_{kj++} := source node of InitialEdge;
myNode= target node of InitialEdge;
SearchEdge(myNode, InitialEdge);
OptimalTestPath();
End

Figure 4. Algorithm for Automatic Best Test Path Generation

```
SearchEdge(Node s, Edge ee)
begin
  nodeType := type of s node;
  If (nodeType != FinalNode &&
                        (k<=totalDecision*2))
    For each edge e_i ∈ E            // i=1,2,…,n
      sNode := source node of edge e_i;
      tNode := target node of edge e_i;
      if (sNode== s )
        if (nodeType==Action ||
                        nodeType==Merge)
              Add s to TP_kj++ ;
        else if (nodeType==Fork)
              countForkOut : = s.getCountNode();
              countForkOut++;
              s.setCountNode(countForkOut);
              if (countForkOut ==1)
                   Add s to TP_kj++ ;
              endif
        else if (nodeType==Join)
              countJoinIn := s.getCountNode();
              countJoinIn++;
              s.setCountNode(countJoinIn);
              if (countJoinIn == s.getCountIn())
                   Add s to TP_kj++ ;
              endif
        else
              if(countDecision<=totalDecision*2)
                   countDecision++;
                    Add s to TP_kj++ ;
              endif
        endif
        SearchEdge(tNode, e_i);
      endif
    endfor
  else
      Add s to TP_kj++ ; k++;
      countDecision := 0;
      Set count for Fork node and Join node with 0;
  endif
 end
```

Figure 5. **Algorithm of function *SearchEdge***

The Modelio Software has the option of exporting the UML diagram to XMI file. Figure 4 shows the model based test path generation algorithm. This system generates all possible test paths based on the extracted information from XMI file according to the *SearchEdge* function as shown in Figure 5. And then, the system optimizes the best test path among from these generated test paths by using the *OptimalTestPath* function as shown in Figure 6. Finally, the best test path to be tested first can be got within a minimum amount of time.

```
OptimalTestPath()
begin
   maxControlNode:= total control node of TP_0;
   maxTotalNode  := total node of TP_0 ;
   BestPath  := TP_0 ;
   for  each test path TP_i ∈ TP       // i=1,2,…n.
     curControlNode := total control node of TP_i ;
     curTotalNode  := total node of TP_i  ;
  if (curControlNode > maxControlNode )
       if (curTotalNode>= maxTotalNode)
  BestPath := TP_i ;
  maxControlNode  := curControlNode ;
         maxTotalNode  := curTotalNode ;
  else
  BestPath := TP_i ;
  maxControlNode  := curControlNode ;
  endif
  endif
  if (curControlNode  == maxControlNode )
  if ( curTotalNode>= maxTotalNode)
  BestPath := TP_i ;
  maxControlNode  := curControlNode ;
  maxTotalNode  := curTotalNode ;
  endif
  endif
  endfor
  Display BestPath as the optimal test path;
End
```

Figure 6. **Algorithm of function *OptimalTestPath***

5. Experimental Results

This section describes the experimental evaluation results in TPOA. It compares their performance while generating test paths for any

activity diagram. All execution times are measured with nanoseconds.

5.1. Simple and Swimlanes Activity Diagrams

The activity diagrams can be two types: simple activity diagram that is without swim lanes and swim lanes activity diagram that is with swim lanes as described in Section 3. Table 1 describes about the different results of simple and swim lanes activity diagrams. In that table, total test paths, total swim lanes, execution time of TPOA for the best test path and execution time for all generated test paths are shown.

Table 1. Difference results of simple and swim lanes activity diagrams

DNo.	Total Test Paths	Total swim lanes	eTime for TPOA (nanoseconds)	eTime (nanoseconds)
D18	6	2	143366	48372729
D19	6	0	126995	48035021
D20	4	2	84123	47260488
D21	4	0	75370	46190746
D49	5	5	101142	48138812
D50	5	0	98287	47117920

The diagrams with swim lanes are D18, D20 and D49. The simple activity diagrams are D19, D21 and D50, and these are same information with the previous diagrams, but without swim lanes. The execution time of the best test path of the diagram with swim lanes is taken more than the simple activity diagram. The execution time of the generated all test paths of the simple diagrams is spent less than the diagram with swim lane.

