Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-021-00389-4

ORIGINAL ARTICLE

Constrained permutation-based test scenario generation from concurrent activity diagrams

Mahesh Shirole[1] · Rajeev Kumar[2]

Received: 17 May 2020 / Accepted: 8 March 2021

Abstract
Concurrency in application systems can be designed and visualized using concurrent activity diagrams. Such diagrams are useful to design concurrency test scenarios for testing. However, the number of test scenarios inside a *fork-join* construct could be exponential in size. The commonly used permutation technique generates all possible test scenarios, but it is exponential in size. Existing UML graph theoretic-based approaches generate a few test scenarios for concurrency testing. But they do not consider the full functionality of concurrent activity diagrams. In this work, we present two constrained permutation-based test scenario generation approaches, namely the *level permutation* and *DFS level permutation* for concurrent activity diagrams. These approaches restrict the exponential size to a reasonable size of test scenarios. It is achieved by generating a subset of permutations at different levels. The generated test scenarios are sufficient to uncover most concurrency errors like synchronization, data-race, and deadlocks. The proposed technique improves interleaving activity path coverage up to 35% compared to the existing approaches.

Keywords UML design · Activity diagram · Level permutation · Test scenario generation

1 Introduction

There has been a growing shift from *sequential* systems to *concurrent* systems due to the advent of multi-core hardware and language-level software support for concurrency. To better utilize multi-core hardware, application systems are designed using concurrency constructs. However, testing a concurrent software remains a challenging task because of its inherent properties like non-deterministic execution and the need for synchronization.

Concurrency in application systems is designed and visualized using UML behavioral models. Concurrent designs could be implemented using a variety of high-level programming language constructs. There could be errors in its implementation. There are several research papers, addressing issues of concurrency errors in implementations using high-level programming language constructs, e.g., Dinning [9], Lu et al. [17], Suss and Leopold [24], Vaziri et al. [26], Yamada et al. [28], etc. It is practical to design test scenarios for concurrency testing early in the development life cycle (i.e., at the design stage) with help of concurrent design models.

A permutation is one common technique to generate test scenarios from concurrent designs represented by activity diagrams. A concurrent activity diagram with n activities inside a fork-join construct has $n!$ different permutations; however, each permutation may not be a valid interleaving test scenario. A sequence $S = < a_1, a_2, \ldots, a_n >$ is a *valid test scenario* if a_1 is the start node, a_n is the end node, and (a_i, a_{i+1}) is an edge such that a_i is the predecessor node of a_{i+1} of the activity diagram. A *valid interleaving test scenario* is a valid test scenario if it maintains causal order among the activities of different threads. Therefore, valid interleaving test scenarios are a subset of all the possible permutations. How to choose a suitable subset remains a challenging task.

Consider two threads, say, $Thread_1 = a_1 - a_2 - a_3$ and $Thread_2 = a_4 - a_5 - a_6$ are inside the *fork-join* construct,

✉ Mahesh Shirole
mrshirole@it.vjti.ac.in

Rajeev Kumar
rajeevkumar.cse@gmail.com

1 Computer Engineering and Information Technology Department, Veermata Jijabai Technological Institute, Mumbai 400 019, India

2 School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi 110 067, India

Published online: 01 April 2021

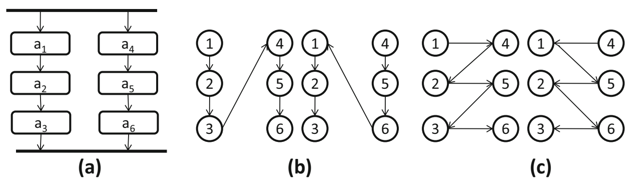


Fig. 1 **a**)UML activity diagram and its test scenarios generated by: **b** modified-DFS and **c** DFS-BFS approaches

as shown in Fig. 1a. A permutation leads to $6! = 720$ test scenarios.

To avoid such an exponential size test scenario set, several researchers have proposed several graph-theoretic techniques to generate a minimum number of test scenarios, e.g., modified-DFS [7] and DFS-BFS [13]. In Fig. 1b and c, activity numbers are used to represent activities for simplicity of the presentation. Figure 1b shows two test scenarios generated using the modified-DFS method; these scenarios are sequentilized test scenarios and have one context switch in each scenario. Note that a sequentilized test scenario is a scenario having activities of one thread followed by the activities of another thread. Figure 1c shows two test scenarios generated using the DFS-BFS method. These scenarios are interleaving test scenarios, which have five context switches in each scenario. Both approaches, namely modified-DFS and DFS-BFS generate test scenarios in proportion to the permutations of the number of threads. Earlier research efforts for UML activity diagram-based approaches, e.g., [6], [8], [16] focused only on sequential aspects for the test scenario generation. Conventional algorithms exhibit limitations to generate concurrent test scenarios, e.g., [18]. Many approaches, e.g., [1], [7], [11], [13], [15], [18], and [23] explored concurrent activity diagrams to generate test scenarios. However, these approaches do not explore interleaving activity space, thereby generating a small number of test scenarios. Some of the recent approaches explore interleaving space of activities using genetic algorithms [3], [20], the memoized-ConstPath algorithm [10], the modified DFS algorithm [25], and dynamic programming [29]. However, there is a need to systematically explore an interleaving activity space and increase the number of test scenarios to improve interleaving activity path coverage [22], thereby uncovering a larger set of concurrency errors.

A permutation is a sequence containing each element from a finite set; each element is to be included in the sequence exactly once [12]. In other words, a permutation is an arrangement of k elements and there are $k!$ permutations. Generating exhaustive test scenarios for all interleaving activity sequences is exponential in size. Constrained/restricted permutations are useful to study the concurrent execution behaviors; thus, they are useful for uncovering concurrency errors. In concurrency, the *happen-before* relationship [14] among tasks is a constraint, which limits the number of permissible executions in a concurrent execution. In Fig. 1a, a relation $a_1 \preceq a_3$ of tasks does not allow a permutation that includes sub-sequence $< \ldots a_3 - a_1 \cdots >$ in a concurrent execution. Thus, a partial order among concurrent tasks limits the number of executions. Restricted permutations [4] avoid some arrangements in various prescribed ways. One may consider a constrained permutation like level permutation, which satisfies happen-before relations in concurrent execution for the testing purpose. For example, in Fig. 1a if we consider tasks: $< a_1, a_4 >$ in the first level (L_1), $< a_2, a_5 >$ in the second level (L_2), and $< a_3, a_6 >$ in the third level (L_3) then the arrangements induced by the order $L_1 - L_2 - L_3$ with internal permutations in each level are constrained permutations. Constrained permutations are shown to be finite [2].

The main focus of the research in this paper is the generation of constrained permutations that lead to valid interleaving test scenarios from concurrent activity diagrams. This paper presents two constrained permutation-based approaches, namely *level permutation* and *DFS level permutation*, to generate test scenarios from concurrent activity diagrams. The generated test scenarios maintain a *happen-before* relationship [14] or a causal order. Therefore, they are useful to uncover synchronization and deadlock errors. The proposed level permutation approach limits the exponential scale-up of test scenarios. The proposed approach improves the interleaving activity path coverage up to 35% in comparison with existing approaches, e.g., Chandler et al. [7], Kundu and Samanta [13], and Shirole and Kumar [21].

