Performance Analysis of Test Path Generation Techniques Based on Complex Activity Diagrams

Walaiporn Sornkliang
Management of Information Technology, School of Informatics
Walailak University, Nakhon Si Thammarat, Thailand
E-mail: vlaiporn@gmail.com

Thimaporn Phetkaew
School of Engineering and Technology, Walailak University, Nakhon Si Thammarat, Thailand
Informatics Innovation Center of Excellence, Walailak University, Nakhon Si Thammarat, Thailand
E-mail: pthimapo@wu.ac.th, thimaporn.p@gmail.com

Keywords: coverage criteria, software testing, test path generation, concurrency test scenario, UML activity diagram

Received: February 5, 2020

Effort reduction in software testing is important to reduce the total cost of the software development project. UML activity diagram is used by the tester for test path generation. It is hard to select the appropriate test path generation technique to diminish the effort of software testing. In the experiment, we compared the efficiency of 12 commonly-used test path generation techniques with both simple activity diagrams and the constructed complex activity diagrams. The experimental results summarized in four aspects. (1) The most appropriate test path generation technique for path testing generates the number of paths equivalent to the target number of all possible paths. (2) The suitable test path generation technique for the concurrency test scenario. (3) The techniques that can generate test paths covering basis path coverage in the case that testing all possible paths for the large or complex object-oriented method is laborious. (4) To compare the efficiency of test path generation algorithms, the percentage test path deviation to the target number of all possible paths is calculated for the constructed complex activity diagrams. We also recommended suitable test path generation methods for each manner of the UML activity diagram.

Povzetek: Avtorji so analizirali uspešnost dvanajst metod za iskanje poti preverjanja programske opreme z enostavnimi in kompleksnimi diagrami aktivnosti.

1 Introduction

Software testing is part of the most important phases of the software development life cycle. Test planning or test design specifications usually occur at the beginning of the system development process. The program can be tested according to the software requirements specification (SRS), or detailed design documents, which reduces the time and cost of software development. Today, most software is developed using Object-Oriented technology and the Unified Modeling Language (UML). UML activity diagram describes the workflow of a sequence of software activities, or concurrent software activities, from the initial activity to the end. An activity diagram is flowchart-like that can be used to generate test paths.

According to Linus's law, given large enough beta-tester and co-developer bases, almost every problem will be characterized quickly and the fix will be obvious to someone [1]. Green software testing takes into consideration the number of people and the amount of equipment allocated to test based on predefined test cases related to energy consumption [2]. Software testing needs to generate test cases to determine the expected output for any program path. There are many methods to generate test paths from a UML activity diagram. Each method is different and yields distinct results because the paths of the program can be traversed in a variety of techniques; thus, selecting the appropriate test path generation method is challenging. If there are too many test paths. It is the cause that uses a lot of effort to design test cases and to execute the program path.

Although general method in the object-oriented programming is not much complex, in case of we want to compare the efficiency of test path generation algorithms, we have to apply those with the complex models of UML activity diagram. There are many types of the control structure of the program such as selection control structure consisting of single-way selection, two-way selection, and nested selection; iteration control structure consisting of pre-test loop and post-test loop; and fork-join structure consisting of simple fork-join, fork-merge concurrent, part-join concurrent, and no-join concurrent. The complex UML activity diagrams are constructed to evaluate test path results of the algorithms by coverage criteria such as statement coverage, branch coverage, activity path coverage, basis path coverage, and path coverage.

The test path generation techniques from UML activity diagram usually depend on graph or tree theory: the test path generation techniques based on tree structure, such as the Dependent Flow Tree [3], the Fault Success Tree Analysis [4], and the Activity Tree [5]; the test path generation techniques based on graph structure, such as the Intermediate Black Box Model [6], the Activity Graph [7], the Activity Flow Table [8], the Activity Convert Grammar [9], the Activity Flow Graph [10], the Test Case Generation Based on Activity Diagram [11], the Activity Dependency Graph [12], and the Intermediate Testable Model [13]. In addition, the test path generation from UML activity diagram can be used the heuristic algorithm e.g., ant colony [14].

Currently, the advantages of the test path generation are applied to several real-world problems, for example, (1) to generate test data using the neighborhood search strategy [15] and using the genetic algorithm [16], (2) to generate test paths for effective chatbot software testing using customized response [17], (3) to optimize test cases by generating test path and selecting test data using Cuckoo search and Bee colony algorithm [18], and (4) to generate optimized test data for saving both testing cost and time [19].

To reduce the effort of software testing, the test paths must be non-redundant, adequate and complete. The purpose of this study was to compare the current test path generation techniques using complex UML activity diagrams constructed by the researcher.

The remainder section of this paper is organized as follows. Section 2 introduces the software testing, test path generation, UML activity diagram, coverage criteria, and literature review. Section 3 explains the experimental setup for this study. The experimental results and discussion of this paper are included in Section 4. Finally, the paper is concluded in Section 5.

2 Background

This section provides a brief overview of software testing, test path generation, UML activity diagram, coverage criteria, and research articles related to this study.

2.1 Software testing

Testing is concerned with bugs or errors, defects or faults, failures, and incidents [20]. Software testing is a crucial part of software quality assurance; it concerns the examination of specifications, designs, and codes [21]. Testing aims to find the detects early in system development. If the fault is found early, then the computational cost will be lower than if the fault is found in the implementation phase. The testing is carried out to assure the quality and reliability of the software. To find the defect as soon as possible, testing activities should start as early as the requirements are derived and should continue until the software is completed [22]. Good test cases have attributes such as a high probability to find bug; the test should not be redundant or too simple or complex [21].

2.2 Test path generation

A test case is a path that covers specific system requirements and data [23]. A test case is made up of a set of test inputs, execution conditions, and expected results developed for the set of objectives [24]. Test cases can be generated automatically from requirements and specifications, design, or the source code [25, 26]. A good test case is a part that has a high probability of finding an as-yet-undiscovered error [27]. To check if the application produces correct outputs, a series of test variables are used as inputs by a tester. Test case generation and also test path generation are an important issue in the software testing field.

2.3 UML activity diagram

An activity diagram is importantly a flowchart that chronologically organizes a set of activities that show the workflow from a start point to the finish that takes place over time [28]. An activity diagram shows the flow from one activity to another. The diagram symbols consist of activities, initial activity, final activity, transition, decision, merge, fork, join, and swimlane, as shown in Table 1.

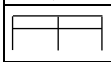
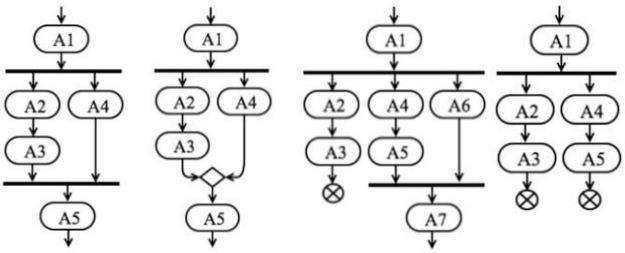
Symbols	Name	Description
	Activity	The process being modelled
	Initial activity	The flow starts in the UML activity diagram
	Final activity	The final step in the UML activity diagram
	Transition	Control flow
	Decision	Alternative activities
	Merge	Brings together one or more incoming flows to accept the single outgoing flow
	Fork	Split transition into multiple fork activities
	Join	The combination of multiple fork activities
	Swimlane	Classification of activities' duty

Table 1: The symbols of the UML activity diagram.