5.2. Concurrent Activity Diagram

In this section, the execution time of TPOA is significant different when the difference six types of concurrent activity diagrams such as D40, D41, D42, D43, D44 and D45, are analyzed. eTime in Table 2 represents the execution time for generating all possible test paths and eTime for TPOA represents the execution time for optimizing the best test path. D40 represents the nested fork and join paired with a loop, D41 is the nested fork join with alternate paths and D42 presents the nested fork and join pair. D43 also represents the simple fork join with a loop, D44 presents the simple fork and join pair with alternate paths and D45 is the simple fork join pair. These concurrent activity diagrams information and the time taken by TPOA are shown in Table 2. The processing time for TPOA may differ slightly from all six diagrams. Overall, it can be seen that the execution time for TPOA takes lesser in only the fork join pair diagrams for both simple and nested than in the other fork join pair diagrams. At both simple and nested, the fork join with a loop is more consuming time for TPOA than the fork join pair with alternate paths. The execution time for all generated test paths is directly the ratio to the execution time for TPOA.

Table 2. Information of the difference concurrent activity diagrams

	Total Nodes	Total Edges	Total Control Nodes	Total Decision Nodes	Total Test Paths	eTime for TPOA (nanoseconds)	eTime (nanoseconds)
D 40	14	16	7	1	2	56570	43630684
D 41	14	16	8	1	2	54709	42487880
D 42	14	15	6	0	1	34969	40703304
D 43	14	15	5	1	2	55247	39629678
D 44	14	15	6	1	2	50843	39583086

D 45	14	14	4	0	1	35086	38710367

8.

Most of these activity diagrams have the total test paths based on total decision nodes in each diagram. The more decision nodes, the more total test paths generate.

These tested diagrams are divided into six different decision node groups. Non decision node group has six diagrams. One decision node group consists of twenty diagrams. Two decision nodes group has also seventeen diagrams and three decision nodes group contains thirteen diagrams. There are seven diagrams in four decision nodes group and one group that has more four decision nodes includes four diagrams. To be more visuals based on what types of diagram information, many experiments are performed upon the ascending order for each type that is either the execution time, total nodes, total control nodes, total test paths or total swim lanes of each diagram.

In non decision node group, the information about the simple and swim lanes activity diagrams is shown in Figure 7. D63 uses the highest execution time of TPOA for the best test path because it has the highest number of swim lanes although other diagrams in this group have either total nodes or total control nodes more than it. All simple diagrams that have the same total test paths take a slight different to the execution time of TPOA based on their total nodes and total control nodes as shown in Figure

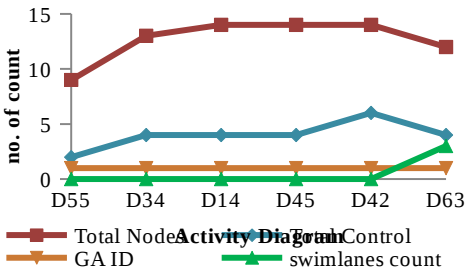

Figure 7. **Diagram Information for Non decision node Group**

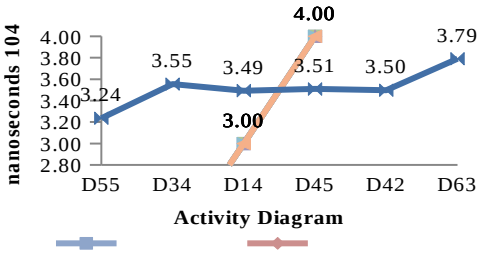

Figure 8. Execution time of TPOA for Non decision node Group

In the over four decision node group, the total number of nodes, control nodes, test paths and swim lanes are shown in Figure 9. Figure 10 illustrate the execution time of TPOA system. The execution time of TPOA takes directly ratio to the total test paths at all simple diagrams. This trend of the execution time of TPOA is similar at the ascending order of something that must be the total number of nodes, control nodes, swim lanes or test paths of this group.