The rest of the paper is organized as follows: Sect. 2 presents a brief overview of UML activity diagrams and their classification. Section 3 includes related work. Section 4 discusses the proposed permutation-based test scenario generation techniques. Experimental results are discussed and analyzed in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Activity diagram and concurrency constructs

Unified modeling language (UML) has become a de-facto standard for modeling, analyzing, and designing of large and complex software systems. UML designs are well suited and supportive for concurrency modeling in various forms of specifications. For example, state machine diagrams support concurrent system design using *concurrent states*, sequence diagrams support concurrent system design using *parallel fragments*, and activity diagrams support concurrent system design using *fork-join* constructs. Activity diagrams having *fork-join* construct(s) depict *concurrency*. The dynamic behavior of such a system plays an important role in the test case design process.

Table 1 Number of paths generated for decision, loop, and fork-join structures of activity diagrams

Structure	No. of paths	Remarks
Decision	2	To achieve a decision, coverage of at least two paths is required
Loop	2	Two test cases are sufficient, looping and not looping [5]
Fork-join	$(n_1 + n_2)!/n_1! \times n_2!$	Two threads with number of activities, n_1 and n_2, in each thread

An activity diagram defines the logic of complex operations and describes the work-flow of systems. An activity diagram usually consists of activities, transitions, decision points, guards, parallel activities, and swim-lanes. The semantics of an activity diagram is described in detail in UML superstructure specifications [19]. The basic idea of an activity diagram is to model activities and possible orders of their execution. Activity diagrams emphasize sequential and concurrent control flows. A sequential control flow results in a single flow of activities one after another. A concurrent control flow results in multiple simultaneous flows of activities. Activity diagrams support concurrent flows using *fork-join* structures. Unlike a $decision$, a $fork$ splits the sequential flow into different concurrent out-going flows converged at the $join$ construct. At execution time, a flow of activities in a sequential flow is deterministic, while a flow of activities in concurrent flow may be non-deterministic. A concurrent flow may take any random path of activities to cover activities inside the fork-join construct.

In an activity diagram, a concurrency construct *fork-join* supports two run-time behaviors, namely *parallel* and *concurrent* computations. Parallel computation is treated as a simultaneous execution of two or more tasks from different *non-dependent* threads. In contrast, a concurrent computation is a simultaneous execution of two or more tasks from different *dependent* threads. Dependent threads have dependent operations, e.g., using shared data/object(s) or waiting for some task to happen. Threads, which are not dependent, are called non-dependent threads. Testing concurrent computations in dependent threads increase confidence in reliability of the software. In a parallel computation, a few test scenarios are sufficient to verify execution of all parallel tasks and synchronization of threads at a *join* construct [13,18]. However, concurrent computation demands a systematic exploration of all possible test scenarios to uncover concurrency errors. In this paper, we explore concurrent activity diagrams for generating all valid interleaving test scenarios.

The complexity of an activity diagram is an important factor for the testing purpose. Higher the complexity, the more difficult is the testing. Therefore, a large number of test cases are required for complex parts of a process flow. In other words, complexity is directly proportional to the number of test scenarios. Thus, calculating a number of test scenarios helps in estimating the testing effort. For example, the number of test scenarios required for decision, loop, and fork-join structures is summarized in Table 1.

The formula for two fork-join structures can be generalized for more than two fork-join structures. Consider a fork-join structure having m exit entries, wherein each exit-entry has n_i activities up to the join where $i \geq 1$. Then, the number of possible execution interleavings is:

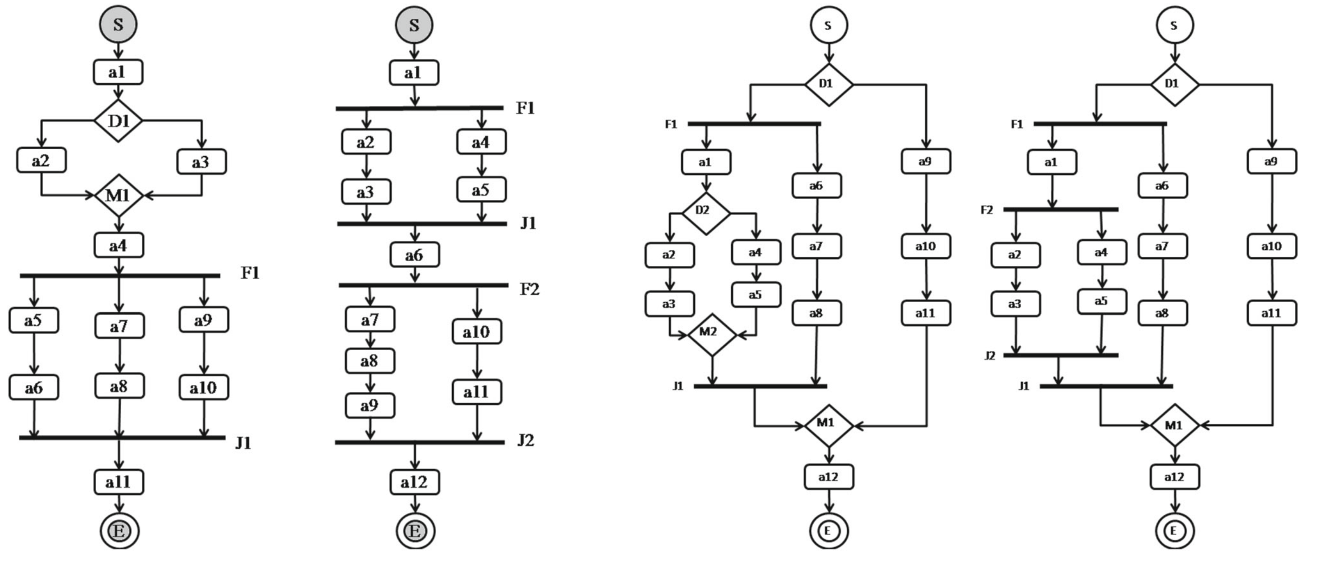
$$\left(\sum_{i=1}^{m} n_i\right)!/\prod_{j=1}^{m}(n_j!) \tag{1}$$

Nesting of different constructs defines a pattern, which is useful to analyze software systems. Xu et al. [27] presented a classification of concurrent activity diagrams based on the nesting inside a fork-join structure. Such classification is useful for understanding the complexity of activity diagrams, thereby estimating the testing effort. Their classification is based on nesting inside the fork-join structure; it can be further extended for non-concurrent and concurrent activity diagrams. In this paper, we have attempted to generalize their classification based on nesting and non-nesting (sequential) of decision/iterations and fork-join constructs.

2.1 Non-concurrent activity diagrams

Non-concurrent activity diagrams do not include fork-join structures. *Non-concurrent sequential activity diagrams* include decision and iteration structures in a sequential way. For example, a decision structure is followed by another decision structure or a decision structure is followed by iteration structure, and vice versa. *Non-concurrent nested activity diagrams* include nesting of decision and iteration structures. For example, a decision structure is a part of another higher-level decision structure or an iteration is a part of another higher-level decision structure, and vice versa.

