Journal of Electronic Testing (2023) 39:71–88
https://doi.org/10.1007/s10836-023-06045-y

DFS-KeyLevel: A Two-Layer Test Scenario Generation Approach for UML Activity Diagram

Xiaozhi Du[1] · Jinjin Zhang[1] · Kai Chen[1] · Yanrong Zhou[1]

Received: 29 September 2022 / Accepted: 21 January 2023 / Published online: 3 February 2023

Abstract
For automatic generation of test scenarios from UML (Unified Modeling Language) activity diagrams (ADs) are very important for improving test efficiency. However, state-of-the-art approaches mainly focus on simple approaches, without specifically considering the case of concurrent activity, which may result in the path explosion problem during the generation of test scenarios. In this paper, we put forward DFS-KeyLevel, a two-layer test scenario generation approach for UML Activity Diagram. First, the ADs of the software under test are modeled and preprocessed, and each concurrent module in each AD is simplified to a composite node. Then, primary test scenarios are generated from the concurrent activity modules using our proposed KeyLevel method. Next, the high-layer test scenarios are generated from the simplified AD with our improved Depth-First Search (DFS) algorithm. Finally, the primary and high-layer test scenarios are combined to generate the final test scenarios for the AD. The experimental results show that this DFS-KeyLevel is superior to the previous approaches. The DFS-KeyLevel can generate more test scenarios under constraints. Compared with DFS-LevelPermutes, the number of test scenarios generated by our DFS-KeyLevel is 1.13 times higher. Compared with Depth-First Search and Breadth-First Search (DFS-BFS) and Improved-DFS (IDFS), the DFS-KeyLevel produced 2.37 times test scenarios. The average coverage rates of staggered activities and total activity logical path coverage (TALPC) of the DFS-KeyLevel are 83.67% and 84% respectively, which is significantly higher than the above three approaches. In addition, when our method is applied to a real embedded system, it significantly reduces test scenarios generated to avoid path explosion while ensuring enough test scenarios.

Keywords UML activity diagram · Test scenario generation · Concurrent module · Embedded system

1 Introduction

Developing high-quality software requires a great deal of work in the testing phase, which is probably the most costly and labor-intensive part of the software development process. To improve the test efficiency, Model-Based Test (MBT) has attracted a lot of researchers and engineers. As the abstract of real systems, models formulate the relationships between the behavior and the behavior or between the behavior and the system, thus facilitating the understanding of systems [1]. MBT is a such black-box testing technique that generates tests from abstract behavioral models. It allows to automate or semi-automate the entire testing process. So MBT has many benefits including reduced cost, reduced time, and improved test quality.

The main objective of the test scenario generation method is to generate as many test scenarios with high coverage as feasible. Therefore, we can deeply understand the system under test (SUT), expose more defects of the system under test, improve the robustness of the software, and make the software better meet the users' needs. Test scenarios are the basis of test cases. Test scenarios with high coverage mean that test cases derived from them have higher coverage and can test the SUT more comprehensively. Test scenario generation should comply with the following specifications: the generated test scenarios meet the requirements of the SUT; for each requirement,

Responsible Editor: V. D. Agrawal.

✉ Xiaozhi Du
xzdu@xjtu.edu.cn

Jinjin Zhang
765814257@qq.com

Kai Chen
1171319431@qq.com

Yanrong Zhou
Zhouyr@stu.xjtu.edu.cn

[1] School of Software Engineering, Xi'an Jiaotong University, Shaanxi, China

the generated test scenarios can identify user actions and objectives; the generated test scenarios have feasibility, reliability and high coverage.

The UML Activity Diagram (AD) is an important diagram for modeling the dynamic aspects of a system [17]. Recently, testing based on UML AD has attracted the attention of researchers. Using UML language specification has changed the development methods of requirements, design and implementation. And testing can also be done at the requirement level to find errors earlier [8, 12, 16, 25]. The AD in UML is used to model the interaction between users and software, which can visually show the execution sequence between activities [14]. So, UML AD has advantage in simplifying complex systems. For example, the embedded system is integrated with hardware and software together, which results a complex system. Using UML to model the embedded system can standardize the analysis and design and facilitate testing.

More importantly, UML AD can better represent the dynamic behavior and state of the concurrent system. However, most researchers only study how to automatically generate test scenarios in simple systems, and do not explore the in-depth research on complex modules such as nested concurrent activity modules or nested loop activity modules. If this part is not handled well, the path explosion will occur, and the generated test scenarios will have errors, redundancies and other problems. Therefore, how to avoid the path explosion of concurrent and loop activities in complex systems and generate test scenarios that meet the coverage criteria is the key issue of this paper.

To address the above issues, we put forward DFS-KeyLevel, a two-layer test scenario generation approach for UML AD. First, the AD of the software under test are modeled, then each AD is preprocessed and converted into a control flow graph (CFG), in which each concurrent module is simplified to a composite node. Next, primary test scenarios are generated from the concurrent activity modules using our proposed KeyLevel algorithm. Then, the high-layer test scenarios are generated from the simplified AD with our improved DFS algorithm. Finally, the primary and high-layer test scenarios are combined to generate the final test scenarios for the AD.

The main contributions of this paper can be summarized as follows:

1. We proposed an innovative two-layer test scenario generation approach for UML ADs, DFS-KeyLevel. For the concurrent activity modules, we put forward KeyLevel methods to generate primary test scenarios. For the simplified AD, we present an improved DFS algorithm to generate high-layer test scenarios.
2. Our DFS-KeyLevel approach can efficiently deal with concurrent activities, which avoids path explosion and meets high coverage. For simple concurrent activity modules, which only contain sequential structures, we put forward the KeyLevel-Simple method to efficiently generate primary test scenarios. For complex concurrent activity modules containing sequential, branch and loop structures, which cannot be dealt with by state-of-the-art approaches, we proposed the KeyLevel-Complex method to generate correct and complete primary test scenarios.
3. We applied the DFS-KeyLevel approach to an actual embedded system, in which a concurrent activity module theoretically has 1,630,423,080 test scenarios. The KeyLevel-Complex method was used and only 217,824 test scenarios are generated, which has a 99.887% reduction rate while ensuring to generate enough test scenarios.

The paper is organized as follows. Section 2 reviews the related work and discusses the unresolved issues. Section 3 describes our DFS-KeyLevel approach in detail. In Section 4, some experiments are taken and the results are analyzed. Lastly, Section 5 concludes this paper and discusses some future work.

2 Related Work

Nowadays, many researchers use the AD to automatically generate test scenarios, which can reduce the time and cost of testing [7]. The business process and concurrent behavior of software development can be demonstrated by AD [4], which is convenient to model the concurrent activity modules. But the concurrent activity is synchronous so that huge test scenarios will be generated in the test process. To avoid path explosion, researchers put forward graph theory technology, machine learning and other technologies.