(a) Simple fork-join. (b) Fork-merge concurrent. (c) Part-join concurrent. (d) No-join concurrent.

Figure 1: Types of the concurrent structure in UML activity diagram.

For a concurrent structure in the UML activity diagram [6, 13], the most common form is classified into

four forms as shown in Figure 1. First, the simple fork-join in Figure 1(a) is a pair of fork node and join node and all activity between a fork and join symbol. Second, the fork-merge concurrent in Figure 1(b) has a merge node that is placed instead of a join which allows multiple flows. The paths of the fork node can also be traversed in the same way as a selection structure. Third, the part-join concurrent in Figure 1(c) has two parts: the first part, it has a join node that is used to converge outgoing flows of a fork node; the second part, it is no converging node at the end of these flows. And last, the no-join concurrent in Figure 1(d) is no converging node at the end of these flows.

2.4 Coverage criteria

The coverage criterion is the degree, expressed as a percentage or a specified coverage item that needs to be exercised by a test suite [29]. For concurrency test scenario, the distinction between paths and their respective coverage criteria will help to effective effort estimation and test management [30].

2.4.1 Normal test scenario

The coverage criteria items can be classified into five items. First, statement coverage requires that all statements must be executed at least once. This condition suggests that an error in a statement cannot be revealed without executing the faulty statement. Second, branch coverage requires every decision of the program to be covered by at least one test path. Thus, each decision leads to two test requirements, the decision is either true or false. If every method must be called at least once, each method leads to one test requirement. Third, activity path coverage is the test path that ensures all activities are tested at least once, and all possible paths are tested for all activities [5]. An activity path is flow of activities from the start activity into the final activity in the activity diagram. Fourth, path coverage is the test path that ensures all paths of the program work. To determine path coverage, all paths need to be covered from start to end. And last, basis path coverage is the test path that ensures the optimal test path is covered. The number of paths to ensure the basis path coverage criterion is satisfied can be determined from Cyclomatic complexity [26]. Cyclomatic complexity is the quantitative software metric of the complexity in a program. The cyclomatic complexity value determines the number of independent paths in a basic program set and the maximum amount of testing needed to ensure that all basis paths are covered at least once. Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways. By definition, $V(G)$ is the complexity of the control flow graph. It can be calculated according to each of the following, as in Equation (1), (2) [21, 26], and Equation (3) [31].

$$V(G) = (E - N) + 2, \tag{1}$$

where E is the number of edges between nodes and N is the number of nodes.

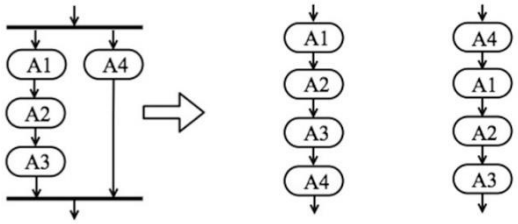
$$V(G) = (P + 1), \tag{2}$$

where P is the number of predicate nodes in control flow graph.

$$V(G) = (R + 1), \tag{3}$$

where R is the number of closed regions of control flow graph.

For the concurrent regions in the UML activity diagram, all possible paths must be traversed from two directions that are left-to-right and right-to-left. For the UML activity diagram in Figure 2(a), there are two possible paths. In the first path in Figure 2(b), it traverses from the left thread to the right thread. In the second path in Figure 2(c), it traverses from the right thread to the left thread. Each thread is listed from the activity of top-level to activity of low-level [32].



(a) Concurrent region. (b) First path. (c) Second path.

Figure 2: The all possible paths in the concurrent regions.

2.4.2 Concurrency test scenario

Execution of activities in the concurrent region leads to concurrency errors if the implementation does not include required restrictions on some of the interleaving paths by setting appropriate synchronization primitives [30].

For the Interleaving Activity Path Coverage (IAPC) criterion [30], let *IP* be the set of interleaving activity paths inside a fork-join structure, and *TS* be the set of test scenarios generated from the activity diagram. The test set *TS* satisfies interleaving activity path coverage, if and only if, $\forall\ p \in IP$, $\exists\ t \in TS$ such that when the program is executed using test scenario '*t*', the interleaving activity path '*p*' of the activity diagram is executed, fully or partially. The interleaving paths are calculated using Equation (4):

$$(\sum_{i=1}^{m} n_i)!/\prod_{i=1}^{m}(n_i!), \tag{4}$$

Where m is the number of threads, n_i is the number of activities in thread i.

2.5 Literature review

In software testing, test paths must be generated to test the software for the expected results. The techniques for test path generation from UML activity diagrams usually depend on tree or graph theory, which are listed in the subsection below.

2.5.1 The test path generation techniques based on tree structures

A tree is a simple hierarchical graph that links together one edge between two nodes. It starts from the root node and

ends at the leaf node. Many tree-based test path generation methods have been developed recently. We describe three methods that are relevant to this research study.

The Dependent Flow Tree [3] is a method that generates test paths from the activity diagram by a constructing dependent flow tree with stores all the information extracted from the XML file of the diagram through the help of a parser. The dependency flow tree consists of nodes and edges. After that, the test paths were generated by using a Depth First Search algorithm that visiting all the nodes and edges exactly once. In this study, the generated test paths include branch coverage and path coverage.

The Fault Success Tree Analysis [4] is a method that generates test paths from an activity diagram by considering the decision of the activity diagram to build a tree. The classification of each tree is built by considering all possible decisions and the paths they took before approaching the next decision. Each decision can pass conditions on the way to the next decision. If conditions were found along the way, the decision conditions are identified in the classification tree and ordered accordingly before reaching the next decision. Then, the test paths are generated from the root node to the leaf node. Next, the fault tree diagram can be generated from the invalid paths of classification tree and the success tree diagram can be generated from the valid path of the classification tree. In this study, the generated test paths include branch coverage.

The Activity Tree [5] is a method that builds test paths from activity diagrams. This suggests the test paths generated from the activity diagrams are transformed into an ordered test flow tree, from the initial activity to the final activity. If activity loops are found in the activity tree, then the activity before the next loop was used as the last activity at the end of the loop. After that, the possible test paths were identified by using a Depth First Search algorithm. If the test paths had loops, this algorithm could search for test paths from the initial node to the last node of the test flow tree only once. In this study, the generated test paths include activity path coverage.

2.5.2 The test path generation techniques based on graph structures

Graph (G) is made up of a set of ordered pairs, $G = (V, E)$, where V is set of nodes, and E is the set of edges, which are the links between nodes. We describe a relevant set of eight methods that use graphs to generate test paths.