In general, non-concurrent sequential activity diagrams have a number of distinct test scenarios equal to the multiplication of each decision/iteration outflow. In other words, if a sequence diagram has k decisions/iterations and each decision/iteration has p_i outflows, then the total number of distinct test scenarios (TS) is calculated as $TS = \prod_{i=1}^{k}(p_i)$ where $i \geq 2$. In nested non-concurrent activity diagrams, the total number of distinct paths is equal to the sum-



(a) SDF **(b)** SFF **(c)** NDFD **(d)** NDFF

Fig. 2 Concurrent activity diagrams: **a** SDF: sequential decision-fork activity diagram, **b** SFF: sequential fork-fork activity diagram, **c** NDFD: nested decision-fork-decision activity diagram, and **d** NDFF: nested decision-fork-fork activity diagram

mation of outflows achieved by unfolding the innermost decision/iteration to the outermost decision/iteration by considering uncounted outflows of higher level.

2.2 Concurrent activity diagrams

Concurrent activity diagrams include fork-join structures. *Concurrent sequential activity diagrams* include decision and fork-join structures in a sequential way. For example, a decision structure is followed by a fork-join structure or a fork-join structure is followed by another fork-join structure and vice versa. Figure 2a shows a sequential decision-fork activity diagram (SDF), and Fig. 2b shows a sequential fork-fork structure diagram (SFF). *Concurrent nested activity diagrams* include nesting of decision and fork-join structures. For example, a decision structure is a part of a higher-level fork-join structure or a fork-join structure is a part of another higher-level fork-join structure, and vice versa. Figure 2c shows a nested decision-fork-decision activity diagram (NDFD), and Fig. 2d shows a nested decision-fork-fork activity diagram (NDFF).

In general, concurrent sequential activity diagrams have a number of distinct test scenarios equal to the multiplication of test scenarios generated by each decision or iteration and fork-join constructs. In concurrent nested activity diagrams, the total number of distinct paths is calculated using the decomposition technique explained by Xu et al. [27] for simple nested fork-join structures.

3 Related work

A depth-first search (DFS)-based test scenario generation algorithm is commonly used for test scenario generation, e.g., [7,11,16,18]. The standard DFS algorithm generates basic paths from control flow graphs (CFGs). For generating valid concurrent test scenarios, there is a necessity to use a specific representation other than the control flow graph of an activity diagram or modify the DFS algorithm. In approaches suggested by [16,18], an activity diagram is represented in a Petri-net-like semantics. These approaches have used a DFS-based algorithm to generate test scenarios. In Kim et al. [11], an activity diagram is transformed into an Input-Output explicit Activity Diagram (IOAD). In Chandler et al. [7], the DFS algorithm is modified to generate test scenarios. In Yimman et al. [29], dynamic programming is used to generate test scenarios. In Walaithip et al. [25], the DFS algorithm is modified inside fork-join structure that Cartesian product of all activities is taken to generate all possible combinations of activities. In Kamonsantiroj et al. [10], a dynamic programming technique with Memoized-ConstPath algorithm is utilized to generate test scenarios.

Linzhang et al. [16] have presented the gray-box method to generate test cases from activity diagrams. Test scenarios are generated using a retrospective DFS algorithm which enumerates all basic paths from an activity diagram. Concurrent tasks inside fork-join structures are stored in a transition transformation relationship table to represent Petri-net-like semantics; this table facilitates the generation of the test scenarios. All the guard conditions along with the transitions of the test scenario are extracted as the path condition. For each test case, test data are generated using the category partition method. However, their algorithm supports only two concurrent threads inside a fork-join. Mingsong et al. [18] have presented an approach to generate test cases based on reverse engineering techniques using activity diagrams. Unlike other test case generation approaches, their approach first generates random test data for a Java program. Test cases (data) are executed on instrumented Java code to generate execution traces of the implementation. Execution traces are compared with the activity diagram's simple paths to reduce initially generated random test cases (data). Simple path coverage criterion restricts test scenarios to a single interleaved path. Although their approach processes concurrent constructs, it is not cost-effective due to initial random test data.

Kim et al. [11] have proposed a graph transformation-based approach for test case generation. Their approach is based on the IOAD flow graph. IOAD abstracts an activity diagram by suppressing internal actions and focuses on external input and output events, thus reducing the complexity of an activity diagram. Their approach suppresses non-external input and output to solve the state explosion problem. IO abstraction reduces the number of test cases generated in the presence of concurrency. However, it is not completely automated. Furthermore, their approach suppresses internal actions that may cause an omission of the system's concurrent behavior. Chandler et al. [7] have presented an approach to generate usage scenarios from activity diagrams. Their approach uses XML meta-data and stores meta-data in a vector form for the usage scenario generation. A modified depth-first algorithm is proposed to generate usage scenarios. However, their approach restricts the interleaving of activities inside fork-join. Kundu and Samanta [13] have presented an approach for generating test cases from activity diagrams at the use case scope. An activity diagram is transformed into an intermediate format called an activity graph. Test cases are generated using the DFS-BFS algorithm. For generating test cases, the DFS algorithm is used for the non-concurrent part and the BFS algorithm is used for the concurrent part of an activity diagram. Activity path coverage criterion is used in their approach. Their test cases help to uncover faults like synchronization and loop faults. Their approach allows the interleaving of activities inside a concurrent structure. However, the total number of concurrent paths generated are fewer and proportional to the factorial of the number of threads.

Yimman et al. [29] have applied a dynamic programming technique to mitigate the exponential interleaving of activities inside the fork-join structure. However, the $dynamicGenConPath$ algorithm needs the input of the threads un-widening tree to generate test scenarios. Walaithip et al. [25] used a modified DFS algorithm for fork-join structure where a Cartesian product of all activities of two threads produce a set of concurrent paths. Then, a precedence relation is applied to filter the required test cases. Both approaches, [29] and [25], will be applicable for activity diagrams with two threads and need to be extended for more than two threads.

Kamonsantiroj et al. [10] have proposed a dynamic programming technique with Memoized-ConstPath algorithm to generate concurrent activity test scenarios. This approach is an improvement over [29] and [25] to accept more than two threads. However, this approach handles single causality constraints for each run of the algorithm.

4 Test scenario generation techniques

One of the techniques to generate all possible interleaved test sequences from an activity diagram is to find all permutations. Among all permutation sequences, some permutations are valid interleaving test scenarios and some are invalid. Consequently, there is a necessity to check every permutation to accept it as a valid interleaving test scenario. As stated earlier, the permutation technique is exhaustive and exponential in size. In order to generate valid interleaving test scenarios, the permutation technique can be applied to a subset of activities inside the fork-join structure.

Graph-theoretic approaches use DFS and its variants like modified-DFS [7] and DFS-BFS [13] to generate interleaving test scenarios. Both approaches generate a limited number of test scenarios, which are in the order of the factorial of the number of threads. These approaches do not consider complexity due to the depth of threads, i.e., the number of activities per thread. In other words, if two threads consist of a single activity or n activities per thread, then the number of test scenarios generated is the same, i.e., two in this example.