At present, more and more heuristic algorithms are applied in the field of test scenario generation. Li and Lam [13] proposed an anti-ant colony algorithm, which is used to generate test threads from UML, and then generate test scenarios through test threads. Jena et al. [9] proposed a test scenario generation method using the Genetic Algorithm (GA). This method takes activity coverage as the criterion to generate test scenarios, but it failed to realize the automatic generation of test scenarios. Anbunathan and Basu [2] proposed using the GA and the paired test method to generate test scenarios under concurrent ADs. This approach is mainly used to reduce the test scenarios of concurrent ADs to avoid path explosion, but it may lead to the omission of some necessary scenarios. Arora et al. [3] proposed a direction-based ant colony algorithm, which improved the pheromone calculation method in the ant colony algorithm by adding cosine value to increase randomness and avoid falling into a local optimal solution, thus generating all test scenarios. However, due to the redundancy of the generated test scenarios during the execution of the algorithm, the time complexity is high.

Besides heuristic algorithms, BFS and DFS algorithms, which are often used by researchers, are constantly improved. Kundu and Samanta [11] proposed improved BFS and DFS approaches to generate test paths, and proposed the active path coverage standard, which can detect more faults. However, the test scenarios generated by this method are limited. Thanakorncharuwit et al. [26] proposed an improved DFS approach with test specifications to generate test scenarios. The algorithm improves DFS by backtracking technology, which makes the process of test scenario generation easier and reduces the test workload, but the applied case system is relatively simple. Shirole and Kumar [24] put forward DFS-LevelPermute approach to avoid path explosion, by dividing the horizontal hierarchy according to the node order of the AD. However, this hierarchical division is more based on the activity model structure, which is not rigorous, and the generated test scenarios have a lot to do with the model structure. Panthi et al. [18] proposed to combine BFS algorithm and DFS algorithm to generate test scenarios. This approach is easy to understand, but the number of test scenarios generated for concurrent modules is limited. The IDFS approach proposed by Fan et al. [5] generates limited test scenarios, so the test coverage is relatively low.

Dynamic programming algorithm can also be used for test scenario generation. Yimman et al. [29] proposed a dynamic programming algorithm to generate test scenarios. By specifying the sequence between threads, a limited number of test scenarios are generated. Kamonsantiroj et al. [10] proposed a method which is an improvement of the method proposed by Yimman et al. [29], and the proposed method of dynamic programming to generate test scenarios can be suitable for a concurrent system including more than 3 thread, which effectively avoids the path explosion. But the test scenarios generated by this method are inadequate, which may lead to the omission of some necessary scenarios.

ADs can also be combined with other UML diagrams to generate test scenarios. Shirole and Kumar [22] put forward a search algorithm (CQS), which generates test scenarios by converting sequence diagrams into ADs, and uses CQS to deal with the randomness of concurrent tasks. This algorithm can avoid data competition, synchronization and deadlock, but the number of test scenarios generated is small. Mahali et al. [15] proposed to use IOAD model to generate test scenarios, which can eliminate deadlock in concurrency to avoid path explosion. However, the generated test scenarios contain redundant nodes, and the proposed method is only applicable to IOAD model, so it is not universal and cannot be applied to UML model. Salman and Yasir [20] put forward a new coverage standard to evaluate the test scenarios generated from UML state diagrams. This coverage standard can improve the effectiveness of the generated test scenarios, especially for loop processing, but it does not deal with concurrent activities. Hamza and Hammad [6] put forward a test scenario generation method based on UML use case diagram. This method takes the use case diagram as input and converts it into AD in the process of processing, and then generates test scenarios from the AD. However, the application system is relatively simple, and there is no in-depth research on complex systems. Shi et al. [21] put forward a test scenario generation method based on UML AD syntax, which can check the correctness of AD structure and automatically generate test scenarios according to user constraints, but the generated scenarios are limited.

To sum up, state-of-the-art approaches solve some issues on generating test scenarios automatically, but these approaches still have the following shortcomings:

1. Some of them are only applicable to models of specific structures or models;
2. Some of them don't conduct sufficient experiments and even have been experimented on simple systems only;
3. Some of them generate very limited test scenarios, which may not meet the testing needs or even miss important test scenarios.

Our proposed approach makes up for above shortcomings well. It is applicable to all UML ADs and has been experimented on both simple and complex systems with good results. More importantly, our approach can generate as many test scenarios as possible while avoiding path explosion.

3 Proposed DFS-KeyLevel Approach

To generate test scenarios for UML ADs, we propose a two-layer test scenario generation approach, called DFS-KeyLevel, which consists of two main steps and is depicted in Fig. 1. At the first step, the software under test (SUT) is first modeled and its behaviors are described with UML ADs, and then each AD is converted into a CFG according to the mapping rules. At the second step, test scenarios are generated with our proposed DFS-KeyLevel, which includes four sub-steps and will be illustrated in detail in the following section.

3.1 Step 1: UML AD Modeling and Preprocessing

The purpose of UML AD modeling is to simulate the execution process of a software system under test and describe its dynamic behaviors [27]. And the UML AD model is generated by abstracting the software requirements document. To model the system quickly and

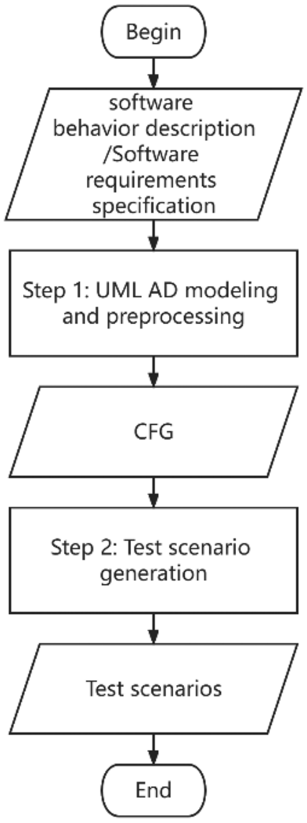

Begin
software behavior description /Software requirements specification
Step 1: UML AD modeling and preprocessing
CFG
Step 2: Test scenario generation
Test scenarios
End

Fig. 1 Overview of test scenario generation

visually, we choose the yED software [28] as the UML AD modeling tool, which can export visual ADs to XML files. And then these XML files are parsed to automatically generate the CFGs.

Figure 2 is the AD of a fake transaction, ATM transferring process, which is taken as an example to explain the test scenarios generation process of our DFS-KeyLevel approach. First, a user plugs in a bank card and enters the password continuedly until the enters are correct. Then the ATM starts the transfer operation after the user inputs the target account b and amount, which is a concurrent activity module. The concurrent module includes four activities, which are organized as two threads. One thread contains transferring to account b and balancing the account b. The other thread contains account b transferring to current account and balancing the current account. In each thread, the two activities are executed in sequence order. However, the execution order of the activities from two threads are indeterminate, because the two threads may be carried out randomly. Finally, the user clicks the button to print the receipt after the transfer operation.

After the software behaviors are modeled with ADs, a set of rules [11] is used to convert ADs into CFGs, which are directed graphs. Based on these rules, the activities and actions in an AD are mapped to the nodes of a directed graph, and the flows in the AD are mapped to the edges of the directed graph. Figure 3 is the CFG of the AD shown in Fig. 2. Each node in the CFG is represented with a number to conveniently further analysis. For branch nodes, their true value and false value are represented by 1 and 0 respectively. The generated CFG is also expressed and saved in XML format, which includes nodes, control flows, constraints and so on. The XML file is taken as the input of the next step: test scenario generation.