The Intermediate Black Box Model [6] is a method that generates test paths from unstructured activity diagrams. The unstructured module is including the loop structure and fork-join structure. The method begins by constructing an activity graph from an activity diagram. Then, the activity graph is classified into a set of groups, including the loop and fork-join structure. The nodes in each group are then combined into a single node. The test paths then searched the graph using the Depth First Search algorithm. If a node has an iteration structure, then the path goes through the loop least once. If all nodes in a fork-join structure, then the path goes through all possible activity. In this study, the generated test paths include path coverage.

The Activity Graph [7] is a method that generates test paths from the activity diagrams. The activity diagram is transformed into an activity graph. The activity graph is a direct graph and is replaced by each node of the activity diagram. The activity graph is ordered using the control flow from the chronological list of activities, including the branch, decision, iteration, and fork-join activities. The test paths then searched the graph using the Depth First Search algorithm that visiting all the nodes and edges exactly once. In this study, the generated test paths include activity path coverage.

The Activity Flow Table [8] is a method that generates test paths from the activity diagram. The activity diagram is used to construct the activity flow table that describes the symbols by numbers given to each activity. Then an activity flow graph is constructed and used the symbols ordered on an activity flow table. The activity flow graph searches all possible paths by using a Depth First Search algorithm that compares each path to a set of criteria using basis path coverage. If a node has an iteration structure, then the path goes through the loop only once. In this study, the generated test paths include basis path coverage and activity path coverage.

The Activity Convert Grammar [9] is a method that generates test paths from activity diagrams and activity convert grammar by constructing an activity dependency table and a decision dependency table from the sets of testing data. Then, the activity dependency table and the decision dependency table construct test paths using the grammar method. The grammar is divided into activities on the Left-Hand Side (LHS) that is used as the dependent activity and activities on the Right-Hand Side (RHS) that tests the branch and fork activity. To generate test paths, the activities are tested chronologically. In the case of a fork activity, the activities are prioritized from left to right and right to left. In this study, the generated test paths include path coverage.

The Activity Flow Graph [10] is a method that generates test paths from activity diagrams. The activity diagrams are used to construct the control flow activity table and values using the conditions of each activity. Then an activity flow graph is constructed and ordered using the control flow from the chronological list of activities including the branch, decision, iteration, and fork activities. The test paths then searched the graph using the Depth First Search. If a node has an iteration structure, then the path goes through the loop only once. Each test path generates a test path. In this study, the generated test paths include activity path coverage.

The Test Case Generation Based on Activity Diagram [11] is a method that generates test paths from activity diagrams according to the following steps. First, the activity dependency table is constructed and used to create an activity dependency graph covering all activities. The activity dependency graph searches all possible paths by using a Depth First Search that compares each path to a set of criteria using basis path coverage. If activity in a loop structure is encountered, the loop is only traversed once.

In this study, the generated test paths include basis path coverage and path coverage.

The Activity Dependency Graph [12] improves the test path generation technique from activity diagrams by constructing an activity dependency table and an activity dependency graph, respectively. Then, the dependency graph is improved by removing the activities which have the same names, decision symbols, fork symbols, join symbols and merge symbols. The improved activity dependency graph is used to construct test paths, which generate the test paths. In this study, the generated test paths include basis path coverage and branch coverage.

The Intermediate Testable Model [13] studies the synthesis of testing situations from activity diagrams. The method begins by constructing a control flow graph from an activity diagram. Then, the control flow graph is classified into a set of groups including the selection, loop, and fork-join structures. The nodes in each group are then combined into a single node. When generating a test path, the tester chooses the structure of interest to build sub-paths. When all the constructs are used to generate the test paths, then that test paths have covered all possible paths. In this study, the generated test paths include path coverage.

The test paths then searched the graph using the Depth First Search algorithm that visiting all the nodes and edges exactly once. In this study, the generated test paths include activity path coverage.

2.5.3 The test path generation techniques based on heuristic algorithm

Orientation-based Ant Colony algorithm (OBACO) [14] is proposed to generate test paths for a concurrent segment of UML activity diagram. Parsing of XMI code takes UML activity diagram as input and results into individual sub-queues of activity nodes present under the fork-join structure. The input of the orientation based ant colony optimization is sub-queues under the fork-join structure of an activity diagram is used for generating combinations between the activity nodes of the sub-queues. The next activity node in the path is decided by pheromone, heuristic values, and orientation factor.

3 Experimental setup

This research looks at 12 commonly-used test path generation techniques. There are three tree-based test path generation algorithms, which are the Dependent Flow Tree (DFT) [3], the Fault Success Tree Analysis (FSTA) [4], and the Activity Tree (ActTree) [5]. There are eight graph-based test path generation algorithms, which are the Intermediate Black Box Model (IBM) [6], the Activity Graph (AG) [7], the Activity Flow Table (AFT) [8], the Activity Convert Grammar (ACG) [9], the Activity Flow Graph (AFG) [10], the Test Case Generation Based on Activity Diagram (TCBAD) [11], the Activity Dependency Graph (ADG) [12], and the Intermediate Testable Model (ITM) [13]. And there is one heuristic-based test path generation algorithm, which is the Orientation-based Ant Colony algorithm (OBACO) [14]. It is difficult to select the appropriate test path generation techniques to reduce the effort of software testing. The researcher has conducted the research by comparing the test path generation techniques using complex UML activity diagrams created by the researcher.

3.1 The UML activity diagrams used in the experiment

To evaluate the test path results of the 12 algorithms, we used both real-world activity diagram and constructed activity diagram. The two real-world activity diagrams, which are a Shopping Mall System activity diagram in Figure 3 and a Library Management System activity diagram in Figure 4. The selection criteria are the activity diagram which describes the daily life work and easy to understand. Two activity diagrams created by the researcher are called a Complex Concurrent Structure activity diagram in Figure 5 and a Complex Control Structure activity diagram in Figure 6. No previous work in the literature review generated test paths from the complex activity diagram. So, we create a complex activity diagram to compare the efficiency of each algorithm in terms of the covered coverage criteria. The control variables of four activity diagrams are the number of paths covered by path testing and basis path coverage. The four activity diagrams employed in the experiment are as follows.

3.1.1 The shopping mall system activity diagram

Figure 3 shows the Shopping Mall System activity diagram, applied from [33, 34], which consists of sequence structure, selection structure, and iteration structure. In this activity diagram does not has the fork-join structure. In this system, a user can select the item, the system can make the billing, and check the member card. The user can select to pay by cash or card. Besides, the user can select the gift service or collect stamp.

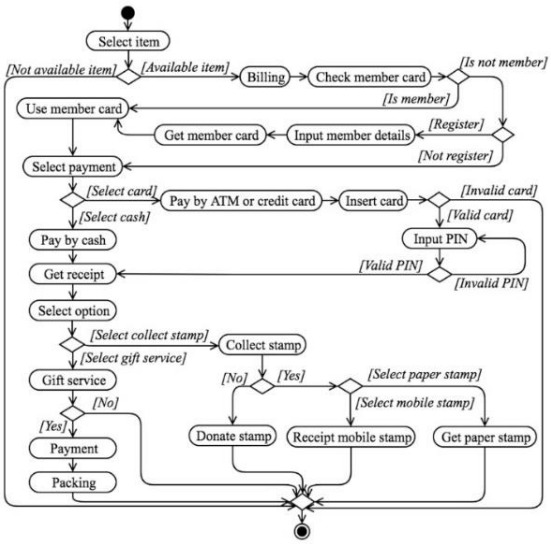


Figure 3: The Shopping Mall System activity diagram.

