A Memorization Approach for Test Case Generation in Concurrent UML Activity Diagram

Suwatchai Kamonsantiroj[1],
Luepol Pipanmaekaporn[2]
Department of Computer and Information Science
King Mongkut's University of Technology North Bangkok
Bangkok, Thailand
suwatchai.k@sci.kmutnb.ac.th[1],
luepol.p@sci.kmutnb.ac.th[2]

Siriluck Lorpunmanee
Department of Data Science and Analytics
Suan Dusit University
Bangkok, Thailand
siriluck_lor@dusit.ac.th

ABSTRACT

Test case generation is the most important part of software testing. Currently, researchers have used the UML activity diagram for test case generation. Testing concurrent system is difficult task due to the concurrent interaction among the threads and the system results in test case explosion. In this paper, we proposed a novel approach to generate test cases for concurrent systems using a dynamic programming technique with tester specification to avoid the path explosion. The tester can configure the concurrency specifications that follow the business flow constraints. In order to evaluate the quality of test cases, activity coverage and causal ordering coverage were measured. By experimental results, the proposed approach is superior as compared to DFS and BFS algorithms. Finally, the proposed approach helps to avoid generating all possible concurrent activity paths which are able to minimize test cases explosion.

CCS Concepts

• **Software and its engineering → Formal software verification**

Keywords

Test Cases Generation; UML Activity Diagram; Software Testing; Concurrency in UML; Dynamic Programming

1. INTRODUCTION

Test case generation is a core phase in any software testing. More than 50 percent of the cost and time are spent on testing the software development [1]. As the complexity and size of software systems grow, more and more time and effort is required for testing. Manual testing consumes a lot effort, and can be error-prone. Thus, it is necessary to develop an automatic testing technique for certain circumstances. Automated testing plays a tremendous role in reducing the time and effort spent during the testing process. Test cases generated automatically can help reliability through increased test coverage. Model-based testing is more efficient and effective than the code based approach as it is the mixed approach of source code and specification requirements to test the software [2-4]. Model based test case generation can be planned at an early stage of the software development life cycle that allows to carry out coding and testing in parallel. The advantage of model based testing is the generated test data at the design stage. This can drastically reduce the testing time and effort [5-6].

ICGDA 2019, March 15–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6245-0/19/03…$15.00
DOI: https://doi.org/10.1145/3318236.3318256

Currently, researchers have used the Unified Modeling Language (UML) for test case generation [6-8]. The Activity Diagram (AD) is one of the important UML models that used when representing the workflows of stepwise activities and actions with support of choice, iteration and concurrency [8-11]. However, finding a test case set from an activity diagram is a terrible task. Because the presence of loop and concurrent activities in the activity diagram results in path explosion, it is not feasible to consider all execution paths for testing. In this paper, we focus on concurrency testing of the activity diagram. A concurrent execution behavior is represented in an activity diagram using fork-join constructs. Concurrent computing is a number of systems in which several computations execute simultaneously and also interacting with each other. Testing a concurrent system is a very difficult task because this type of system can reveal different concurrency condition.

There are major two categories available for concurrent test case generation, single UML model and combinational UML models. First, consider the single UML model. Paper [2] defined activity path coverage criterion which can be used for both loop and concurrent testing of activity diagrams. Paper [2] selected only one representative concurrent activity path form the set of concurrent activity paths that satisfied the same set of precedence relations. The selected concurrent activity path corresponded to the Breadth-First-Search (BFS) traversal of them in the activity graph. Paper [5] proposed an approach of grey-box testing using activity diagram. Test scenarios were generated from an activity diagram following basic path coverage criterion. The basic path coverage criterion helped to reduce path explosion in the presence of the loop. This paper used Depth-First-Search (DFS) traversal from an initial activity state for the final activity state. Next, The I/O explicit Activity Diagram (IOAD) model was proposed by Kim et al. [12]. This technique first built an I/O explicit Activity Diagram

and then converts it into a directed graph. The directed graph is traversed using DFS to generate a set of basic paths. There are other UML models for concurrent testing available such as communication diagram [13], statechart diagram [14], and sequence diagram [15]. Next, concurrent test case generation using the combinational UML models i.e. the combination of sequence diagram and activity diagram [16-17]. In paper [16], an only sequence diagram is converted into an activity diagram to help to traverse sequential and concurrent behaviors. In another combinational UML [17], the activity diagram is converted into an activity graph and sequence diagram is similar converted into a sequence graph. Finally an activity graph and sequence graph are combined to form an activity-sequence graph. The BFS is used to traverse the concurrent testing where the DFS is used to the rest nodes of an activity-sequence graph. However, these approaches do not consider the causality relationship between activities in concurrent testing.