3.2 Step 2: Test Scenario Generation

After AD modeling and preprocessing, test scenarios are generated using our proposed DFS-KeyLevel approach, which is depicted in Fig. 4. DFS-KeyLevel approach consists of four sub-steps. The first sub-step is to simplify AD. The concurrent activity module of an CFG will be abstracted as a composite node. The second sub-step is primary test scenario generation. We proposed KeyLevel-Simple method to deal with the concurrent activity modules with simple structures and KeyLevel-Complex method to deal with the concurrent activity modules with complex structures. The third sub-step is high-layer test scenario generation. We use the improved DFS algorithm to generate scenarios from the simplify AD. The final sub-step is combination of two-layer test scenarios.

3.2.1 Sub-Step 2.1 AD Simplification

The first sub-step is to simplify AD. In order to deal with concurrent activity modules better, it is necessary to simplify the AD first. The concurrent activity module of an CFG starts with a fork node and ends with a join node. The concurrent activity module can exist as a separate system within the overall AD. Other activities in the AD do not affect the concurrent activities going on inside the concurrent activity module. The same is true in the CFG transformed by the AD. And this is the theoretical basis of our two-layer test scenario generation approach. Moreover, generating test scenarios from concurrent activity modules is complex. If the CFG is considered as only one layer to generate test scenarios, additional concurrent test scenario generation is required when the test scenario covers the concurrent activity module, which is inefficient when this operation is performed in the whole CFG. Therefore, according to the independence of the concurrent activity module, such a concurrent structure is abstracted as a composite node to be further processed. For the ATM transferring process example, Fig. 5 is the simplified CFG, in which the node X refers to the concurrent activity module (including nodes 12, 13, 14 and 15) in Fig. 3.