3.1.2 The Library Management System activity diagram

Figure 4 shows the Library Management System activity diagram, applied from [35, 36], which consists of sequence structure, selection structure, iteration structure, and fork-join structure. In this system, the user inserts the card, inputs the password, and then the system checks the account. If it is the account created, the system checks the status of the account, and the user can decide to return or borrow the book. If it is not the account created, the user registers the new account. The system checks the availability of the book. If the book is available, the system will decrease book availability and increase the number of the book borrowed.

3.1.3 The Complex Concurrent Structure activity diagram

Figure 5 shows the Complex Concurrent Structure activity diagram that is constructed to generated test path focus on fork-join structure. This activity diagram has control structure such as sequence structure, selection structure, and fork-join structure. In this activity diagram does not has the iteration structure. For the selection structure, there is a nested selection. For the fork-join structure, we use fork-join structure classification of the concurrent module i.e., simple fork-join, fork-merge concurrent, part-join concurrent, and no-join concurrent.

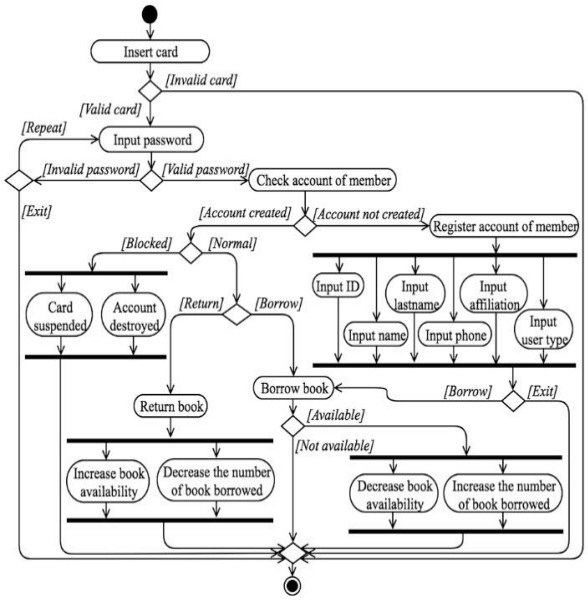


Figure 4: The Library Management System activity diagram.

3.1.2 The Complex Control Structure activity diagram

Figure 6 shows the Complex Control Structure activity diagram consists of all type of control structure. This activity diagram is comprised of the sequence structure, selection structure, iteration structure, and fork-join structure. For selection structure, there is one-way, two-way, and nested selection. Iteration structure consisting of pre-test loop and post-test loop. Within the fork-join structure, there are one-way selection, two-way selection, and nested selection, pre-test loop, post-test loop, simple fork-join, fork-merge concurrent, part-join concurrent, and no-join concurrent.

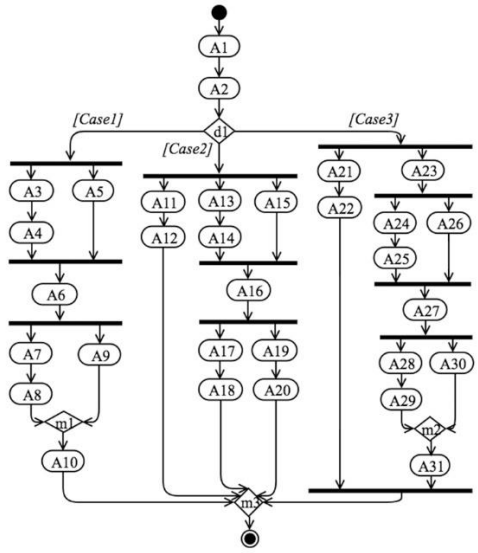


Figure 5: The Complex Concurrent Structure activity diagram.

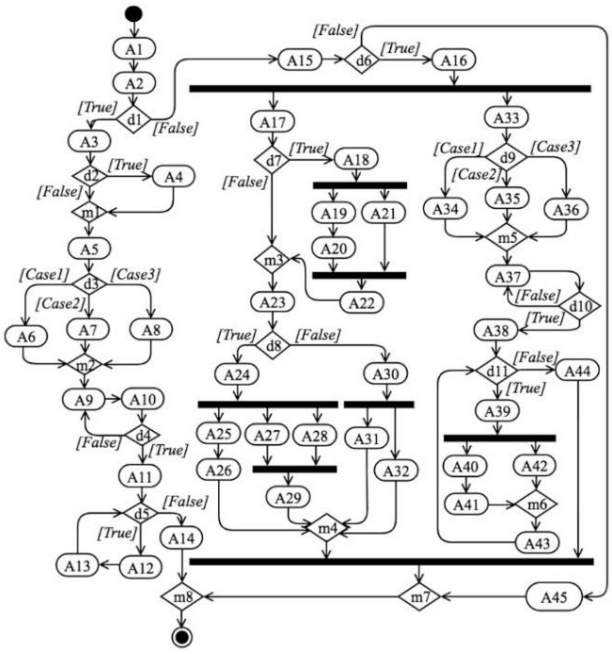


Figure 6: The Complex Control Structure activity diagram.

3.2 The target number of test path

To compare the efficiency of test path generation algorithms, four UML activity diagrams are used to construct in the form of the control flow graphs (CFG), and then each control flow graph is searched to find the target number of all possible path using a Depth First Search algorithm and to find the number of basis path using Equation (1). For path coverage in the case of the concurrent regions, all possible paths must be traversed

from two directions that are left-to-right and right-to-left [32]. The first direction was started from the activity on the left transition and goes as far as it can down a given path to reach the join symbol, then backtracks until it finds an unexplored path, and then explores it. These procedures are repeated until the entire concurrent region has been explored. For the second direction, it was started from the activity on the right transition. For the concurrency test scenario, the target number of all possible path in the fork-join structure is calculated by using Equation (4).

4 Experimental results and discussion

In this section, we show the experimental results for 12 test path generation techniques with four activity diagrams. For simplicity to show the result, we use the word "Shopping" short for Shopping Mall System activity diagram, the word "Library" short for Library Management System activity diagram, "Concurrent" short for Complex Concurrent Structure activity diagram, and "Complex" short for Complex Control Structure activity diagram. The results of test path generation techniques based on tree structures, graph structures, and heuristic algorithm are shown in Table 2. The target number of all possible paths (TAP) and the target number of basis paths (TBP) are shown in the table to compare with the number of test paths of each technique. The coverage criterion used in the experiment are statement coverage (SC), branch coverage (BC), activity path coverage (AP), basis path coverage (BP), and path coverage (PC). The symbol ✓ means the technique found out coverage of that criteria, whereas symbol × means the technique is not satisfied with that criteria.

For the concurrency test scenario, we show the experimental results for the Intermediate Black Box Model (IBM) and the Intermediate Testable Model (ITM) which are the test path generation methods focused on the concurrency region. Table 3 shows the coverage percentage of test path generation techniques for the interleaving activity path coverage (IAPC) criteria. Both of the Intermediate Black Box Model (IBM) and the Intermediate Testable Model (ITM) generate the same number of paths as the target paths.