Inside the fork-join construct, each thread is like a single sequential flow. A non-deterministic behavior happens only in case of a context switch. Due to the elapsed time in the timeshare system, synchronization, or causal order of events, DFS processes the CFG representation of an activity diagram. In such a scenario, DFS follows deterministic sequential flow in each thread. Although activities inside a fork-join structure follow the non-deterministic behavior, the execution of DFS inside CFG is sequential. Hence, DFS generates a fixed number of test scenarios.

We present two constrained permutation-based techniques, namely the level permutation algorithm and the DFS level permutation algorithm to generate valid interleaving test scenarios.

Test scenario generator framework: Test scenario generator framework is shown in Fig. 3. Object Management Group (OMG) has specified several modeling specifications for UML diagrams such as MOF, XMI and CWM. XML Metadata Interchange (XMI) was applied to transport UML

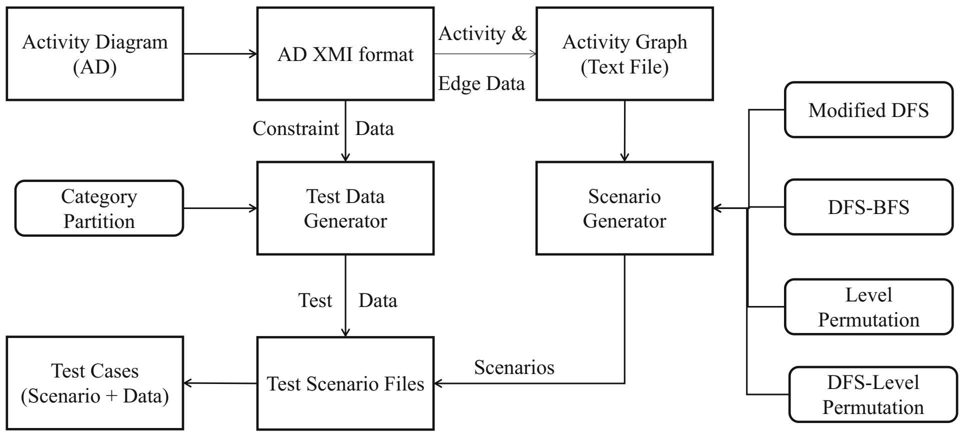


Fig. 3 Test scenario generation framework

models by generating a special XSD by using rules of XMI to the concrete UML meta-model. It is an open standard file format that enables the interchange of model information between models and tools. Various UML tools use proprietary XMI extensions to store the diagram information. Therefore, we prefer to extract the required activity diagram information from XMI to represent a CFG, which we call an activity graph. Similarly, we also extract information from XMI files regarding constraints for generating test data. Test scenario generators process an activity graph to generate test scenarios, as shown in Fig. 3.

4.1 Level permutation

The level permutation algorithm ($LevelPermute$) takes input from an activity graph and outputs a set of test scenarios. Algorithm 1 lists the $LevelPermute$ algorithm. In this algorithm, Function $getForkJoinActivities()$ returns activities inside a fork-join construct. All the activities inside the *fork-join* structure are divided into different levels. The immediate neighbor of the fork node activities is assigned to the first level, the immediate neighbors of the first level are assigned to the second level, and so on up to the join node; the join node is the end node. Function $assignLevelNumbers()$ assigns level numbers to activities and returns the highest level number. Then, the activities at each level are permuted to generate sub-sequences. Function $permutation()$ permutes activities and returns all permutations. Thereafter, each of the current level's sub-sequence is augmented with the previous level sub-sequences. The process of adding and appending sub-sequences to test scenarios set TS is given in line numbers from 7 to 26 of Algorithm 1. This includes adding first-level sub-sequences in the first iteration. The algorithm then increases the size of TS to accommodate all possible combinations of TS and new sub-sequences in each level. After that, sub-sequences of each level are appended to TS. Finally, this process returns a set of test scenarios inside the *fork-join* structure. This approach maintains the *happen before* relationship [14] among activities of threads.

Algorithm 1 Level Permutation algorithm (LevelPermute)

```
Input: $actGraph$: Activity Graph
Output: $TS$: Set of test scenarios
 1: $TS = \emptyset$;
 2: $forkJoinActivityList = getForkJoinActivities(actGraph)$;
 3: $levels = assignLevelNumbers(forkJoinActivityList)$;
 4: for ($i = 1 \rightarrow levels; i++$) do
 5:    $levelActivities = forkJoinActivityList.getActivitiesAtLevel(i)$;
 6:    $subSequencesList = permutation(levelActivities)$;
 7:    if ($TS == \emptyset$) then                    ▷ first time add paths to TS
 8:       for ($j = 1 \rightarrow subSequencesList.size(); j++$) do
 9:          $TS.add(subSequencesList.get(j))$;
10:       end for
11:    else     ▷ increase size of $TS$ in multiple of subsequence size
12:       $tSize = TS.size()$;
13:       $sSize = subSequencesList.size()$;
14:       $tsNewSize = (tSize * sSize) - tSize$;
15:       for ($p = 1 \rightarrow tsNewSize; p++$) do
16:          $sequence = TS.get(p$ mod $sSize)$;
17:          $TS.add(sequence)$;
18:       end for
19:  ▷ append every element in $subSequenceList$ to every element
     in $TS$
20:       for ($p = 1 \rightarrow TS.size(); p++$) do
21:          for ($q = 1 \rightarrow subSequencesList.size(); q++$) do
22:             $index = q + (p - 1) * TS.size()$;
23:             $TS.append(index, subSequencesList.get(q))$;
24:          end for
25:       end for
26:    end if
27: end for
28: return $TS$
```

We consider two examples as shown in Figs. 4 and 5. We illustrate the LevelPermute algorithm's working for generating a test scenario using these two examples. In the first example, Fig. 4 consists of two threads and six activities. Both threads have an equal number of activities, three in each thread. There are three levels, each consisting of two activities and generating two sub-sequences at each level. The concatenation of all sub-sequences generates eight