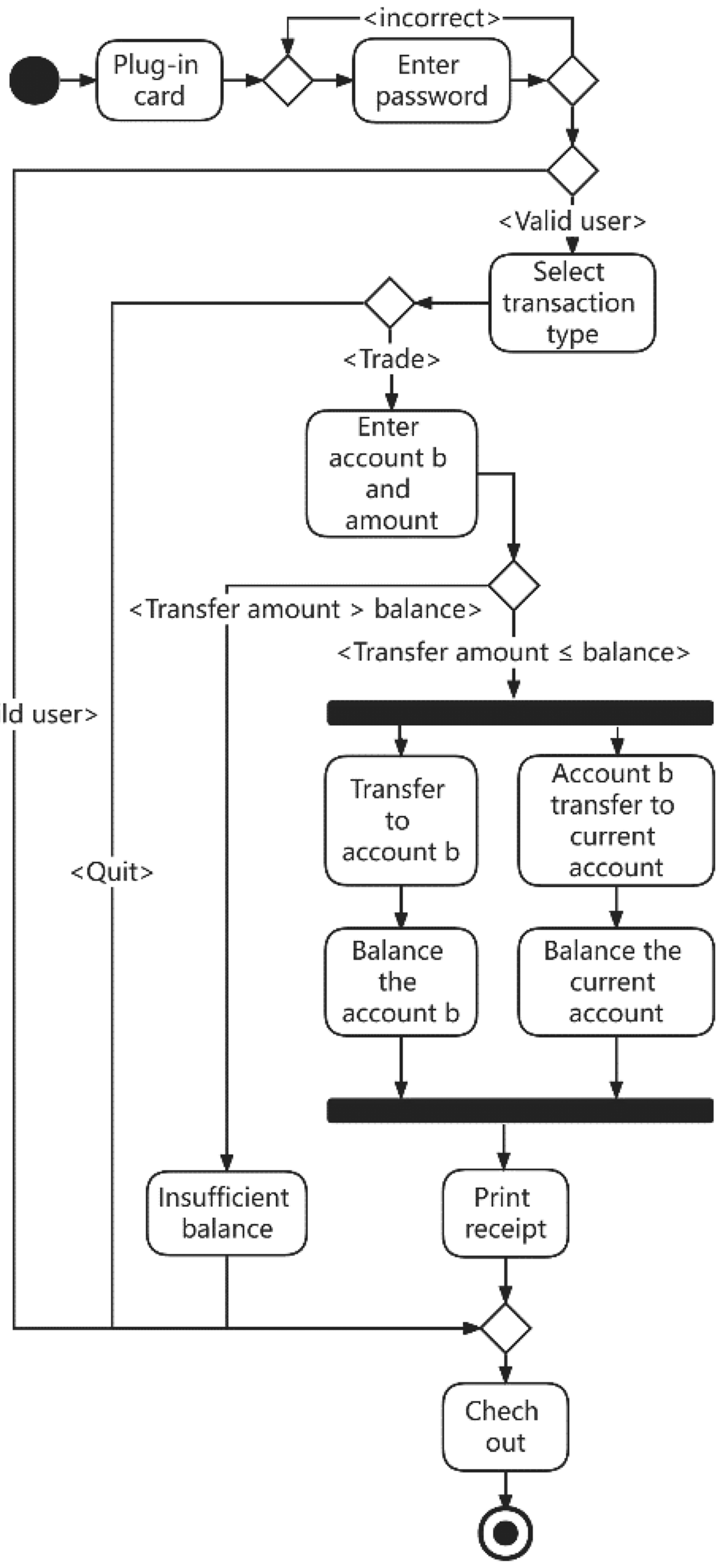


Fig. 2 AD of ATM transferring process

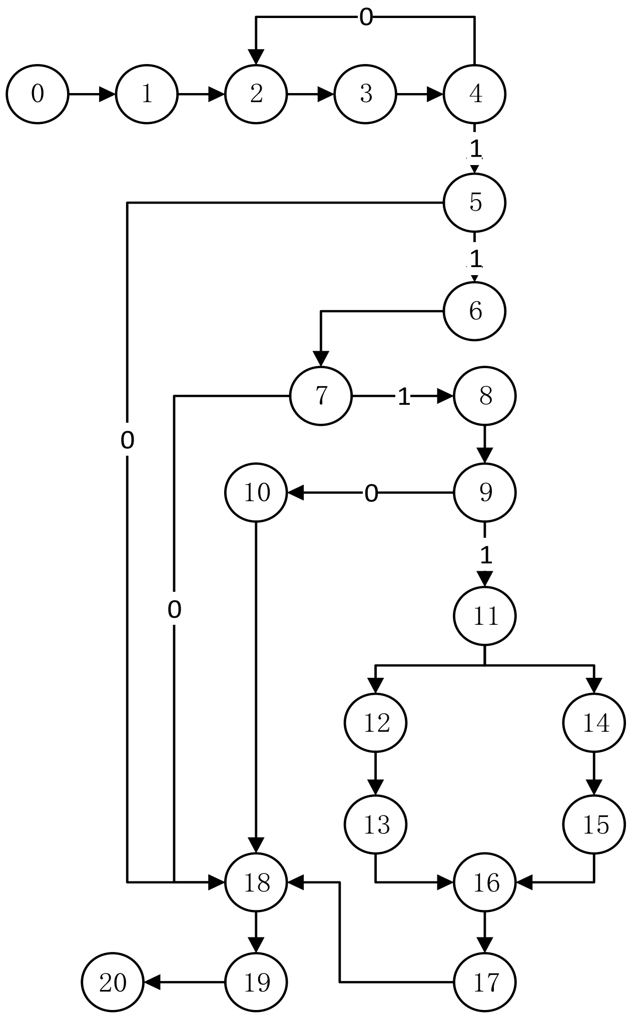


Fig. 3 CFG of ATM transferring process

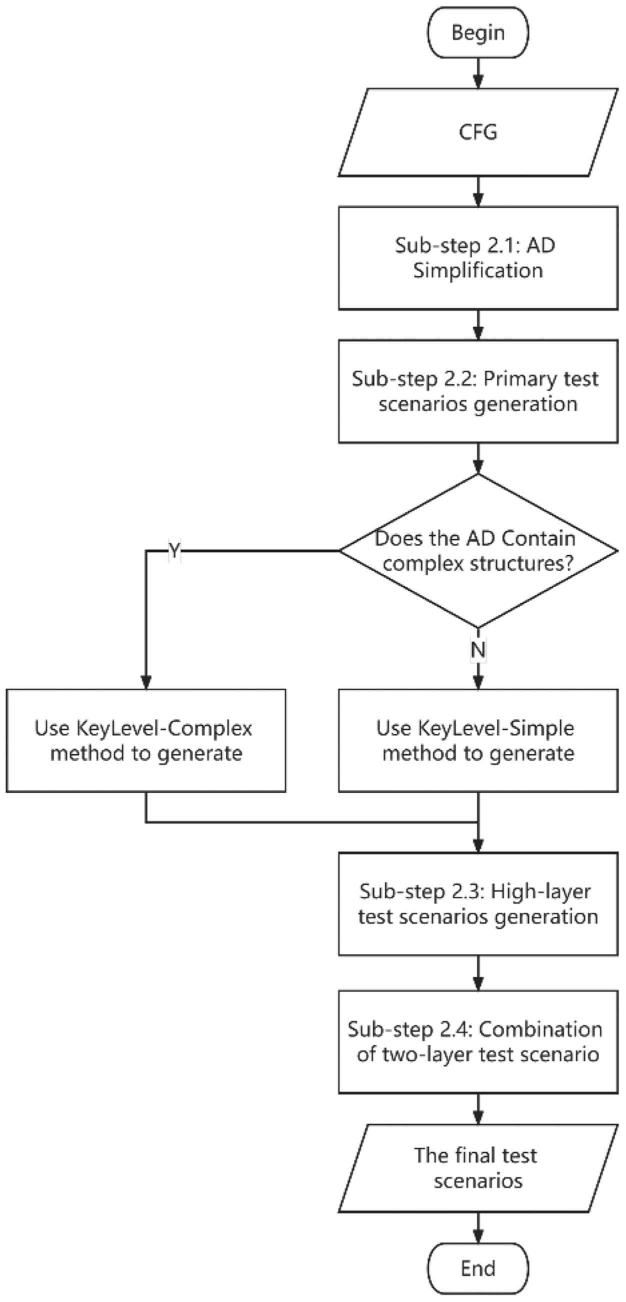

Begin
CFG
Sub-step 2.1: AD Simplification
Sub-step 2.2: Primary test scenarios generation
Does the AD Contain complex structures?
Y
N
Use KeyLevel-Complex method to generate
Use KeyLevel-Simple method to generate
Sub-step 2.3: High-layer test scenarios generation
Sub-step 2.4: Combination of two-layer test scenario
The final test scenarios
End

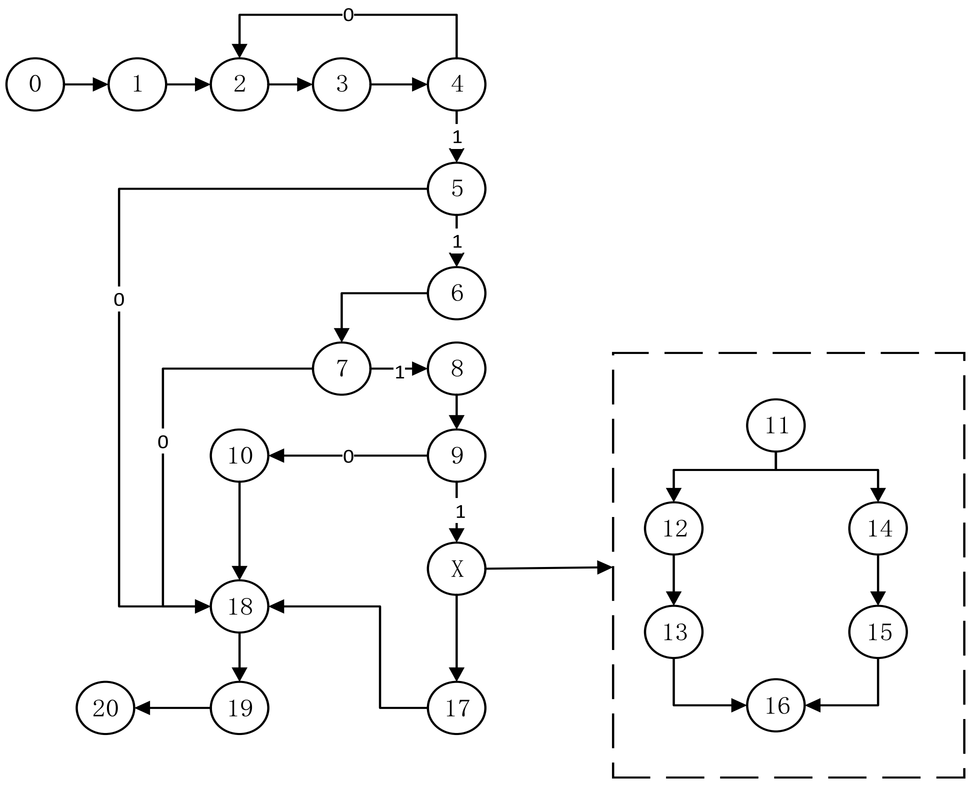
Fig. 4 Procedure of test scenario generation

3.2.2 Sub-Step 2.2 Primary Test Scenario Generation

The second sub-step is to generate test scenarios from concurrent activity modules, in which each thread may contain sequential, branch and loop structures. In the branch structures, not all nodes can participate in the final concurrent processing; And in the loop structures, some nodes will repeatedly participate in the final concurrent processing. Therefore, it is more difficult to deal with the concurrent activity modules which contain branch and loop structures than those which only contain sequential structures. Thus, we refer to the former as the complex concurrent activity module and the latter as the simple concurrent activity module. The examples of both types are shown in Fig. 6. To solve the problem of path generation explosion of concurrent activity modules, this paper proposes two KeyLevel methods to generate constrained test scenarios under the coverage condition. The KeyLevel-Simple method is proposed to generate test scenarios efficiently from simple concurrent activity module. The KeyLevel-Complex method is presented to get accurate results while generating test scenarios from complex concurrent activity module. Next, we will describe the two methods in detail.