In this paper, we proposed a novel approach to generate test cases in concurrent UML activity diagrams using a dynamic programming technique with tester specification. Dealing with the concurrent problems, we can avoid the generation of an entire set of the path explosion by allowing the tester to configure the concurrency specification $C(a_{ij}, a_{mn})$ which means the activity a_{ij} will occur a head of activity a_{mn}.The tester can configure the concurrency specifications that follow the business flow constraints or causality relationship. For example, the business flow activity of order processing, if a check payment activity is not confirmed then the order will not be shipped.

2. PROBLEM DEFINITION

2. 1 Activity Diagram with Concurrent Activities

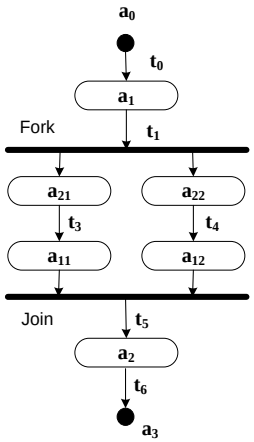

Figure 1. Example of concurrent activities diagram

The UML activity diagram includes concurrent activities and there might be a combination explosion of activity sequences from the system to be tested. The execution of concurrent process begins from the fork node and finishes on join node. To understand the concept of concurrent path explosion, let us consider the example of activity shown in Fig. 1. Given Q is the number of columns or number of parallel processing, n_i is the number of activity node of column q, $q \in \{1,2,\dots,Q\}$, and a_{pq} is activity node where $p \in \{1,2,\dots,n_i\}$. Let defined the precedence relation between two activities a_{iq} and a_{jq} which means activity a_{iq} has occurred a head of activity a_{jq} and it is denoted as $a_{iq} \prec a_{jq}$, In an example of Fig.1, given $Q = 2$, $n_1 = 2$, $n_2 = 2$, and four concurrent activities $a_{11}, a_{21}, a_{12}, a_{22}$. The set of precedence relation is $S = \{a_{21} \prec a_{11},\ a_{22} \prec a_{12}\}$.Then we can report six concurrent activity paths which satisfy all those relations and that are shown.

$$
\begin{aligned}
P_1 &= Fork \xrightarrow{(t_1)} a_{21} \xrightarrow{(t_3)} a_{11} \xrightarrow{con} a_{22} \xrightarrow{(t_4)} a_{12} \xrightarrow{(t_5)} Join \\
P_2 &= Fork \xrightarrow{(t_1)} a_{21} \xrightarrow{con} a_{22} \xrightarrow{con} a_{11} \xrightarrow{con} a_{12} \xrightarrow{(t_5)} Join \\
P_3 &= Fork \xrightarrow{(t_1)} a_{21} \xrightarrow{con} a_{22} \xrightarrow{(t_4)} a_{12} \xrightarrow{con} a_{11} \xrightarrow{(t_5)} Join \\
P_4 &= Fork \xrightarrow{(t_1)} a_{22} \xrightarrow{(t_4)} a_{12} \xrightarrow{con} a_{21} \xrightarrow{(t_3)} a_{11} \xrightarrow{(t_5)} Join \\
P_5 &= Fork \xrightarrow{(t_1)} a_{22} \xrightarrow{con} a_{21} \xrightarrow{con} a_{12} \xrightarrow{con} a_{11} \xrightarrow{(t_5)} Join \\
P_6 &= Fork \xrightarrow{(t_1)} a_{22} \xrightarrow{con} a_{21} \xrightarrow{(t_3)} a_{11} \xrightarrow{con} a_{12} \xrightarrow{(t_5)} Join
\end{aligned} \qquad (1)
$$

where P_i is i-th of activity path, t_i is transition state, and $a_{iq} \xrightarrow{con} a_{jq}$, $a_{pi} \xrightarrow{con} a_{pj}$ or $a_{pi} \xrightarrow{con} a_{jq}$ are the concurrent flow relations between activity a_{iq} and a_{jq} which satisfy set of precedence relation S. For a complex and large system, it is common to have an explosion of concurrent activity paths because there would be a large number of parallel processing or threads processed.