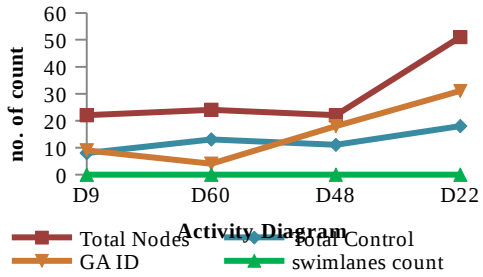

Figure 9. **Diagram Information for Over four decision node Group**

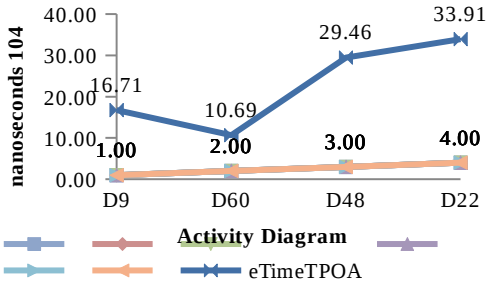

Figure 10. Execution time of TPOA for Over four decision node Group

6. Conclusion

Good software testers cannot avoid the models. MBT has emerged as a useful and efficient testing method for realizing adequate test coverage of systems. The TPOA system can generate not only efficient test paths but also the best test path to lesser effort. This helps in saving time and increases the quality of generated test paths. Besides, the TPOA approach reduces the cost of software development and improves quality of the software. The TPOA system also gives the highest performance of optimized test path generation. The system can perform for test path optimization from only activity diagram. The TPOA system is suitable for any activity diagram i.e. simple fork-join and nested fork-join activities but this activity diagram must obey the standard and rules of UML activity diagram. The system limits the diagrams that are impossible repeatedly nested loops.

References

[1] M. Khandai, A. A. Acharya, D.P. Mohapatra, 2011. "A Survey on Test Case Generation from UML Model". International Journal of Computer Science and Informational Technologies, Vol. 2(3).

[2] P. N. Boghdady, N. L. Badr, M. Hashem and M. F. Tolba. "A Proposed Test Case Generation Technique Based on Activity Diagrams". International Journal of Engineering and Technology, June, 2011. Vol. 11, No. 03.

[3] A.V.K. Shanthi and G. Mohan Kumar. "Automated Test Cases Generation from UML Sequence Diagram". International Conference on Software and Computer Applications (ICSCA 2012), Vol. 41, Singapore.

[4] G. Bhattacharjee and P. Pati, "A Novel Approach for Test Path Generation and Prioritization of UML Activity Diagrams using Tabu Search Algorithm". International Journal of Scientific and Engineering Research, Volume 5, Issue 2, February 2014.

[5] S. Dhir, 2012. “Impact of UML Techniques in Test Case Generation”. International Journal of Engineering Science and Ad-vanced Technology, March–April. Vol. 2, Issue 2, pp. 214-217.

[6] D. Jeya Mala, E. Ruby, V. Mohan, “A Hybrid Test Optimization Framework- Coupling Genetic Algorithm with Local Search Technique”. Computing and Informatics, Vol. 29, pg. 133–164, 2010.

[7] J. Mala and Geetha, “Object Oriented Analysis and Design Using UML”, Tata McGraw-Hill Education, 2013.

[8] http://www.uml-diagrams.org/activity-diagrams-controls.html.

[9] A.V.K. Shanthi; G. MohanKumar. “A Novel Approach for Automated Test Path Generation using TABU Search Algorithm”, International Journal of Computer Applications (0975 – 888) Volume 48–No.13, June 2012.

[10] Xu, D., Li, H., Lam, C.P., 2005 “Using Adaptive Agents to automatically Generate Test Scenarios from the UML Activity Diagrams”, Proceedings of the 13th Asia-Pacific Software Engineering Conference.