KeyLevel-Simple Method It mainly includes the following two steps: In the first step, we propose the KeyNode algorithm to calculate the key degree value of nodes, which is to divide the directed graph, so that it has certain constraints to avoid path explosion. In the second step, according to this constraint, we propose the KeyPermute algorithm to

Fig. 5 Simplified ATM AD

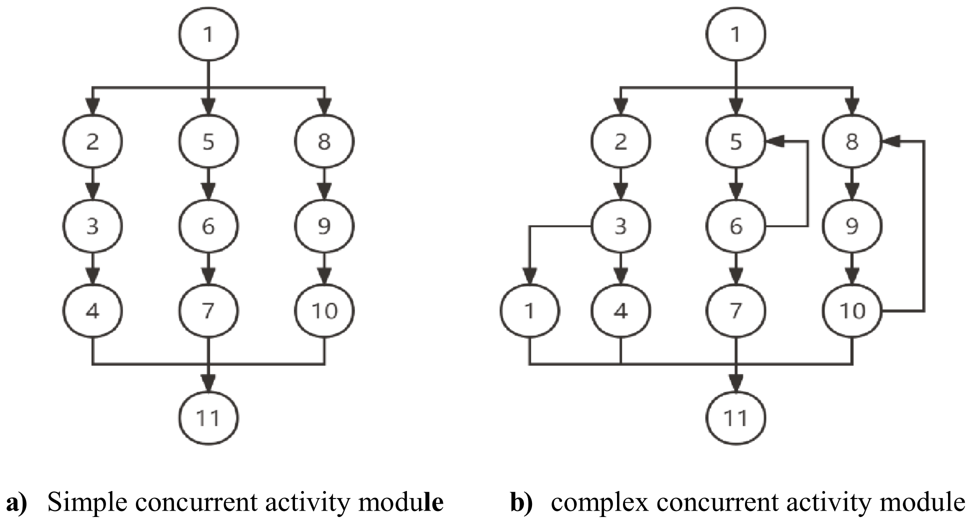


generate test scenarios. Finally, the two steps are integrated into the KeyLevel-Simple method.

We calculate the value of each node in the concurrent activity module, and all nodes are layered according to this value. If the value of the current node is larger, it means that the activity will be accessed first when the program is executed. This method can effectively and objectively grade concurrent modules, so it can avoid generating invalid test scenarios in the process of execution.

The idea of node key degree value calculation is described as follows: all nodes in the concurrent AD are calculated according to their out-degree and in-degree, and results are used as the standard of hierarchical division. Each node has its own direct neighbors and indirect neighbors. The direct neighbors are the adjacent nodes, and the indirect neighbors is the direct neighbors of the adjacent nodes. For Fig. 5, the direct neighbors of node 5 are node 6 and node 18, while the indirect neighbors are connected nodes after

Fig. 6 Different types of concurrent activity modules


1
2
3
4
5
6
7
8
9
10
11
a) Simple concurrent activity module
b) complex concurrent activity module

node 6 and node 8. For the current node, the direct neighbors are most affected by it, and the indirect neighbors will be affected by the adjacent relationship with the current node. Therefore, the result calculated according to the out-degree and in-degree indicates the key degree of the node, and its calculation formula is shown in (1):

$$f(n) = \begin{cases} \sum_{i=1}^{m} \lambda f\left(n_i\right) + P \\ P \end{cases} \tag{1}$$

$f(n)$ is the key degree value of the current node n; m—— the out-degree of the current node; $f\left(n_i\right)$ is the cumulative value of the key degree values of the immediate neighbors; λ is impact factor, that is, the importance of the impact node. P is the in-degree of node n; Since the last node is the end node and has no degree, its value is a fixed value P.

This paper designs an algorithm KeyNode to calculate the value of each node in the concurrent AD as shown in Algorithm 1. In this algorithm, the input of the algorithm is the adjacency matrix, which is the representation of a directed graph, and the output of the algorithm is a set with constraints calculated according to the above formula. First of all, it is necessary to calculate the value of each node's in-degree and out-degree by topological sorting. Then, it uses recursive method to calculate the value of each node in the directed graph by using the formula (1). The variable sum is used to record the cumulative value of the current node's key degree value. Since the value of the last node is fixed, it is first added to the temporary set weightLevel. The remaining nodes are added to the temporary set weightLevel after calculation. When all nodes are calculated, the nodes need to be sorted according to the order of key degree value, the sorted nodes are added to the result set, and finally the set Result is returned. Using this algorithm for compound node X in Fig. 5, the final partition result is shown in Fig. 7 Concurrent activity modules after division below. It can be clearly seen that the nodes of the divided concurrent activity modules follow a constraint relationship, which indicates that the activity nodes with high key degree will be accessed first.

Algorithm 1: KeyNode for Calculating Node Criticality Value

```
Input:   CFG
Output:  Node result set
1:   For j= 0 to edges.length do
2:     temp=0;
3:     For i= 0 to edges[0].length do
4:         If edges[i][j]==0 then
5:             inMap.put(j, inMap.getOrDefault(j, temp) + 1);
6:         End
7:     End
8:   End
9:   Topological();
10:  For i= level.size-2 to 0 do
11:    For j= level.get(i).size()-1 to 0 do
12:        x=current node;
13:        For y← 0 to edge[0].length do
14:            sum+=weightLevel.get(y)×0.8+P;
15:        End
16:    End
17;    weightLevel.put(x,sum)
18:  End
19:  myMap();
20:  result=map.LastLevel();
21:  Return result;
```

Next, we propose a backtracking combination algorithm (KeyPermute), which is shown in Algorithm 2. In the KeyPermute algorithm, the output of the KeyNode algorithm is used as the input of the KeyPermute algorithm, and the output of this algorithm is the test scenario set. The main idea of this algorithm is to traverse the result set, and every time a layer is traversed, the permute function will generate a full permutation of the set of nodes at current layer, in which the backtracking algorithm that is one of the more general solutions for the full permutation generation algorithms is used to calculate the full permutation. The fully permutation results of the current layer will be hierarchically added in the result set. And then the result set is expanded according to the number of nodes in the next layer of the current layer. Finally, we combine each layer of the result set to generate primary test scenarios. The whole algorithm process is the model test scenario generation process.

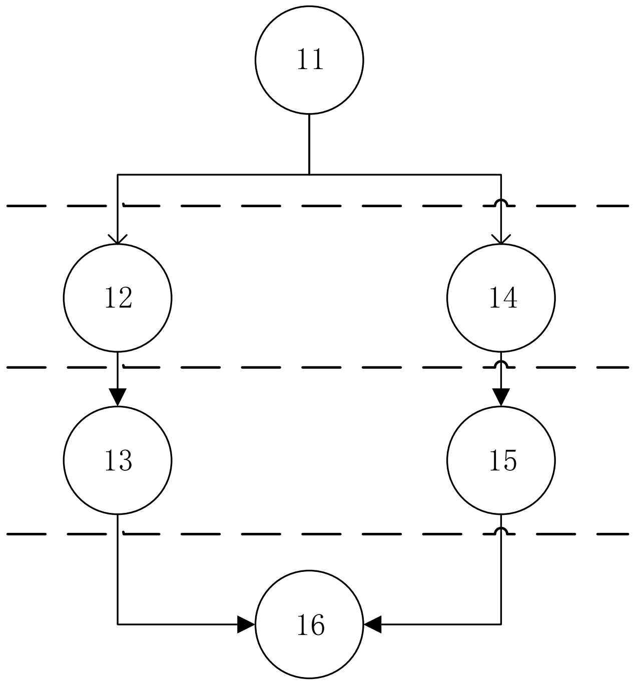


Fig. 7 Concurrent activity modules after division

As can be seen from Fig. 7, the concurrent activity module divided according to the KeyNode algorithm comes with a kind of constraint, which can just avoid generating invalid test scenarios, and it is not necessary to generate test scenarios according to the constraint conditions again as described in reference [26]. The constraint relation generated in this section is more objective, because it is calculated according to the fixed access degree of each node, so the constraint relation is fixed.