Test path generation techniques based on tree structures

Activity diagram	Target		DFT						FSTA						ActTree					
	TAP	TBP	Test paths	Coverage criteria					Test paths	Coverage criteria					Test paths	Coverage criteria				
				SC	BC	AP	BP	PC		SC	BC	AP	BP	PC		SC	BC	AP	BP	PC
Shopping	49	11	49	✓	✓	✓	✓	✓	49	✓	✓	✓	✓	✓	37	✓	✓	✓	✓	×
Library	33	17	65	✓	✓	✓	✓	✓	17	✓	✓	✓	✓	×	10	✓	✓	✓	×	×
Concurrent	17	10	14	✓	✓	✓	✓	×	3	✓	✓	✓	×	×	7	✓	✓	✓	×	×
Control	565	20	58	✓	✓	✓	✓	×	73	✓	✓	✓	✓	×	115	✓	✓	✓	✓	×

Test path generation techniques based on graph structures

Activity diagram	Target		IBM						AG						AFT					
	TAP	TBP	Test paths	Coverage criteria					Test paths	Coverage criteria					Test paths	Coverage criteria				
				SC	BC	AP	BP	PC		SC	BC	AP	BP	PC		SC	BC	AP	BP	PC
Shopping	49	11	49	✓	✓	✓	✓	✓	49	✓	✓	✓	✓	✓	34	✓	✓	✓	✓	×
Library	33	17	5,777	✓	✓	✓	✓	✓	65	✓	✓	✓	✓	✓	33	✓	✓	✓	✓	✓
Concurrent	17	10	139	✓	✓	✓	✓	✓	14	✓	✓	✓	✓	×	14	✓	✓	✓	✓	×
Control	565	20	60,505	✓	✓	✓	✓	✓	58	✓	✓	✓	✓	×	30	✓	✓	✓	✓	×

Activity diagram	Target		ACG						AFG						TCBAD					
	TAP	TBP	Test paths	Coverage criteria					Test paths	Coverage criteria					Test paths	Coverage criteria				
				SC	BC	AP	BP	PC		SC	BC	AP	BP	PC		SC	BC	AP	BP	PC
Shopping	49	11	49	✓	✓	✓	✓	✓	50	✓	✓	✓	✓	✓	49	✓	✓	✓	✓	✓
Library	33	17	33	✓	✓	✓	✓	✓	18	✓	✓	✓	✓	×	65	✓	✓	✓	✓	✓
Concurrent	17	10	17	✓	✓	✓	✓	✓	7	✓	✓	✓	×	×	14	✓	✓	✓	✓	×
Control	565	20	565	✓	✓	✓	✓	✓	173	✓	✓	✓	✓	×	58	✓	✓	✓	✓	×

Activity diagram	Target		ADG						ITM					
	TAP	TBP	Test paths	Coverage criteria					Test paths	Coverage criteria				
				SC	BC	AP	BP	PC		SC	BC	AP	BP	PC
Shopping	49	11	34	✓	✓	✓	✓	×	49	✓	✓	✓	✓	✓
Library	33	17	26	✓	✓	✓	✓	×	5,777	✓	✓	✓	✓	✓
Concurrent	17	10	14	✓	✓	✓	✓	×	139	✓	✓	✓	✓	✓
Control	565	20	24	✓	✓	✓	✓	×	60,505	✓	✓	✓	✓	✓

Test path generation technique based on heuristic algorithm

Activity diagram	Target		OBACO					
	TAP	TBP	Test paths	Coverage criteria				
				SC	BC	AP	BP	PC
Shopping	49	11	49	✓	✓	✓	✓	✓
Library	33	17	65	✓	✓	✓	✓	✓
Concurrent	17	10	17	✓	✓	✓	✓	✓
Control	565	20	565	✓	✓	✓	✓	✓

Table 2: A comparison of the result of test path generation techniques.

Activity diagram	Target	IBM		ITM	
	IAPC	Test paths in the fork-join structures	Interleaving activity path coverage criteria (%)	Test paths in the fork-join structures	Interleaving activity path coverage criteria (%)
Library	5,772	5,772	100	5,772	100
Concurrent	139	139	100	139	100
Control	60,480	60,480	100	60,480	100

Table 3: A comparison of the result of test path generation techniques for the concurrency test scenario.

The percentage deviation (PD) of each test path results can be calculated to determine how much each method deviates from all possible paths, generated from the Complex Concurrent Structure activity diagram and the Complex Control Structure activity diagram. The equation to calculate the percentage deviation is listed below, as in Equation (5).

$$PD = \left(\frac{TP - N}{N} \right) x100, \tag{5}$$

where *PD* is the percentage deviation, *TP* is the number of test paths by each method, and *N* is the target number of the all possible paths of the activity diagram.

For example, from Table 2 the Dependent Flow Tree (DFT) could generate 14 paths of the Complex Concurrent Structure activity diagram, but the target number of all possible paths of the Complex Concurrent Structure activity diagram was 17 paths. So, the percentage deviation of DFT in Equation (5) is ((14-17)/17) x 100 = -17.65%, as shown in Table 4. A positive value indicates that the number of generated test paths is larger than the target number of all possible paths. A negative value indicates that the number of generated test paths is smaller than the target number of all possible paths.

Table 4 shows the percentage deviation of test path generation techniques for the Complex Concurrent Structure activity diagram and the Complex Control Structure activity diagram.

Test path generation techniques	Complex Concurrent Structure activity diagram (TAP=17 paths)			Complex Control Structure activity diagram (TAP=565 paths)		
	Number of test paths	Number of the different test path	Percentage deviation (%)	Number of test paths	Number of the different test path	Percentage deviation (%)
DFT	14	-3	-17.65	58	-507	-89.73
FSTA	3	-14	-82.35	73	-492	-87.08
ActTree	7	-10	-58.82	115	-450	-79.65
IBM	139	122	717.65	60,505	59,940	10,608.85
AG	14	-3	-17.65	58	-507	-89.73
AFT	14	-3	-17.65	30	-535	-94.69
ACG	17	0	0.00	565	0	0.00
AFG	7	-10	-58.82	173	-392	-69.38
TCBAD	14	-3	-17.65	58	-507	-89.73
ADG	14	-3	-17.65	24	-541	-95.75
ITM	139	122	717.65	60,505	59,940	10,608.85
OBACO	17	0	0.00	565	0	0.00

Table 4: The percentage deviation of test path generation techniques for the Complex Concurrent Structure activity diagram and the Complex Control Structure activity diagram.

Figure 7 shows the percentage deviation of test paths generated of the Complex Concurrent Structure activity diagram. Figure 8 shows the percentage deviation of test paths generated of the Complex Control Structure activity diagram. The positive value indicates that the number of generated test paths is larger than the target number, whereas a negative value indicates that the number of generated test paths is smaller than the target number.