The total number of concurrent activity paths (T) can be defined as:

$$T = \frac{n!}{n_1!\, n_2!\, n_3! \dots\ n_Q!} \qquad (2)$$

Where n is the number of concurrent activity nodes, n_q is the number of activity nodes of column $q \in \{1, \dots, Q\}$. In an example of Fig.1, The number of all concurrent activity nodes $n = 4, n_1 = 2, n_2 = 2$. Then the total number of concurrent activity paths is $T = 4!/2!\,2! = 6$.

2. 2 Dealing Concurrent Activities with Tree Model

In this section we will discuss the tree model for concurrent test cases generation. First of all, let's convert the concurrent activities diagram into the concurrent activities graph that is shown in Fig. 2.

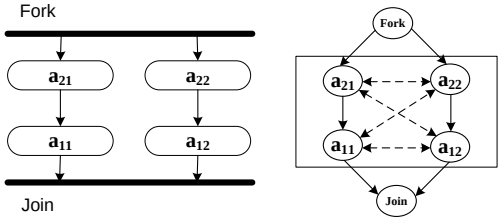

Figure 2. Converting activity diagram into activity graph

All test cases of concurrency structure can be generated with tree model. In this paper, we assume that tree model has the following properties:

1. Given a graph $G = (A_{fj}, E)$ consists of the set of activity nodes (A_{fj}) and the set of relation from one activity to another are represented by edges (E).
2. The activity path (P_i) must be the sequence of unique activity.
3. Let Root node can be derived from the highest activity node of each column $a_{n_1 1}, a_{n_2 2}, \dots a_{n_Q q}$, $q \in Q$. For example, a_{21} and a_{22} are the two root nodes of Fig. 2.

4. Given the current node, there are three options for traversal: its left-node, its right-node and its leaf node or NULL.
   1) If traversing activity is continued on the same column or thread then the sequence of activities must be ordered from the upper level to the lower level $a_{n_i j} \rightarrow a_{(n_i-1)j}$, where n_i is the number of activity node of column $j, j \in Q$.
   2) If traversing activity is crossed to other threads, the highest activity node that has not been traversed is selected.
   3) If the current node is the leaf node, this step is finished and then it goes back to previous node again.

The property 2 implies that the number of root node will be equal to the number of the columns or number of parallel processing. The two generated tree models are illustrated in Fig.3.

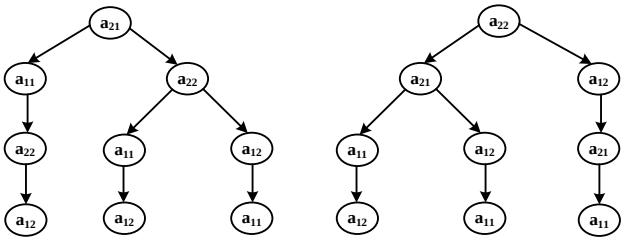

Figure 3. Concurrency structure generated with tree model

2. 3 Test Coverage Criteria

The definition of coverage from the UML activity diagram is the measurement of test quality which is one of the key issues in software testing. The coverage criterions of activity diagram are based on the match between the path from activity diagrams and the execution paths from implementation method [9].To generate test cases for concurrent activity diagram, we consider the following two test case coverage criterions:

- ***Activity coverage*** desires that all activity states in the concurrency among activities of activity diagram be covered. The value of activity coverage is the ratio of the visited activities to all activities in the concurrency among activities of activity diagram.
- ***Casual ordering coverage*** is used for concurrency among activities of activity diagrams. Casual ordering coverage desires that means activity a_{ij} has occurred a head of activity a_{mn} in the activity diagram can be covered. The value of casual ordering coverage is the ratio of the visited paths to all the casual ordering paths in the concurrency among activities of activity diagram.

3. PROPOSED METHOD

3. 1 Generating Test Cases for Concurrent Systems Using A Dynamic Programming

3.3.1 Creating all test cases from 2 columns concurrent activity

The previous paragraph has been shown that the concurrency structure can be generated with tree model as shown in Fig.3. Based on the precedence relation and mapping rules [2], we obtain the concurrent activities graph. The generated tree model can be shown in Fig.4 as tree of recursive calls.