Algorithm 2: KeyPermute for Generating Concurrent Activity Module Test Scenarios

```
Input:   Node result set
Output:  Test scenarios set TS
1:   For arrayList to result do
3:     List tempNode ;
4:     For integer to arrayList do
5:         tempNode.add(integer);
6:     End
7:   End
8:   permute=permutes();          // The permutes() method uses a backtracking
     algorithm to generate full permutations
9:   res.add(permute)
10:  For i= 1 to res.size() do
11:      size=res.get(i).size();
12:      listAll=res.get(i);
13:      x=size×TS.size()
14:      While TS.size()!=x do
15:          Lists=list.get(j);
16:          copyList=new List(lists)
17:          TS.add(copyList)
18:      End
19:      j=0;
20:      While j<TS.size() do
21:        For subList← to listAll do
22:          TS.get(j).add(subList)
23:        End
24:      End
25:  End
26:  return TS
```

The test scenarios generated by this algorithm follows the above constraints. But it can be found that a certain number of test scenarios can be generated by using the method proposed in this paper. However, this method only considers the interaction of threads in concurrent activity modules, and does not consider the sequential execution of threads. Therefore, in order to be more comprehensive, this paper uses DFS algorithm to assist in generating test scenarios. The test scenarios generated by the concurrent AD in Fig. 5 are shown in Table 1. It can be seen that the concurrent activity module generates 6 test scenarios.

Table 1 Number of test scenarios for ATM concurrent active modules

Number	Test scenarios
1	11-12-14-13-15-16
2	11-12-14-15-13-16
3	11-14-12-13-15-16
4	11-14-12-15-13-16
5	11-12-13-14-15-16
6	11-14-15-12-13-16

KeyLevel-Complex Method KeyLevel-complex method also consists of two steps: dividing the node hierarchy and generating a test scenario with conditional constraints. The second step is exactly the same as KeyLevel-simple method, so we will not repeat it here, but mainly elaborate the first step of KeyLevel-complex method: dividing the node hierarchy.

In the branch structure, nodes under different branch paths cannot participate in concurrency at the same time, so some nodes must be eliminated. In the loop structure, according to the number of loop triggers, nodes on the loop path will participate in concurrency many times, leading to the recurrence of the same node, so some nodes need to recur and be divided into different levels. Therefore, it is more difficult to deal with concurrent activity modules which include branch and loop structures. So, this paper proposes the KeyLevel-Complex method which divides the node hierarchy from the perspective of path combination. In an activity path, the earlier the activity appears, the higher priority it has [19].

In this paper, the KeyPath algorithm is designed to divide the node hierarchy, as shown in Algorithm 3. In this algorithm, the input of the algorithm is the adjacency matrix, which is a representation of a directed graph, and the output is multiple sets with constraints. Firstly, according to the adjacency matrix, all simple paths of each thread are generated individually. The simple paths are to ensure that no invalid and unreasonable test scenarios are generated when generating test scenarios and to set the constraints for generating test scenarios in the complex concurrent activity module to avoid the path explosion problem. Secondly, the simple paths of each thread are combined in different sets. Each set contains one path of each thread, and all sets are different. Finally, we top align the paths in each set, which means the initial nodes in each path will be in the same position, and any node in the same position will be divided into the same hierarchy. Eventually multiple sets of nodes with constraints are generated.

Algorithm 3: KeyPath for Dividing Node Hierarchy

```
Input:   CFG
Output:  Node result set
1:  For i= 0 to thread.size do
2:    For j=0 to thread[i].node.size do
3:      If node[j] is Join then
4:        If node[j] is Decide then
5:          path=DFS(node[j]);
6:          thread[i].simplePath.add(path);
7:        Else if node.limit>0 then
8:            For k=1 to node.nextnode.size do
9:                path=DFS(node.nexnode[k]);
10:               thread[i].simplePath.add(path);
11:           End
12:         End
13:       End
14:     End
15:   End
16: End
17: For i=0 to thread.size do
18:   For j=0 to thread[i].simplePath.size do
19:     Set[j].add(thread[i].simplePath[j]);
20:   End
21: End
22: For i=0 to set.size do
23:   For j=0 to set[i].simplePath.size do
24:     For k=0 to set[i].simplePath[j].node.size do
25:       nodeLayer[k].add(set[i].simplePath[j].node[k]);
26:     End
27:   End
28: End
29: For i=0 to nodeLayer.size do
30:   layerArrange[i]=arrange(nodeLayer[i]);
31: End
32: Return layerArrange;
```

We take Fig. 6b as an example. This concurrent activity module has three threads. We name the left thread as thread 1, the middle thread as thread 2 and the right thread as thread 3. To describe conveniently, the number of loops is set to 0 or 1. The full simple path generated from thread 1 are: {{2, 5, 8}, {2, 5, 9}}; the full simple path generated from thread 2 are: {{3, 6, 10}, {3, 6, 3, 6, 10}}; the full simple path generated from thread 3 are: {{4, 7, 11}, {4, 7, 11, 4, 7, 11}}. Combining the paths of all threads gives 8 sets, and we choose the set: {{2, 5, 8}, {3, 6, 3, 6, 10}, {4, 7, 11, 4, 7, 11}} to illustrate. The result of top aligning the three paths is shown in Fig. 8, in which the dotted line shows the hierarchy. The final node hierarchy result is: {{2, 3, 4}, {5, 6, 7}, {8, 3, 11}, {6, 4}, {10, 7}, {11}}. The result contains 6 layers, and then we perform full arrangement of the nodes within each layer. Next, keeping the hierarchical order unchanged, we combine the results of each layer of nodes arrangement, which generates 864 paths. In this concurrent activity module, the theoretical number of paths generated without constraints is 395,136, which are too many to test. But the number of paths generated by using KeyLevel-Complex method is only 3024. The number of generated paths is reduced by about 99.24%, which makes test possibly.

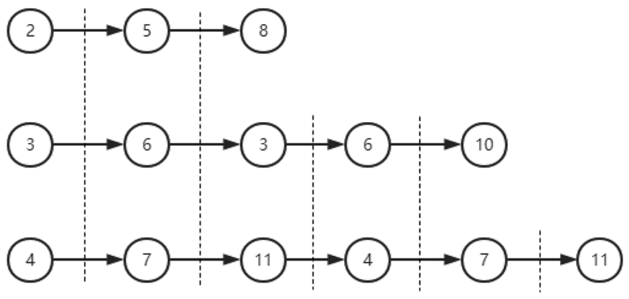


Fig. 8 Top alignment path

Time Complexity Analysis Test scenario generation of concurrent activity module is essentially a permutation problem, so it is also an NP-complete problem. In this paper, the following optimizations are made based on the characteristics of test scenario generation: First, the nodes on each thread of the concurrent activity module are in relative order, and test scenarios containing paths that do not conform to the relative order are illegitimate test scenarios. In this paper, these illegitimate test scenarios are eliminated, which simplifies the complexity of the problem and also ensures the legitimacy of the generated test scenarios; secondly, each node is assigned a key value, and nodes with the same key value are assigned to the same layer. This further reduces the complexity of the problem by not traversing low key values when the layer with high key values has not been traversed. In this paper, the above two steps reduce the time complexity of test scenario generation for concurrent activity module to an acceptable range. Next, we will analyze the time complexity of Algorithm 2 and Algorithm 3 in detail, respectively.