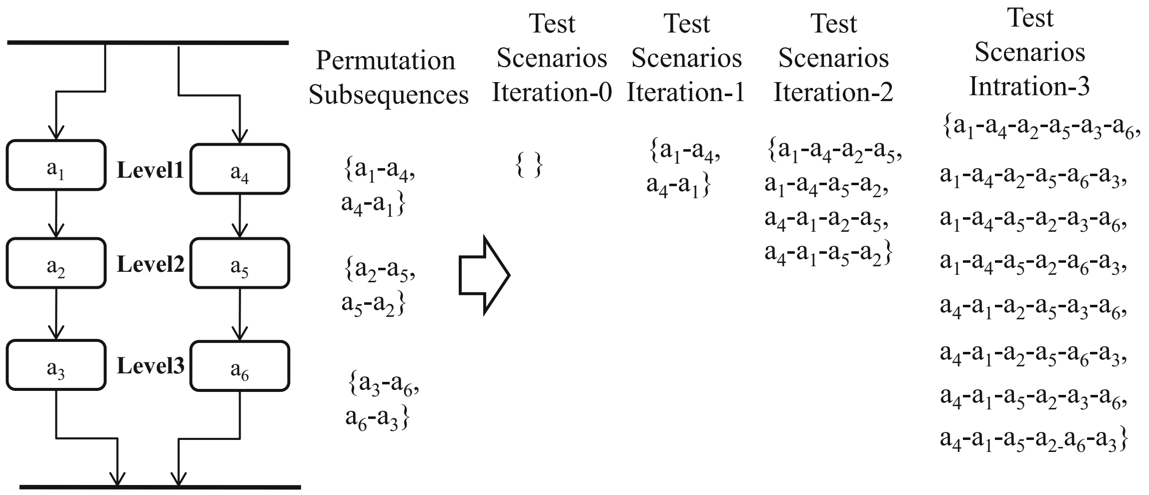

Permutation Subsequences
Test Scenarios Iteration-0
Test Scenarios Iteration-1
Test Scenarios Iteration-2
Test Scenarios Intration-3
a1
a4
Level1
a2
a5
Level2
a3
a6
Level3
{a1-a4, a4-a1}
{a2-a5, a5-a2}
{a3-a6, a6-a3}
{ }
{a1-a4, a4-a1}
{a1-a4-a2-a5, a1-a4-a5-a2, a4-a1-a2-a5, a4-a1-a5-a2}
{a1-a4-a2-a5-a3-a6, a1-a4-a2-a5-a6-a3, a1-a4-a5-a2-a3-a6, a1-a4-a5-a2-a6-a3, a4-a1-a2-a5-a3-a6, a4-a1-a2-a5-a6-a3, a4-a1-a5-a2-a3-a6, a4-a1-a5-a2.a6-a3}

Fig. 4 An illustration of the LevelPermute algorithm for the activity diagram with equal activities per thread

sequences, which are valid test scenarios inside the fork-join construct. The level permutation has generated $2!*2!*2! = 8$ sequences as compared to $6! = 720$ sequences generated by permutation; DFS-BFS generated 2! = 2 sequences.

In the first example, level 1 has two activities a_1 and a_4, level 2 has two activities a_2 and a_5, and level 3 has two activities a_3 and a_6, as shown in Fig. 4. During the execution of the LevelPermute algorithm, permutation of activities a_1 and a_4 results in $< a_1 - a_4,\ a_4 - a_1 >$. Similarly, the permutation at level 2 and level 3 is $< a_2 - a_5,\ a_5 - a_2 >$ and $< a_3 - a_6,\ a_6 - a_3 >$, respectively. The algorithm executes in three iterations as there are three levels, as shown in Fig. 4. In the first iteration, the algorithm adds two permuted sub-sequences to the test-set TS; $TS =< a_1 - a_4,\ a_4 - a_1 >$. In the second iteration, the algorithm first increases TS size to four and replicates TS's previous two sub-sequences. After that, the algorithm appends the second level permutation sub-sequences $< a_2 - a_5,\ a_5 - a_2 >$ to each sequence in TS. At this stage, $TS =< a_1 - a_4 - a_2 - a_5,\ a_4 - a_1 - a_2 - a_5,\ a_1 - a_4 - a_5 - a_2,\ a_4 - a_1 - a_5 - a_2 >$. In the third iteration, the algorithm increases TS's size to eight and replicates TS's previous four sequences. Next, the algorithm appends third level permutation sub-sequences to each sequence in TS, generating a total of eight sequences at the end of the algorithm, as illustrated in Fig. 4. In this way, $TS =< a_1 - a_4 - a_2 - a_5 - a_3 - a_6,\ a_1 - a_4 - a_2 - a_5 - a_6 - a_3,\ a_1 - a_4 - a_5 - a_2 - a_3 - a_6,\ a_1 - a_4 - a_5 - a_2 - a_6 - a_3,\ a_4 - a_1 - a_2 - a_5 - a_3 - a_6,\ a_4 - a_1 - a_2 - a_5 - a_6 - a_3,\ a_4 - a_1 - a_5 - a_2 - a_3 - a_6,\ a_4 - a_1 - a_5 - a_2 - a_6 - a_3 >$.

In the second example, we have taken non-equal number of activities in each thread as shown in Fig. 5. The figure consists of three threads and six activities. The first thread has three activities, the second thread two activities, and the third thread a single activity. Thus, there are three levels, each consisting of three, two, and one activity, respectively. At each level, there are six, two, and one sub-sequence generated, respectively. Finally, the concatenation of all these sub-sequences generates twelve sequences. These are valid test scenarios inside the fork-join construct. Thus, the $LevelPermute$ algorithm has generated $3! * 2! * 1! = 12$ sequences, whereas the permutation had generated $6! = 720$ sequences; DFS-BFS generated 2!=2 sequences only.

Performance analysis: The performance of test scenario generator can be measured in terms of the number of valid test scenarios generated. The total number of test scenarios generated by the LevelPermute algorithm is calculated as follows. Let there be n levels in an activity diagram. If level L_i, $1 \le i \le n$, has $|L_i|$ activities, then level L_i has $|L_i|!$ sub-sequences. Let CTS be the total number of concurrent test scenarios. Then, the total number of valid interleaving test scenarios generated by the level permutation technique inside the *fork-join* with n levels is given by:

$$CTS(LevelPermute) = \prod_{i=1}^{n} |L_i|! \tag{2}$$

One of the most commonly used techniques to generate interleaving test scenarios is the BFS technique inside the fork-join construct for activity diagrams. In comparison with the BFS algorithm, the LevelPermute algorithm generates a larger number of interleaving test scenarios. The total number of valid interleaving test scenarios generated by BFS inside the fork-join with m threads and n levels is $CTS(BFS) = m!$. The permutation algorithm generates the total number of sequences for the fork-join construct with m threads, n levels, and k activities of all threads is $CTS(Permutation) = k!$. The following equation holds for the BFS, level permutation, and permutation algorithms inside the *fork-join* construct.