In case of the Complex Concurrent Structure activity diagram, the percentage deviation of the Intermediate Black Box Model (IBM), and the Intermediate Testable Model (ITM) is 717.65%, this means that they generated excessive test paths. The percentage deviation of the Fault Success Tree Analysis (FSTA) is -82.35%, this means that it generated inadequate test paths. The percentage deviation of the Activity Convert Grammar (ACG) and the Orientation-based Ant Colony algorithm (OBACO) are 0%, this means that they generate the equivalent test paths.

In case of the Complex Control Structure activity diagram, the percentage deviation of the Intermediate Black Box Model (IBM), and the Intermediate Testable Model (ITM) is 10,608.85%, this means that they generated excessive test paths. The percentage deviation of the Activity Dependent Graph (ADG) is -95.75%, this means that it generates inadequate test paths. The percentage deviation of the Activity Convert Grammar (ACG) and the Orientation-based Ant Colony algorithm (OBACO) are 0%, this means that they generate the same number of paths as the target paths.

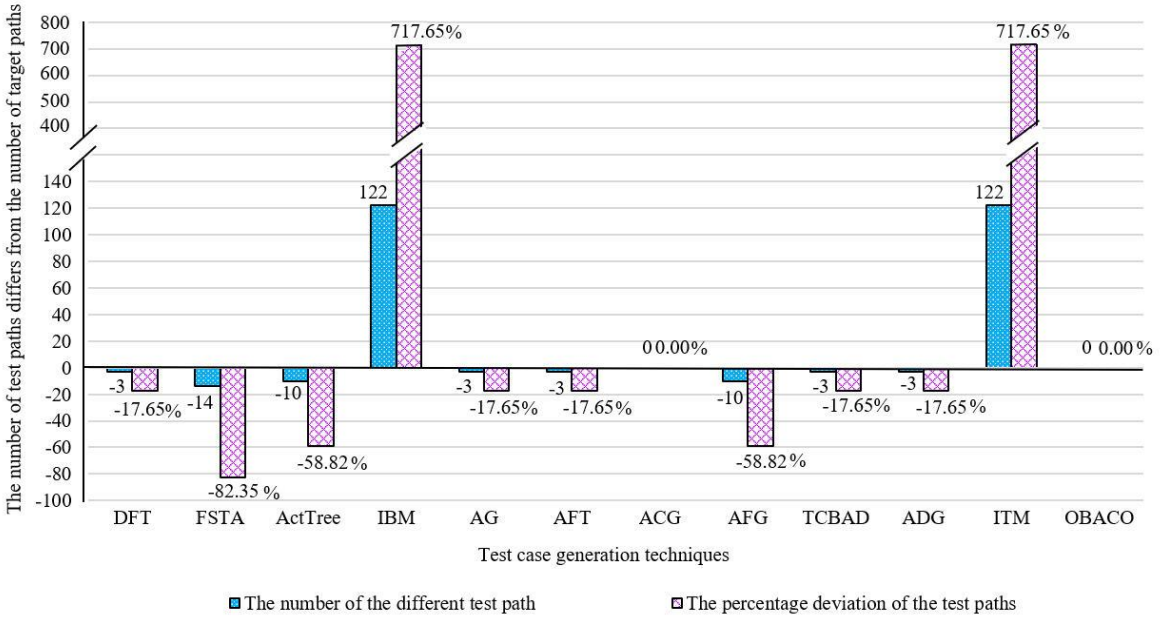


Figure 7: A comparison of the percentage deviation between the number of test paths generate from the Complex Concurrent Structure activity diagram and the target number of all possible paths.

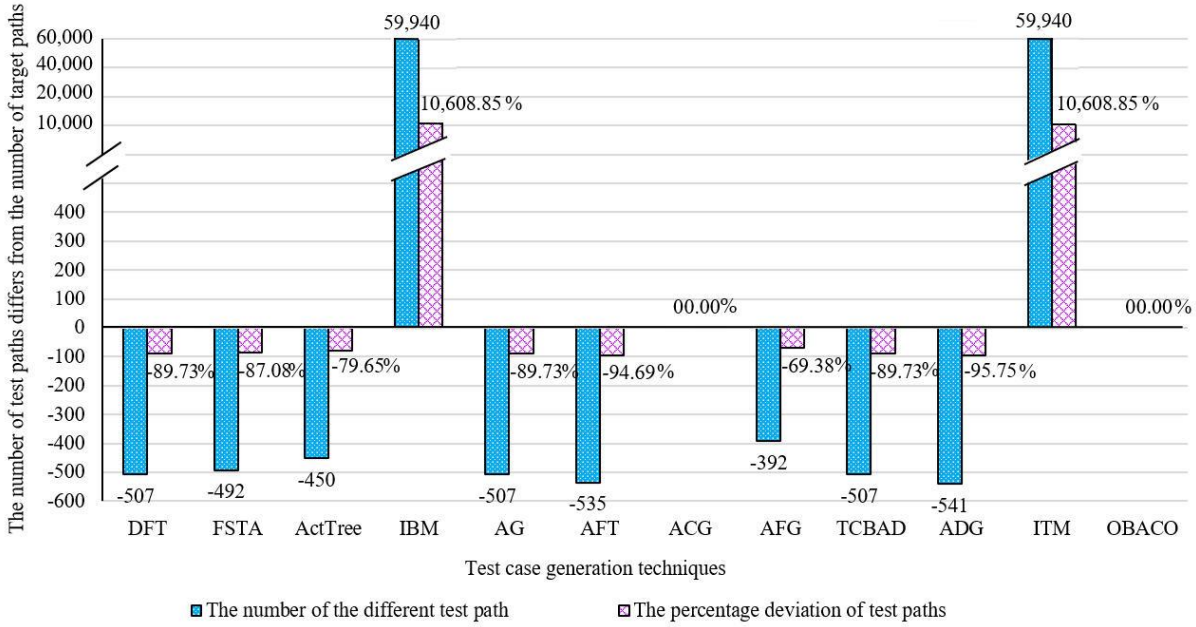


Figure 8: A comparison of the percentage deviation between the number of test paths generate from the Complex Control Structure activity diagram and the target number of all possible paths.

The difference between the test path results and the target number of all possible paths is mainly occurred at the fork-join structure. A fork-join symbol in a UML activity diagram is a control node that splits a transition into multiple concurrent transitions. Test path generation of the activities within the fork-join of the 12 test path generation techniques is different. The number of test paths depends on the number of transitions and the number of activities in each transition. The path traversal to find test paths for the fork-join structure is shown in Table 5.

The experimental results from Table 2 - 4 can be summarized into four aspects: (1) the aspect of covering the path coverage for both simple (real-world) activity diagrams and our proposed complex activity diagram, (2) the aspect of covering the interleaving activity path coverage for the concurrency test scenario, (3) the aspect of covering the basis path coverage, and (4) the aspect of the efficiency of test path generation algorithms applying with the complex activity diagram. And, we also suggested the proper activity diagram for each test path generation method, as shown in Table 6.