Algorithm 2 involves permutations, so the time complexity is significant. But our algorithm still has some advantages. Assume that the concurrent activity module contains **n** nodes and **m** threads. In the worst case, each thread is of the same length, which means that **n** nodes can be divided into $l = n/m$ layers. Based on the backtracking algorithm used in this paper, the time complexity is $O(n * n!)$ if the full permutation of n nodes is performed directly. algorithm 2 permutes each layer first, and then combines the results of each layer. In the worst case there is only one layer, $m = n$. At this point the time complexity is $O(n * n!)$ as in a direct full permutation of **n** nodes. In the general case, $m < n$, then based on the previous worst-case assumptions, each layer has to perform a full permutation of m nodes. The time complexity is $O(m * m!)$, generating **m!** results. The results of each layer are then combined, and the time complexity is, so the time complexity of algorithm 2 is $O\left(m * m! + m!^{\frac{n}{m}}\right)$ .The time complexity of algorithm 2 is better than the general permutation algorithm, and the permutation algorithm in this paper is implemented recursively, which can further reduce the time complexity.

Algorithm 3 is used to deal with concurrent activity modules containing loops and branching structures, and the complexity will be higher compared to Algorithm 2. Assuming that the concurrent activity module contains **n** nodes, the general full permutation algorithm cannot solve the permutation problem that contains both loop and branch structures. Because of these two structures, the number of nodes involved in a complete concurrent activity is not necessarily **n**. Therefore, in this paper, we first determine the total number of nodes involved in concurrent activities, which is the first step in algorithm 3: generating the full simple path for each thread. This step uses the DFS algorithm with a time complexity similar to that of the DFS algorithm. Assuming that the number of nodes in this simple path is **n** and the number of edges is **e**, the time complexity is $O(n + e)$ under the condition that the neighboring node information is stored in the neighboring table used in this paper. However, the time complexity may fluctuate because of the presence of loops and branching structures. The second step is to combine the simple paths generated by each thread to form a completed concurrent activity. We suppose there are **m** threads and the number of simple paths generated by each thread is $n_i(i = 1, 2, 3 \ \ldots, \ m)$, then the time complexity of the combination is $O\left(\prod_{i=1}^{m} n_i\right)$. In a complete concurrent activity, the time complexity of the test scenario generation is the same as in algorithm 2. Assuming **n** nodes, **m** threads, **l** layers, and the same number of nodes per layer in the worst case, the time complexity is $O\left(m * m! + m!^{\frac{n}{m}}\right)$. So, the time complexity of Algorithm 3 is $O\left(\prod_{i=1}^{m} n_i * \left(m * m! + m!^{\frac{n}{m}}\right)\right)$.

3.2.3 Sub-Step 2.3: High-Layer Test Scenario Generation

The third sub-step is high-layer test scenario generation. This paper improves the DFS algorithm to adapt to the test scenarios generation. In the improved DFS algorithm, the processing of branch nodes and circular nodes is different from traditional DFS. In the process of traversing simplified AD with improved DFS algorithm, if the current node is a branch node, one of the adjacent nodes of the current branch node is randomly selected to traverse. For the loop activity modules, the algorithm can loop N times, but for the sake of testability, we set N to 1. So, if the current node is a circular node, choose to traverse once or not.

Table 2 Test scenarios of ATM AD

No.	Test scenario
1	0-1-2-3-4-5-18-19-20
2	0-1-2-3-4-5-6-7-18-19-20
3	0-1-2-3-4-5-6-7-8-9-10-18-19-20
4	0-1-2-3-4-5-6-7-8-9-11-12-14-13-15-16-17-18-19-20
5	0-1-2-3-4-5-6-7-8-9-11-12-14-15-13-16-17-18-19-20
6	0-1-2-3-4-5-6-7-8-9-11-14-12-13-15-16-17-18-19-20
7	0-1-2-3-4-5-6-7-8-9-11-14-12-15-13-16-17-18-19-20
8	0-1-2-3-4-5-6-7-8-9-11-12-13-14-15-16-17-18-19-20
9	0-1-2-3-4-5-6-7-8-9-11-14-15-12-13-16-17-18-19-20
10	0-1-2-3-4-2-3-4-5-18-19-20
11	0-1-2-3-4-2-3-4-5-6-7-18-19-20
12	0-1-2-3-4-2-3-4-5-6-7-8-9-10-18-19-20
13	0-1-2-3-4-2-3-4-5-6-7-8-9-11-12-14-13-15-16-17-18-19-20
14	0-1-2-3-4-2-3-4-5-6-7-8-9-11-12-14-15-13-16-17-18-19-20
15	0-1-2-3-4-2-3-4-5-6-7-8-9-11-14-12-13-15-16-17-18-19-20
16	0-1-2-3-4-2-3-4-5-6-7-8-9-11-14-12-15-13-16-17-18-19-20
17	0-1-2-3-4-2-3-4-5-6-7-8-9-11-12-13-14-15-16-17-18-19-20
18	0-1-2-3-4-2-3-4-5-6-7-8-9-11-14-15-12-13-16-17-18-19-20

3.2.4 Sub-Step 2.4: Combination of Two-Layer Test Scenarios

The complete test scenarios of the activity diagram are generated by the combination of the above two scenarios. In the process of traversing the simplified AD with the improved DFS algorithm, the nodes with X marks in the test scenarios generated in the simplified AD need to be decompressed. When the two layers test scenarios are finally merged, the composite nodes in the high-layer test scenarios are expanded into the previously generated primary test scenarios, at which time the high-layer test scenario and the primary test scenario are combined into the final test scenario. Assuming that there are currently M test scenarios containing X marks, and N test scenarios generated by concurrent activity modules, the total number of test scenarios is M*N. For ATM AD, 18 test scenarios are generated by our approach, and the details of test scenarios are shown in Table 2.

4 Experimental Results and Discussion

Several experiments were taken to evaluate the DFS-KeyLevel approach, which was implemented with C++in Visual Studio 2022 under Windows10 system, Intel 2.3 GHz octa-core Core i7 and 32GB memory. First, we conduct the comparison experiments between our DFS-KeyLevel and state-of-the-art approaches. Then our

Table 3 Detailed AD information of five software cases

Model Name	Number	Ca	Nca	La	Nla
GraphicsUtility	16	Y	2	N	0
OrderProcess	18	Y	1	Y	1
AirPortCheck	27	Y	3	N	0
MakeCall	37	Y	1	Y	1
ATM	21	Y	1	Y	1

approach was applied into a real embedded system, which contains very complex concurrent modules. In the experiments, the value λ of in the formula (1) is 0.8.

In the comparison experiments, five different types of software cases were adopted, including ATM transferring process shown in Fig. 1, GraphicsUtility [24] for Calculate coordinates, OrderProcess [24] for Order process, AirPortCheck [24] for airplane security check and MakeCall[1] for phone call process. The detailed information about these five cases is shown in Table 3, where the "Number" refers to the quantity of nodes in each case; "Ca" refers to whether there are concurrent activity modules; "Nca" is the number of concurrent activity modules; "La" refers to whether there are loop activity modules; and "Nla" is the quantity of loop activity modules.