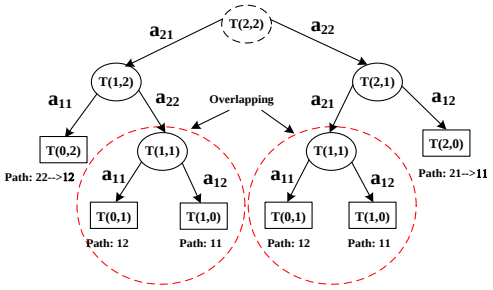

Figure. 4 Tree of recursive calls made by the recursive algorithm for Activity graphs for □(□, □)

As shown in Fig.4, given $\mathrm{T}(2,2)$ is the set of paths from 2 columns concurrent activity $\mathrm{Q} = 2$, it founds that there are the problems with overlapping subproblems.We have known that Dynamic programming is a technique for solving problems with overlapping subproblems [18].It can be calculated by the recurrence relation as the following equation

$$T(n_1, n_2) = a_{n_1 1} \cup T(n_1 - 1, n_2) + a_{n_2 2} \cup T(n_1, n_2 - 1) \quad (3)$$

with there are some initial condition paths

$$T(n_1, 0) = \cup_{i=n_1}^{1} a_{n_1 1} \quad (4)$$

$$T(0, n_2) = \cup_{i=n_2}^{1} a_{n_2 2} \quad (5)$$

where $T(n_1, n_2)$ is the set of paths from 2 columns concurrent activity $Q = 2$, n_i is the number of activity of column $i, i \in \{1,2\}$. $T(n_1, 0)$ and $T(0, n_2)$ are the initial condition paths or based cases that are used to stop the recurrence relation. For example $T(2,0) = a_{21} \rightarrow a_{11}$ and let's denote path $a_{21} \rightarrow a_{11}$ as $21 \rightarrow 11$.

3.3.2 Creating all test cases from Q columns concurrent activity

To compute $T(n_1, n_2, \ldots, n_Q)$, the set of paths from Q columns concurrent activity can be calculated by the recurrence relation as the following equation

$$T(n_1, n_2, \ldots, n_Q) = \sum_{i=1}^{Q} a_{n_i i} \cup T(n_i - 1, n_{-i}) \quad (6)$$

and there are some initial condition paths

$$P[n_i, 0_{-i}] = \cup_{i=1}^{Q} \cup_{j=n_i}^{1} T_i(j, 0_{-i}) \quad (7)$$

where $T(n_1, n_2, \ldots, n_m)$ is the recurrent relation for generate of all concurrent paths, n_i is the number of activity nodes of column $i \in \{1, \ldots, Q\}$. n_{-i} is set of n_j which $j \neq i$, and $\cup_{i=1}^{Q} \cup_{j=n_i}^{1} T_i(j, 0_{-i})$ are the initial condition paths. An example for the initial condition paths, let's assume $Q = 3, n_1 = 1, n_2 = 2, n_3 = 3$ and it is illustrated in Fig. 5.

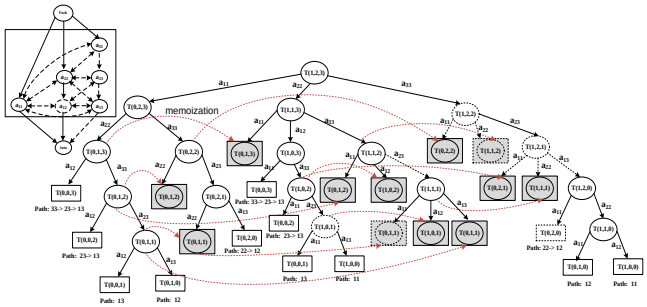

Figure. 5 Tree of recursive calls made by the recursive algorithm for T(1,2,3)

Then the recurrence relation as the following equation

$$T(1,2,3) = a_{11}\,T(0,2,3) + a_{22}\,T(1,1,3) + a_{33}\,T(1,2,2)$$

and the initial condition paths which follow (7) are:

If $i = 1$ and $j \in n_1$

$P[1, 0_{-1}]$ *or* $P[1,0,0] = 11$

Else If $i = 2$ and $j \in n_2$

$P[1, 0_{-2}]$ *or* $P[0,1,0] = 12$

$P[2, 0_{-2}]$ *or* $P[0,2,0] = 22 \rightarrow 12$

Else If $i = 3$ and $j \in n_3$

$P[1, 0_{-3}]$ *or* $P[0,0,1] = 13,$

$P[2, 0_{-3}]$ *or* $P[0,0,2] = 23 \rightarrow 13$

$P[3, 0_{-3}]$ *or* $P[0,0,3] = 33 \rightarrow 23 \rightarrow 13$

End

An algorithm of concurrent path using a memorized technique has the following procedures:

Algorithm Name: Memoized-Path($G, n_1, n_2, \dots, n_Q$)

Input: $G, n_1, n_2, \dots, n_Q$
Output: Concurrency path P_i

1. **Initialize values:** Let' s assign P[$i_1, i_2, \dots, i_Q$] ={},
2. Each table entry initially contains a special value ({})
3. to indicate that the entry has not yet to be filled
4. in for all $i_1, i_2, \dots, i_Q$ where $0 \le i_1 \le n_1$,
5. $0 \le i_2 \le n_2$,…,$0 \le i_Q \le n_Q$
6. **for** i=1 to Q // initial condition paths
7. **for** j=1 to n_1
8. **for** k= n_1 to 1
9. P[j, 0_{-i}] = P[$j, 0_{-i}$] ∪ kj
10. **return** $T(n_1, n_2, \dots, n_Q)$
11.
12. **Algorithm Name:** T($n_1, n_2, \dots, n_Q$)
13. **if** (n_1=1 && n_{-1}=0_{-1})||…||(n_1=n_1 && n_{-1}=0_{-1})
14. **return** P[n_1, 0_{-1}]
15. **else if** (n_2=1 && n_{-2}=0_{-1})||…||(n_2=n_2 && n_{-2}=0_{-2})
16. **return** P[n_2, 0_{-2}]
17. **else if** (n_Q=1 && n_{-Q}=0_{-Q})||…||(n_Q=n_Q && n_{-Q}=0_{-Q})
18. **return** P[n_Q, 0_{-Q}]
19. **else if** (P[$n_1, n_2, \dots, n_Q$]! = { })
20. **return** P[$n_1, n_2, \dots, n_Q$]
21. **else**
22. P[$n_1, n_2, \dots, n_Q$] = $\sum_{i=1}^{Q} a_{n_i i} \cup T(n_i - 1, n_{-i})$
23. **return** P[$n_1, n_2, \dots, n_Q$]

3. 2 Creating A Conditional Test Cases from Q Columns Concurrent Activity

In concurrency testing, a conditional test cases need to be generated in the specified order that follow the business flow constraints. In this paper, we assume that the tester can configure the concurrency specifications that follow the business flow constraints. The business constraint can be defined as:

$$C(a_{ij}, a_{nm}) \qquad (8)$$

where $C(a_{ij}, a_{nm})$ is the conditional activity or the causal ordering among activities in the concurrent execution that was assigned by tester. The a_{ij} has occurred a head of activity a_{nm}.

For example, suppose the $T(1,2,3)$ is the 3 columns concurrent activity and the tester can be specified the business constraint $C(a_{11}, a_{33})$ or $C(11,33)$. It can be illustrated in Fig.6.

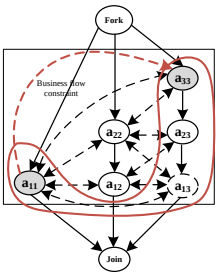

Figure 6. The activity graph of $T(1, 2, 3)$ with the business constraint $C(a_{11}, a_{33})$ or $C(11, 33)$

As shown in Fig. 6, the business flow constraint $C(11,33)$ was considered to generate the test cases. Dealing with the concurrent, we can avoid the generation of an entire set of the path explosion by allowing the tester to configure the concurrency constraint or specification $C(a_{ij}, a_{nm})$. This will make the task of test case generation process easier and reduce testing effort. Our proposed approach modifies memory function with $C(a_{ij}, a_{nm})$ by using top-down dynamic programming allowing traversal of concurrent activity diagram. The Algorithm Memoized-ConstPath is shown:

Algorithm Name: Memoized-ConstPath($G, n_1, n_2, \dots, n_Q, C(a_{ij}, a_{nm})$)

Input: $G, n_1, n_2, \dots, n_Q, C(a_{ij}, a_{nm})$
Output: Concurrency path P_i

1. **Initialize values:**
2. Let' s assign P[$i_1, i_2, \dots, i_Q$] = {}, Each table entry
3. initially contains a special value ({}) to
4. indicate that the entry has not yet to be filled
5. in for all $i_1, i_2, \dots, i_Q$ where $0 \le i_1 \le n_1, 0 \le i_2 \le n_2$,
6. …,$0 \le i_Q \le n_Q$
7. **for** i=1 to Q // initial condition paths
8. **for** j=1 to n_1
9. **for k=** n_1 to 1
10. P[j, 0_{-i}] = P[$j, 0_{-i}$] ∪ kj
11. **return** $T(n_1, n_2, \dots, n_Q, C(a_{ij}, a_{nm}))$
12. **Algorithm Name:** T($n_1, n_2, \dots, n_Q, C(a_{ij}, a_{nm})$)
13. **If** (n_1=1 && n_{-1}=0_{-1})||…||(n_1=n_1 && n_{-1}=0_{-1})
14. **return** P[n_1, 0_{-1}]
15. **else if** (n_2=1 && n_{-2}=0_{-1})||…||(n_2=n_2 && n_{-2}=0_{-2})
16. **return** P[n_2, 0_{-2}]
17. **else if** (n_Q=1 && n_{-Q}=0_{-Q})||…||(n_Q=n_Q && n_{-Q}=0_{-Q})
18. **return** P[n_Q, 0_{-Q}]
19. **else if** P[$n_1, n_2, \dots, n_Q$]! = { } && $a_{ij} \prec a_{mn} = true$
20. **return** P[$n_1, n_2, \dots, n_Q$]
21. **else**
22. **if** $a_{ij} \prec a_{mn} = true$
23. P[$n_1, n_2, \dots, n_Q$] = $\sum_{i=1}^{Q} a_{n_i i} \cup T(n_i - 1, n_{-i})$
24. **return** P[$n_1, n_2, \dots, n_Q$]

4. TEST RESULTS

In this section, we describe our proposed test cases generation method. The following example identify the quality of the test cases generated by DFS, BFS, and Memoized-ConstPath (MCP) which is the proposed method.

The example presented in Fig.6 is a concurrent relation with business flow constraint $C(11,33)$.This example includes casual ordering on the concurrency among activities for three parallel

processing and the number of all the causal ordering paths is equal to 15. The test paths that generated by DFS, BFS, and MPC can be shown in Table 1.

Table 1. Result of the generated test cases from the example presented in Fig.6

Algorithm	Coverage Criteria		
	Activity	Causal Ordering	
DFS	6	2/15	(1) 11→22→12→33→23→13 (2) 22→12→11→33→23→13 (3) 33→23→13→22→12→11
BFS	6	2/15	(1) 11→22→33→12→23→13 (2) 22→11→33→12→23→13 (3) 33→11→22→23→12→13
MCP	6	15/15	(1) 11→22→12→33→23→13 (2) 11→22→33→12→23→13 (3) 11→22→33→23→12→13 (4) 11→22→33→23→13→12 (5) 11→33→22→12→23→13 (6) 11→33→22→23→12→13 (7) 11→33→22→23→13→12 (8) 11→33→23→22→12→13 (9) 11→33→23→22→13→12 (10) 11→33→23→13→23→13 (11) 22→11→12→33→23→13 (12) 22→11→33→12→23→13 (13) 22→11→33→23→12→13 (14) 22→11→33→23→13→12 (15) 22→12→11→33→23→13

As shown in Table 1, the activity coverage of DFS, BFS and MPC are equal to 6. First, the DFS can generates 3 test paths. The test path (1) and (2) are valid the tester define constraint. On the other hand, test path (3) is not valid. Then the causal ordering coverage value is 2/15. Next, the BFS algorithm shows similar causal ordering ratio to DFS although it can generates different test path suits. Finally, the MPC algorithm can be able to maintain the tester define constraint in the test path suits. There are 15 valid test paths and then the causal ordering coverage is 1. These results support that DFS and BFS do not ensure the business flow constraint, whereas our proposed method can do. That is because of DFS and BFS choose the only one path as representative the concurrent activity paths to solve the problem of path explosion of concurrent activities.