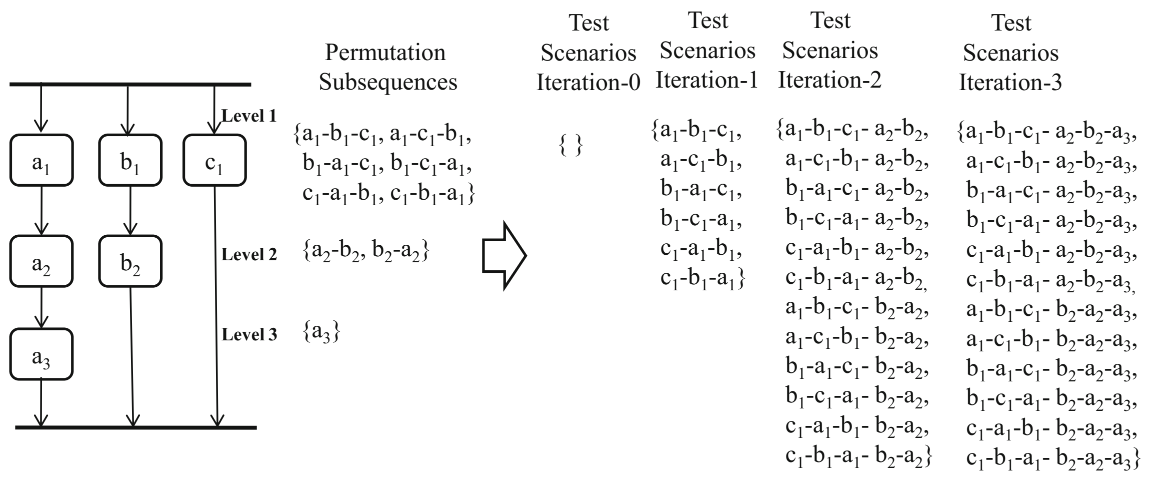

Level 1
Level 2
Level 3
Permutation Subsequences
{a1-b1-c1, a1-c1-b1, b1-a1-c1, b1-c1-a1, c1-a1-b1, c1-b1-a1}
{a2-b2, b2-a2}
{a3}
Test Scenarios Iteration-0
{ }
Test Scenarios Iteration-1
{a1-b1-c1, a1-c1-b1, b1-a1-c1, b1-c1-a1, c1-a1-b1, c1-b1-a1}
Test Scenarios Iteration-2
{a1-b1-c1- a2-b2, a1-c1-b1- a2-b2, b1-a1-c1- a2-b2, b1-c1-a1- a2-b2, c1-a1-b1- a2-b2, c1-b1-a1- a2-b2, a1-b1-c1- b2-a2, a1-c1-b1- b2-a2, b1-a1-c1- b2-a2, b1-c1-a1- b2-a2, c1-a1-b1- b2-a2, c1-b1-a1- b2-a2}
Test Scenarios Iteration-3
{a1-b1-c1- a2-b2-a3, a1-c1-b1- a2-b2-a3, b1-a1-c1- a2-b2-a3, b1-c1-a1- a2-b2-a3, c1-a1-b1- a2-b2-a3, c1-b1-a1- a2-b2-a3, a1-b1-c1- b2-a2-a3, a1-c1-b1- b2-a2-a3, b1-a1-c1- b2-a2-a3, b1-c1-a1- b2-a2-a3, c1-a1-b1- b2-a2-a3, c1-b1-a1- b2-a2-a3}

Fig. 5 An illustration of the LevelPermute algorithm for the activity diagram with non-equal number of activities per level

$$CTS(BFS) \leq CTS(LevelPermute) \leq CTS(Permutation) \tag{3}$$

4.2 DFS level permutation

This subsection describes the DFS level permutation (DFS-LevelPermute) algorithm. The BFS and LevelPermute algorithms generate valid concurrent test scenarios inside the *fork-join* structure. In most applications, a *fork-join* structure is surrounded by other structures too. A test scenario is complete if it has first activity as the start activity and last activity as the end activity of an activity diagram. In other words, a complete test scenario also includes activities outside the fork-join structure for testing purposes. Therefore, we have proposed the DFS-LevelPermute algorithm to generate complete and valid test scenarios. The proposed algorithm is a combination of the DFS and LevelPermute algorithms.

The DFS algorithm is used for non-concurrent parts of activity diagrams, while the LevelPermute algorithm is used for concurrent parts of activity diagrams inside *fork-join*. In the proposed algorithm, the LevelPermute algorithm is first executed to generate concurrent test scenarios (CTS) inside the *fork-join* structure. Then, the standard DFS algorithm traverses the activity diagram from the start node to the end node to generate the valid paths. After that, the prefix and suffix are separated from each path that includes fork-join. A prefix is a portion from the start node to the fork node. A suffix is a portion from the join node to the end node of the path. Finally, each CTS sequence is concatenated (sandwiched) between the suffix and the prefix of the path to generate all the complete and valid concurrent test scenarios. Figure 6 represents an illustration of the DFS-LevelPermute algorithm. The LevelPermute algorithm generates four sequences from activities a_2, a_3, a_4, a_5 and the DFS algorithm generates two paths. Then, their prefix $S - a_1 - F$ and suffix $J - a_6 - E$ are separated. Finally, all sequences from CTS are concatenated between prefix and suffix to generate four additional distinct test scenarios.

5 Experimental results

We implemented the above algorithms to automate the process of test case generation. Test scenarios are generated using the DFS-LevelPermute algorithm. Test data are generated using the category partition method for each test scenario. A test scenario and the associated test data together form a test case for integration level and system-level testing. We have conducted all the experiments using 3.00 GHz Intel Core 2 Quad CPU with a 4 GB RAM desktop computer. Test sets are generated from activity diagrams. For this, first an activity diagram is translated to an intermediate representation (IR), a CFG-like structure, which we have named as an 'Activity graph'. Then, we have generated test scenarios from this IR using the modified DFS, DFS-BFS, CQS, and DFS-LevelPermute algorithms. The proposed algorithms are evaluated using concurrent activity diagrams from a few test-bed and application scenarios. A test-bed of four different types of activity diagrams is chosen from Fig. 2.

We implemented the modified-DFS [7], DFS-BFS [13], CQS [21], and the proposed DFS-Level permute algorithms to generate valid concurrent test scenarios for a fair comparison. Basic path coverage [16], simple path coverage [18], and interleaving activity path coverage [22] criteria are used to evaluate coverage metrics of test scenarios generated by the above algorithms. The results are shown in Table 2, which contains the description of activity diagram, namely name, maximum valid paths, total generated paths, basic path coverage, simple path coverage, and interleaving activity path coverage as obtained by the DFS-BFS, the modified-DFS,