Test path generation techniques	The path traversal in fork-join structure
DFT, AG, AFT, ADG, TCBAD	It starts from the top activity on the left thread, traverses through the activities in the low level in the same thread till it reaches the join symbol, and then it goes out of the join symbol.
ACG	There are two directions. In the first direction, it traverses from the left thread to the right thread. And in the second direction, it traverses from the right thread to the left thread. Each thread is listed from the activity of top-level to activities of low-level.
FSTA, ActTree, AFG	It starts from the top activity on the left thread and goes as far as it can down a given path to reach the join symbol, then backtracks until it finds an unexplored path, and then explores it. These procedures are repeated until it traverses through all thread and finally it goes out of the join symbol.
IBM, ITM	The calculation of all possible paths in the fork-join structure is $N!/(n!*n!)$, where N is the sum of all activities in the fork-join, and n is the sum of all activities in each transition.
OBACO	The paths in the fork-join structure are generated through the use of separate ant agents for performing traversal with the sub-transition.

Table 5: The path traversal in the fork-join structure in test path generation techniques.

Test path generation techniques	Suitable UML activity diagram
DFT, AG, TCBAD	Simple activity diagram with sequence, selection, iteration, or fork-join structure.
ActTree, AFT, ADG	Simple activity diagram with sequence and selection structure.
FSTA	Simple activity diagram with sequence, selection, and iteration structure.
AFG	Simple or complex activity diagram focusing on loop testing.
IBM, ITM	Simple or complex activity diagram focusing on concurrency test scenario in the fork-join structure.
ACG, OBACO	Simple or complex activity diagram with sequence, selection, iteration, or fork-join structure.

Table 6: UML activity diagrams which are suitable for the test path generation techniques.

(1) For path coverage in the case of the simple (real-world) activity diagram, the Dependent Flow Tree (DFT), the Intermediate Black Box Model (IBM), the Activity Graph (AG), the Activity Convert Grammar (ACG), the Test Case Generation Based on Activity Diagram (TCBAD), the Intermediate Testable Model (ITM), and the Orientation-based Ant Colony algorithm (OBACO) could generate the number of test paths that satisfied the path coverage, while the Fault Success Tree Analysis (FSTA), the Activity Flow Table (AFT), and the Activity Flow Graph (AFG) could generate the number of test paths that occasional satisfied the path coverage. For path coverage in the case of the constructed complex activity diagrams, only the Intermediate Black Box Model (IBM), the Activity Convert Grammar (ACG), the Intermediate Testable Model (ITM), and the Orientation-based Ant Colony algorithm (OBACO) could generate the number of test paths that satisfied the path coverage. There is an exceptional case for the Activity Tree (ActTree), which satisfied to activity path coverage and the Activity Dependency Graph (ADG), which satisfied for basis path coverage.

(2) For the concurrency test scenario, in both cases of the simple (real-world) activity diagram and the constructed complex activity diagrams, the Intermediate Black Box Model (IBM) and the Intermediate Testable Model (ITM) could generate the number of test paths that satisfied 100% interleaving activity path coverage.

(3) In case that it is difficult and take a lot of effort to test all possible paths for complex activity diagram, the tester should select the basis paths coverage instead. The experimental results depict that the Dependent Flow Tree (DFT), the Intermediate Black Box Model (IBM), the Activity Graph (AG), the Activity Flow Table (AFT), the Activity Convert Grammar (ACG), the Test Case Generation Based on Activity Diagram (TCBAD), the Activity Dependency Graph (ADG), the Intermediate Testable Model (ITM), and the Orientation-based Ant Colony algorithm (OBACO) could generate test paths that covered the basis paths.

(4) For efficiency comparison of test path generation algorithms with the constructed complex activity diagrams., Figure 7 and 8 depict that the Activity Convert Grammar (ACG) and the Orientation-based Ant Colony algorithm (OBACO) could generate the equivalent test paths to the target number of all possible paths. Whereas, the Intermediate Black Box Model (IBM), and the Intermediate Testable Model (ITM) could multiply the number of test paths if there were a lot of activities in the fork-join structure.

5 Conclusion

This paper presents the performance analysis of test path generation algorithms. To compare the efficiency of test path generation algorithms, we applied 12 commonly-used techniques with both simple (real-world) activity diagrams and the constructed complex activity diagrams. In this research, we constructed two complex activity diagrams, i.e., the Complex Concurrent Structure activity diagram and the Complex Control Structure activity diagram.

The experimental results show that to test all possible paths, the Activity Convert Grammar (ACG) and the Orientation-based Ant Colony algorithm (OBACO) are the most appropriate test path generation technique which can generate the number of paths equivalent to all possible paths. Besides, other test path generation techniques such as the Intermediate Black Box Model (IBM) and the Intermediate Testable Model (ITM) can cover path coverage, but there are too many test paths. However,

these two methods can cover 100% interleaving activity path coverage for the concurrency test scenario.

Testing all possible paths for the large or complex object-oriented method is laborious, the tester should select the basis paths coverage instead. The Dependent Flow Tree (DFT), the Intermediate Black Box Model (IBM), the Activity Graph (AG), the Activity Flow Table (AFT), the Activity Convert Grammar (ACG), the Test Case Generation Based on Activity Diagram (TCBAD), the Activity Dependency Graph (ADG), the Intermediate Testable Model (ITM), and the Orientation-based Ant Colony algorithm (OBACO) are the appropriate test path generation techniques that can cover the basis path coverage of both the simple and complex activity diagram.

References

[1] Matjaž Gams and Tine Kolenik. Relations between electronics, artificial intelligence and information society through information society rules. Electronics, 10(4): 514, 2021. https://doi.org/10.3390/electronics10040514

[2] Mahdi Dhaini, Mohammad Jaber, Amin Fakhereldine, Sleiman Hamdan, and Ramzi A. Haraty. Green computing approaches - A survey. Informatica, 45(1): 1-12, 2021. https://doi.org/10.31449/inf.v45i1.2998

[3] Oluwatolani Oluwagbemi and Hishammuddin Asmuni. Automatic generation of test cases from activity diagrams for UML based testing (UBT). Jurnal Teknologi (Science & Engineering), 77(13): 37-48, 2015. https://doi.org/10.11113/jt.v77.6358

[4] Pimthip Paiboonkasemsut and Yachai Limpiyakorn. Reliability tests for process flow with fault tree analysis. In 2015 2nd International Conference on Information Science and Security (ICISS), IEEE, 14-16 December, Seoul, South Korea, pp. 1-4, 2015. https://doi.org/10.1109/ICISSEC.2015.7371028

[5] Ranjita Kumari Swain, Vikas Panthi, Durga Prasad Mohapatra, and Prafulla Kumar Behera. Prioritizing test scenarios from UML communication and activity diagrams. Innovations in Systems and Software Engineering, 10(3): 165-180, 2014. https://doi.org/10.1007/s11334-013-0228-5

[6] Yufei Yin, Yiqun Xu, Weikai Miao, and Yixiang Chen. An automated test case generation approach based on activity diagrams of SysML. International Journal of Performability Engineering, 13(6): 922-936, 2017. https://doi.org/10.23940/ijpe.17.06.p13.922936

[7] Namita Khurana, Rajender Singh Chhillar, and Usha Chhillar. A novel technique for generation and optimization of test cases using use case, sequence, activity diagram and genetic algorithm. Journal of Software, 11(3): 242-250, 2016. https://doi.org/10.17706/jsw.11.3.242-250