5. CONCLUSIONS

In this paper, we have proposed a novel approach to generate test cases on the concurrency among activities of activity diagram using dynamic programming technique with tester specification. We proposed a Memoized-ConstPath algorithm to generate all activity paths which can ensure coverage for concurrency. In order to evaluate the quality of test cases, activity coverage and causal ordering coverage were measured. By experimental results, the proposed approach is superior as compared to DFS and BFS algorithm. Although the proposed algorithm reports are good, there is still a need to develop a more complex system. This would be another direction to achieve better performances.

6. REFERENCES

[1] M. Rajib, Fundamentals of Software Engineering, 2nd ed.,India: Prentice Hall Co., 2009.

[2] K. Debasish, and S. Debasis, "A Novel approach to generate test cases from UML activity diagram. Journal of Object," Technology, 2009, pp. 65-83.

[3] G. Jerry, T. H.-S, W. Ye, Testing and Quality Assurance for Component-Based Software. USA Publication: Artech House, Inc. Norwood, 2003.

[4] S.Monalisa, and M. Rajib, "Automatic test case generation from uml models," In: IEEE 10th International conference on Information Technology, 2007, pp. 196-201.

[5] B. Grady, R. James, and J. Ivar, Unified Modeling Language User Guide, first ed., Addison Wesley, 1998.

[6] L.Wang, J.Yuan, X.Yu, J.Hu, X.Li, G.Zheng, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method," In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, 2004, pp. 284-291.

[7] K. Monalisha, A. Arup, and M.Durga, "A Survey on Test Case Generation from UML Model," International Journal of Computer Science and Information Technologies, vol 2, 2011, pp. 1164-1171.

[8] M. Chen, X. Qiu, and X. Li, "Automatic Test Case Generation for UML Activity Diagrams," In: ACM Proceedings of the 2006 international workshop on Automation of software test, 2006.

[9] C. Mingsong, Q. Xiaokang, X. Wei, W. Linzhang, Z. Jianhua, and L. Xuandong, "UML Activity Diagram-Based Automatic Test Case Generation For Java Programs," The Computer Journal, vol 52, issue 5,1 Aug 2009, pp 545–556, https://doi.org/10.1093/comjnl/bxm057.

[10] T. Walaithip, K Suwatchai, and P. Luepol, "Generating Test Cases from UML Activity Diagram based on Business Flow Constraints," In: ACM the Fifth International Conference on Network, Communication and Computing, 2016, pp. 155-160.

[11] C. Mingsong, M. Prabhat, and K. Dhrubajyoti, "Coverage-driven Automatic Test Generation for UML Activity Diagram," In The Association for Computing Machinery ACM Great Lakes Symposium on VLSI, 2008.

[12] K. Hyungchoul, K. Sungwon, B. Jongmoon, and K. Inyoung, "Test Cases Generation from UML Activity Diagrams," In The Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Paralle/Distributed Computing. SNPD, 2007, pp. 556-561.

[13] S.Philip,M. Rajib, and K. Pratyush, "Automatic test case generation from UML communication diagrams," Information and Software Technology, 2007, pp. 158-171.

[14] B. Adnan, S. A, and R. Sita, "Testing Concurrency and Communication in Distributed Objects," In Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238), 1998.

[15] K. Monalisha, A. Arup, and M. Durga, "A Novel Approach of Test Case Generation for Concurrent Systems Using UML Sequence Diagram," In 2011 3rd International Conference on Electronics Computer Technology. 8-10 April 2011, pp. 157-161

[16] S.Mahesh, and K. Rajeev, “Testing for concurrency in UML Diagrams,” In ACM SIGSOFT Software Engineering Notes, vol 37, September 2012, pp. 1-8.

[17] Dalai S, Acharya A A, Mohapatra D P, “Test Case Generation For Concurrent Object-Oriented Systems Using Combinational Uml Models,” International Journal of Advanced Computer Science and Information Technologies, vol 3, 2011, pp. 97-102.

[18] Bellman, R. E. and Dreyfus, S. E., Applied Dynamic Programming, Princeton Legacy Library Paperback, 2015.