Fig. 6 An illustration of the DFS Level Permutation Algorithm

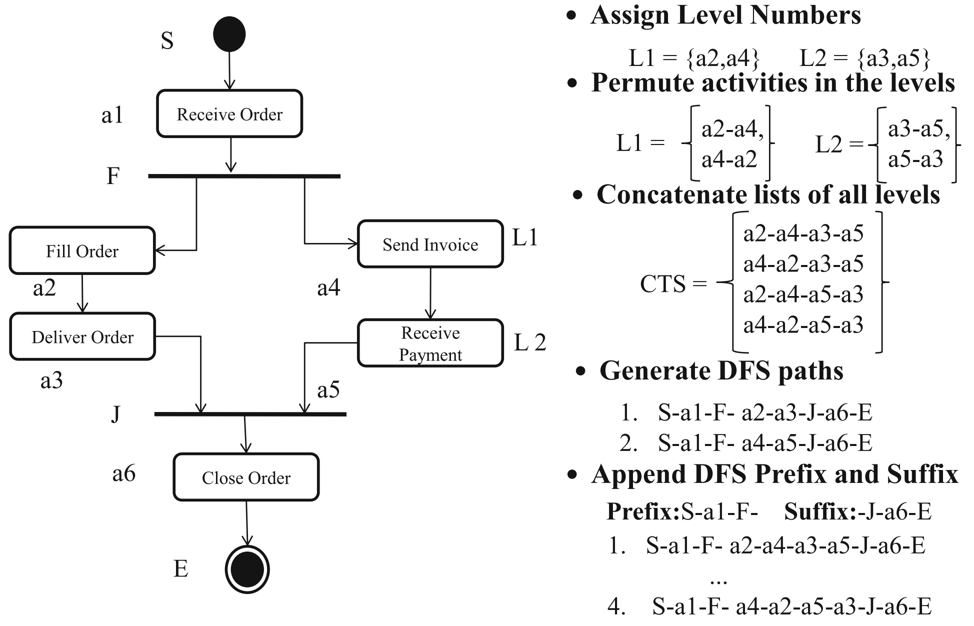

S
a1
Receive Order
F
Fill Order
a2
Send Invoice
L1
a4
Deliver Order
a3
Receive Payment
L 2
a5
J
a6
Close Order
E
• Assign Level Numbers
L1 = {a2,a4} L2 = {a3,a5}
• Permute activities in the levels
L1 = {a2-a4, a4-a2} L2 = {a3-a5, a5-a3}
• Concatenate lists of all levels
CTS = {a2-a4-a3-a5, a4-a2-a3-a5, a2-a4-a5-a3, a4-a2-a5-a3}
• Generate DFS paths
1. S-a1-F- a2-a3-J-a6-E
2. S-a1-F- a4-a5-J-a6-E
• Append DFS Prefix and Suffix
Prefix:S-a1-F- Suffix:-J-a6-E
1. S-a1-F- a2-a4-a3-a5-J-a6-E
...
4. S-a1-F- a4-a2-a5-a3-J-a6-E

Table 2 Algorithm, total paths generated, basic path coverage, simple path coverage, and interleaving activity path coverage (IAPC) for concurrent activity diagrams

Diagram	Algorithm	Total	Coverage (%)		
(Max. paths)		paths	basic path	simple path	IAPC
SDF (180)	DFS-BFS	12	NA	100.0	6.7
	Modified DFS	6	NA	100.0	3.3
	CQS	6	NA	100.0	3.3
	DFS-LevelPermute	72	NA	100.0	40.0
SFF (60)	DFS-BFS	4	NA	100.0	6.7
	Modified DFS	4	NA	100.0	6.7
	CQS	4	NA	100.0	6.7
	DFS-LevelPermute	16	NA	100.0	26.7
NDFD (56)	DFS-BFS	5	100.0	100.0	8.9
	Modified DFS	5	100.0	100.0	8.9
	CQS	5	100.0	100.0	8.9
	DFS-LevelPermute	17	100.0	100.0	30.4
NDFF (720)	DFS-BFS	13	100.0	100.0	1.8
	Modified DFS	13	100.0	100.0	1.8
	CQS	13	100.0	100.0	1.8
	DFS-LevelPermute	49	100.0	100.0	6.8

CQS, and DFS-LevelPermute algorithms. These algorithms were tested on the SDF, SFF, NDFD, and NDFF activity diagrams, as shown in Fig. 2.

SDF and SFF diagrams do not include basic (non-interleaving) paths; therefore, the basic path coverage for them was not calculated. NDFD and NDFF diagrams include one non-interleaving (basic) path. All the algorithms, considered in this work, generated basic paths that lead to 100% basic path coverage for NDFD and NDFF diagrams. All algorithms also achieved 100% simple path coverage for all concurrent activity diagrams. For concurrent activity diagrams, one representative interleaving activity path with a distinct set of activities achieves 100% simple path coverage. Table 2 shows that all the algorithms have generated at least one concurrent test scenario. It was observed that exploring execution interleavings in nested activity diagrams is harder than in sequential activity diagrams. For concurrent nested activity diagrams, it is not possible to generate interleaving paths directly, hence achieving interleaving activity path coverage on original diagrams is difficult. But after the decomposition of these diagrams into simple concurrent activity diagrams, as suggested in Xu et al., all the algo-

Table 3 Application scenario activity diagram's attributes: total number of activities, basic paths, simple paths, and interleaving activity paths (IAP)

App. No.	Activity diagram	# Activities	# Basic paths	#Simple paths	# IAP paths
App 1	Airport check in	13	01	02	07
App 2	ATM	21	03	02	14
App 3	Order processing	18	02	02	30
App 4	Graphics utility	14	00	01	56

Table 4 Total valid test scenarios generated, percentage coverage of basic path (BPC), simple path (SPC), and interleaving activity path (IAPC) by DFS-BFS, modified-DFS, CQS, and DFS-LevelPermute algorithms for application scenarios

App. No.	Path	Coverage (%)			Path	Coverage (%)		
		BPC	SPC	IAPC		BPC	SPC	IAPC
	DFS-BFS				Modified DFS			
App 1	05	100.0	100.0	62.5	05	100.0	100.0	62.5
App 2	07	100.0	100.0	41.2	07	100.0	100.0	41.2
App 3	06	100.0	100.0	18.8	06	100.0	100.0	18.8
App 4	02	NA	100.0	03.6	02	NA	100.0	03.6
	CQS				DFS-LevelPermute			
App 1	05	100.0	100.0	62.5	05	100.0	100.0	62.5
App 2	07	100.0	100.0	41.2	07	100.0	100.0	41.2
App 3	06	100.0	100.0	18.8	10	100.0	100.0	31.3
App 4	02	–	100.0	03.6	06	–	100.0	10.7

rithms have generated valid concurrent test scenarios. The results indicated the nature of interleaving activity path coverage after the decomposition of diagrams as shown in the table. It was also observed that the DFS-LevelPermute test scenario generation algorithm achieved better interleaving activity path coverage for sequential concurrent activity diagrams and nested concurrent activity diagrams.

To verify the effectiveness of the proposed algorithm for a few real application scenarios, we have chosen four application scenarios, namely automatic teller machine (ATM) money withdrawal, airport check-in, order processing, and graphics utility. All these activity diagrams depict different aspects of the design. Their attributes are listed in Table 3. The data source and source code of these applications are made available at *GitHub* repository[1].

Table 4 shows the experimental results for all the above considered application scenarios. The DFS-LevelPermute algorithm achieved an increase in interleaving activity path coverage up to 35% for testbed activity diagrams and up to 12% for other application scenarios over other contemporary algorithms. For applications APP1 and APP2, the number of paths generated by each algorithm is the same though they generated different test scenarios. It was also observed that the DFS-BFS and the DFS-LevelPermute algorithms generated similar test scenarios for a single level. For applications APP3 and APP4, the DFS-LevelPermute algorithm achieved high interleaving activity path coverage as compared to the other algorithms. Thus, the proposed level permutation-based algorithms generated a higher number of test scenarios, thereby achieving higher interleaving activity path coverage. This increases the probability of detecting larger concurrency errors.

6 Conclusions

Previous graph theoretic-based approaches for concurrent test scenario generation did not consider the complexity of a concurrent activity diagram, and thus, such approaches generated a fewer test scenarios for concurrency testing. To mitigate this problem, this paper presented two constrained permutation-based test scenario generation algorithms from concurrent activity diagrams for achieving the higher interleaving activity path coverage. Test scenarios generated using the proposed LevelPermute algorithm were found useful for parallel and concurrent computation for testing structural and concurrency errors. This algorithm generated a higher number of test scenarios as compared to the other contemporary algorithms. Experimental results indicated that the proposed DFS-LevelPermute algorithm improves interleaving activity path coverage up to 35% for concurrent activity diagrams. The proposed approaches were shown to work for small- to medium-sized applications. The current ongoing work is aimed at scaling up the proposed techniques for larger activity diagrams.