[8] Ajay Kumar Jena, Santosh Kumar Swain, and Durga Prasad Mohapatra. A novel approach for test case generation from UML activity diagram. In 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), IEEE, 7-8 February, Ghaziabad, India, pp. 621-629, 2014. https://doi.org/10.1109/ICICICT.2014.6781352

[9] Kanjanee Pechtanun and Supaporn Kansomkeat. Generation test cases from UML activity diagram based on AC grammar. In 2012 International Conference on Computer and Information Science (ICCIS), IEEE, 12-14 June, Kuala Lumper, Malaysia, pp. 895-899, 2012. https://doi.org/10.1109/ICCISci.2012.6297153

[10] Ranjita Kumari Swain, Vikas Panthi, and Prafulla Kumar Behera. Generation of test cases using activity diagram. International Journal of Computer Science and Informatics, 4(1): 35-44, 2014. https://www.interscience.in/ijcsi/vol4/iss1/8

[11] Chanda Chouhan, Vivek Shrivastava, and Parminder S Sodhi. Test case generation based on activity diagram for mobile application. International Journal of Computer Applications, 57(23): 4-9, 2012. https://doi.org/10.5120/9436-3563

[12] Pakinam N. Boghdady, Nagwa L. Badr, Mohamed A. Hashim, and Mohamed F. Tolba. An enhanced test case generation technique based on activity diagrams. In The 2011 International Conference on Computer Engineering & Systems, IEEE, 29 November - 1 December, Cairo, Egypt, pp. 289-294, 2011. https://doi.org/10.1109/ICCES.2011.6141058

[13] Ashalatha Nayak and Debasis Samanta. Synthesis of test scenarios using UML activity diagram. Software and Systems Modeling, 10: 63-89, 2011. https://doi.org/10.1007/s10270-009-0133-4

[14] VinayArora, Maninder Singh, and Rajesh Bhatia. Orientation-based ant colony algorithm for synthesizing the test scenarios in UML activity diagram. Information and Software Technology, 123: 106292, 2020. https://doi.org/10.1016/j.infsof.2020.106292

[15] Sapna Varshney and Monica Mehrotra. A hybrid particle swarm optimization and differential evolution based test data generation algorithm for data-flow coverage using neighborhood search strategy. Informatica, 42(3): 417-438, 2018. https://doi.org/10.31449/inf.v42i3.1497

[16] Aman Jaffari, Aman Jaffari, and Jihyun Lee. Automatic test data generation using the activity diagram and search-based technique. Applied Sciences, 10(10): 3397, 2020. https://doi.org/10.3390/app10103397

[17] Mani Padmanabhan. Sustainable test path generation for chatbots using customized Response. International Journal of Engineering and Advanced Technology, 8(6): 149-155, 2019. https://doi.org/10.35940/ijeat.D6515.088619

[18] Lakshminarayana P and T V SureshKumar. Automatic generation and optimization of test case using hybrid cuckoo search and bee colony algorithm. Journal of Intelligent Systems, 30(1): 59-72, 2021. https://doi.org/10.1515/jisys-2019-0051

[19] Manju Khari and Prabhat Kumar Khari. An effective meta-heuristic cuckoo search algorithm for test suite optimization. Informatica, 41(3): 363-377, 2017.

http://www.informatica.si/index.php/informatica/article/view/1174/1069

[20] Paul C. Jorgensen. Software testing a craftsman's approach. 4th ed., CRC Press, London, 2014.

[21] Roger S. Pressman. Software engineering: a practitioner's approach. 7th ed., McGraw-Hill, New York, NY, 2010.

[22] Subashni, S. and Satheesh Kumar N. Software testing using visual studio 2010. Packt Publishing, Birmingham, UK, 2010.

[23] Mahesh Shirole and Rajeev Kumar. UML behavioral model based test case generation: a survey. ACM SIGSOFT Software Engineering Notes, 38(4): 1-13, 2013. https://doi.org/10.1145/2492248.2492274

[24] IEEE Computer Society. 829-2008-IEEE standard for software and system test documentation. The Institute of Electrical and Electronics Engineers, New York, NY, pp.1-84, 2008. https://doi.org/10.1109/IEEESTD.2008.4578383

[25] Itti Hooda and Rajender Chhillar. A review: study of test case generation techniques. International Journal of Computer Applications, 107(16): 33-37, 2014. https://doi.org/10.5120/18839-0375

[26] Md. Abdur Rahman, Md. Abu Hasan, Khaled Shah, and Md. Saeed Siddik. Multipartite based test case prioritization using failure history. International Journal of Advanced Science and Technology, 129: 25-42, 2019. http://sersc.org/journals/index.php/IJAST/article/view/1353/1084

[27] B.B. Agarwal, S.P. Tayal, and M. Gupta. Software engineering & testing. Jones and Bartlett, Sudbury, Massachusetts, pp.157-164, 2010.

[28] Grady Booch, James Rumbaugh, and Ivar Jacobson. The unified modeling language user guide. Addison-Wesley Longman, Reading, Massachusetts, pp. 311-331, 1998.

[29] Mauro Pezz and Michal Young. Software testing and analysis: process, principles, and techniques. John Wiley & Sons, Hoboken, NJ, pp. 211-230, 2008. https://ix.cs.uoregon.edu/~michal/book/Samples/book.pdf

[30] Mahesh Shirole and Mahesh Shirole. Concurrency coverage criteria for activity diagram. IET Software, 15(1): 43-53, 2021. https://doi.org/10.1049/sfw2.12009

[31] Frank Tsui, Orlando Karnal, and Barbara Bernal. Essentials of software engineering. 4th ed., Jones & Bartlett Learning, Burlington, Massachusetts, pp. 170-171, 2017.

[32] Farid Meziane and Sunil Vadera. Artificial intelligence applications for improved software engineering development: new prospects. Information Science Reference, Hershey, New York, NY, pp. 248, 2010. https://doi.org/10.4018/978-1-60566-758-4

[33] Prateeva Mahali and Prateeva Mahali. Model based test case prioritization using UML activity Diagram and evolutionary algorithm. International Journal of Computer Science and Informatics, 4(2): 76-81, 2014. https://doi.org/10.47893/ijcsi.2014.1177

[34] Sonali Khandai, Sonali Khandai, and Sonali Khandai. Prioritizing test cases using business criticality test value. International Journal of Advanced Computer Science and Applications, Science and Information Organization, 3(5): 103-110, 2011. https://doi.org/10.14569/IJACSA.2012.030516

[35] Oluwatolani Oluwagbemi, Hishammuddin Asmuni. An approach for automatic generation of test cases from UML diagram. International Journal of Software Engineering and Its Applications, 9(8): 87-106, 2015. https://www.earticle.net/Article/A252751

[36] Monalisha Khandai, Arup Abhinna Acharya, and Durga Prasad Mohapatra. Test case generation for concurrent system using UML combinational diagram. International Journal of Computer Science and Information Technology, 2(3): 1172-1182, 2011. http://ijcsit.com/docs/Volume%202/vol2issue3/ijcsit2011020344.pdf