Acknowledgements The major part of this work was done, while the authors were at Indian Institute of Technology (IIT) Kharagpur. The

[1] Data Source: https://github.com/MaheshShirole/LevelPermutation

authors thank the reviewers for their valuable comments. The authors are also thankful to the Editor-in-Chief, the Editor, and the Editorial Office Assistant(s) for managing this manuscript.

References

1. Ahmad T, Iqbal J, Ashraf A, Truscan D, Porres I (2019) Model-based testing using UML activity diagrams: a systematic mapping study. Comput Sci Rev 33:98–112. https://doi.org/10.1016/j.cosrev.2019.07.001
2. Albert M, Atkinson M (2005) Simple permutations and pattern restricted permutations. Discrete Math 300(1–3):1–15. https://doi.org/10.1016/j.disc.2005.06.016
3. Anbunathan R, Basu A (2019) Combining genetic algorithm and pairwise testing for optimised test generation from UML ADs. IET Softw 13:423–433. https://doi.org/10.1049/iet-sen.2018.5207
4. Atkinson M (1999) Restricted permutations. Discrete Math 195(13):27–38. https://doi.org/10.1016/S0012-365X(98)00162-9
5. Beizer B (1990) Software Testing Techniques, 2nd edn. Van Nostrand Reinhold Co., New York
6. Cartaxo E, Neto F, Machado P (2007) Test case generation by means of UML sequence diagrams and labeled transition systems. In: Proceedings of the IEEE international conference on systems, man and cybernetics, pp 1292–1297 (2007). https://doi.org/10.1109/ICSMC.2007.4414060
7. Chandler R, Lam CP, Li H (2005) AD2US: An automated approach to generating usage scenarios from UML activity diagrams. In: Proceedings of the 12th Asia-Pacific software engineering conference, pp 9–16 https://doi.org/10.1109/APSEC.2005.25
8. Dinh-Trong T, Ghosh S, France R (2006) A systematic approach to generate inputs to test UML design models. In: Proceedings of the 17th international symposium on software reliability engineering, pp 95–104 . https://doi.org/10.1109/ISSRE.2006.10
9. Dinning A, Schonberg E (1991) Detecting access anomalies in programs with critical sections. ACM SIGPLAN Notices 26(12):85–96. https://doi.org/10.1145/127695.122767
10. Kamonsantiroj S, Pipanmaekaporn L, Lorpunmanee S (2019) A memorization approach for test case generation in concurrent UML activity diagram. In: Proceedings of the 2nd international conference on geoinformatics and data analysis, pp 20–25. ACM Press https://doi.org/10.1145/3318236.3318256
11. Kim H, Kang S, Baik J, Ko I (2007) Test cases generation from UML activity diagrams. In: Proceedings of the 8th ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing, pp 556–561 https://doi.org/10.1109/SNPD.2007.525
12. Knuth DE (2005) The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations. Addison-Wesley Professional, Boston
13. Kundu D, Samanta D (2009) A novel approach to generate test cases from UML activity diagrams. J Object Technol 8(3):65–83. https://doi.org/10.5381/jot.2009.8.3.a1
14. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565. https://doi.org/10.1145/359545.359563
15. Lima L, Tavares A, Nogueira SC (2020) A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP. Sci Comput Program. https://doi.org/10.1016/j.scico.2020.102497
16. Linzhang W, Jiesong Y, Xiaofeng Y, Jun H, Xuandong L, Guoliang Z (2004) Generating test cases from UML activity diagram based on gray-box method. In: Proceedings of the 11th Asia-Pacific software engineering conference, pp 284–291 https://doi.org/10.1109/APSEC.2004.55
17. Lu S, Park S, Zhou Y (2012) Finding atomicity-violation bugs through unserializable interleaving testing. IEEE Trans. Software Engineering 38(4):844–860. https://doi.org/10.1109/TSE.2011.35
18. Mingsong C, Xiaokang Q, Xuandong L (2006) Automatic test case generation for UML activity diagrams. In: Proceedings of the international workshop automation of software Test, pp 2–8. ACM https://doi.org/10.1145/1138929.1138931
19. OMG: UML superstructure v2.4.1 (2011). http://www.omg.org/spec/UML/2.4/Superstructure/PDF/
20. Shirole M, Kommuri M, Kumar R (2012) Transition sequence exploration of UML activity diagram using evolutionary algorithm. In: Proceedings of the 5th India software engineering conference, pp 97–100. ACM https://doi.org/10.1145/2134254.2134271
21. Shirole M, Kumar R (2012) Testing for concurrency in UML diagrams. SIGSOFT Softw Eng Notes 37(5):1–8. https://doi.org/10.1145/2347696.2347712
22. Shirole M, Kumar R (2021) Concurrency coverage criteria for activity diagrams. IET Software 15(1):43–54. https://doi.org/10.1049/sfw2.12009
23. Sun C, Zhang B, Li J (2009) TSGen: A UML activity diagram-based test scenario generation tool. In: Proceedings of the international conference on computational science and engineering, Vol 2, pp 853–858 https://doi.org/10.1109/CSE.2009.99
24. Süß M, Leopold C (2008) Common mistakes in OpenMP and how to avoid them: A collection of best practices. In: Proceedings of the International Conference on OpenMP shared memory parallel programming, pp 312–323. Springer-Verlag. https://doi.org/10.1007/978-3-540-68555-5_26
25. Thanakorncharuwit W, Kamonsantiroj S, Pipanmaekaporn L (2016) Generating test cases from UML activity diagram based on business flow constraints. In: Proc. 5th Int. Conf. Network, Communication & Computing, p. 155–160. ACM, New York, NY, USA https://doi.org/10.1145/3033288.3033311
26. Vaziri M, Tip F, Dolby J (2006) Associating synchronization constraints with data in an object-oriented language. ACM SIGPLAN Notices 41(1):334–345. https://doi.org/10.1145/1111320.1111067
27. Xu D, Li H, Lam CP (2005) Using adaptive agents to automatically generate test scenarios from the UML activity diagrams. In: Proceedings of the 12th Asia-Pacific software engineering conference, pp 385–392. IEEE Computer Society https://doi.org/10.1109/APSEC.2005.110
28. Yamada Y, Iwasaki H, Ugawa T (2011) SAW: Java synchronization selection from lock or software transactional memory. In: Proceedings of the 17th international conference on parallel and distributed systems, IEEE, pp 104–111 https://doi.org/10.1109/ICPADS.2011.112
29. Yimman S, Kamonsantiroj S, Pipanmaekaporn L (2017) Concurrent test case generation from UML activity diagram based on dynamic programming. In: Proceedings of the 6th international conference software and computer applications, pp 33–38. ACM Press https://doi.org/10.1145/3056662.3056699