4.1 Comparative Experiment of Test Scenario Generation Number

To illustrate the effect of our DFS-KeyLevel (DKL) approach, DFS-LevelPermutes (DLP) [24], DFS-BFS (DB) [18] and IDFS (ID) [5] approaches are taken as baselines.

The generation results of the number of test scenarios of concurrent modules are shown in Fig. 9 below. As can be seen from the Fig. 9, the KeyLevel-Simple (KLS) and LevelPermutes (LP) methods generate more test scenarios as the complexity of the concurrent activity module increases. And no matter which model the method in this paper is applied to, the number of effective test scenarios generated is obviously more than that of the other three methods. The more the number of test scenarios, the effectiveness of the KeyLevel-Simple method proposed in this paper can be demonstrated.

The generated results of the number of test scenarios for the AD are shown in Fig. 10 below. For the AD, the number of test scenarios of concurrent activity modules contained in it greatly affects the number of test scenarios generated by the AD. The more test scenarios generated by concurrent activity modules, the more the total number of test scenarios. Of course, if there are loop activity modules in the AD, the number generated will increase with the number of loop activity modules. As shown in Fig. 10, for all ADs, the number of test scenarios generated by DFS-KeyLevel in this paper is better than the other three approaches, while DFS-LevelPermutes approach does not deal with loop activity modules, so it does not perform well in ATM model and MakeCall model. It can be seen from the information of MakeCall model in Table 3 that this model has both loop activity modules and concurrent activity modules, so the number of test scenarios generated by this approach is the largest.

It can be clearly seen from Figs. 9 and 10 that the approach in this paper is better than the test scenarios generated by DFS-LevelPermutes, DFS-BFS and IDFS approach, and the more complex the model, the more obvious the advantages of the approach in this paper, such as GraphicsUtility model and Muchurian model. From the angle of increasing the number of concurrent activity test scenarios, the number of scenarios generated by the KeyLevel-Simple method in this paper is 0.85 times higher than that of LevelPermutes method and 3.03 times higher than that of DFS-BFS and IDFS approaches. From the perspective of increasing the number of test scenes of AD, the number of scenes generated by DFS-KeyLevel approach in this paper is increased by 1.13 times compared with DFS-LevelPermutes approach, and by 2.37 times compared with DFS-BFS and IDFS approaches.

4.2 Comparative Experiment of Test Coverage

Coverage criterion is one of the criteria to measure the quality of test scenarios. The higher the coverage of test scenarios in each test scenario, the higher the coverage of test scenarios. Therefore, the test coverage is equal to the test scenario coverage at this time. To verify the effectiveness of the approach, activity coverage [23], simple path coverage [24] and concurrency coverage [3] are selected in the experiment for the concurrent activity module of AD. Simple path coverage refers to the coverage of sequential paths and sequential non-interleaving concurrent paths. Active coverage refers to the coverage of representative paths generated by the BFS algorithm. Concurrency coverage refers to the coverage of interleaved concurrent paths. These three test coverage criteria analyze the experimental results; For the AD, node coverage [29], edge coverage and TALPC are selected in the experiment to verify the feasibility of the proposed approach.

TAPC (Total Activity Path Coverage Criterion) [23] is used to evaluate the whole AD. It is a collection of basic paths and interlaced paths of concurrent activities. However, the basic path requires that each activity can only be executed once, which cannot evaluate the concurrent activity

[1] Data Source: https://gitee.com/vanishkk/two-layer-test-scenario-generation/tree/master.

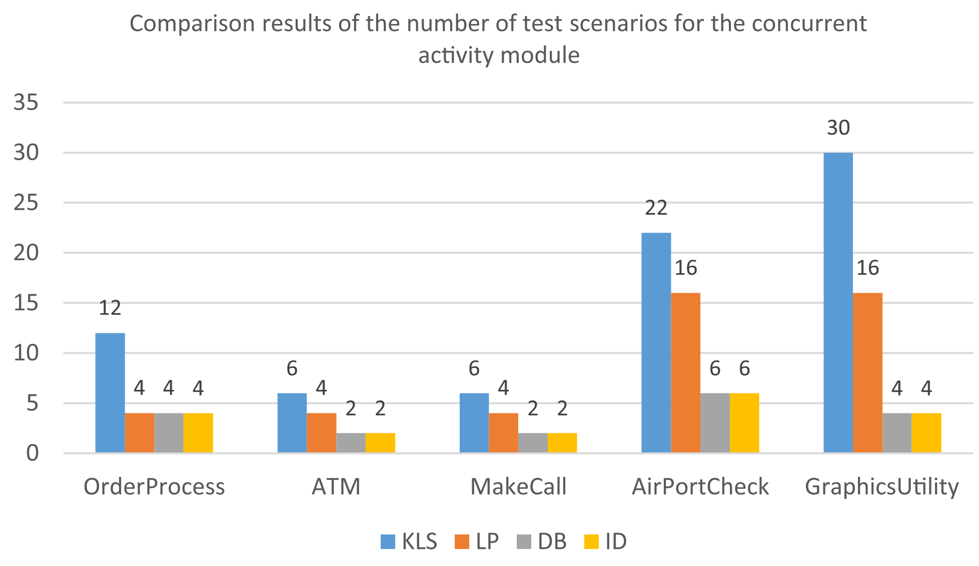


Fig. 9 Comparison results of the number of test scenarios for the concurrent activity modules

modules of nested loops. Therefore, this paper modifies this, and puts forward Total Activity Logical Path Coverage (TALPC) to evaluate the AD, which is a collection of logical paths and concurrent paths of concurrent activities and can guarantee the effective coverage of loop activities. However, the loop needs to be limited.

As can be seen from Table 4, these four methods are applied to all models, and the activity coverage and simple path coverage reach 100%, which shows that the method in this paper meets the most basic coverage criteria of concurrent activity modules. On these five models, the average concurrent coverage of this algorithm is 83.76%, that of LevelPermutes method is 48.68%, and that of DFS-BFS and IDFS approaches is 25.08%. Because the GraphicsUtility model is a nested concurrent AD, the generated test scenario is only the test scenario under concurrent activities, and there are many nodes of concurrent activities and modules, so it is impossible to achieve 100% by using this method. If you want to achieve 100% coverage of concurrent staggered execution, you need to diversify the calculation algorithm of the KeyLevel-Simple method and divide it several times.

Table 5 shows the experimental results of coverage of test scenarios generated by AD. Node coverage and edge coverage are the most basic conditions to evaluate the quality of test scenarios. It can be seen from the table that the node coverage and edge coverage of DFS-KeyLevel approach, DFS-BFS and IDFS approach in this paper all reach 100%; While for TALPC, the DFS-KeyLevel approach in this paper has higher coverage than DFS-LevelPermutes, DFS-BFS a and IDFS approaches. On these five models, the average TALPC of this approach is 84%, that of DFS-LevelPermutes approach is 39.98%, and that of DFS-BFS and IDFS approaches is 35.56%. As the GraphicsUtility model is a concurrent AD, the coverage of TALPC is the same as that of concurrent coverage.

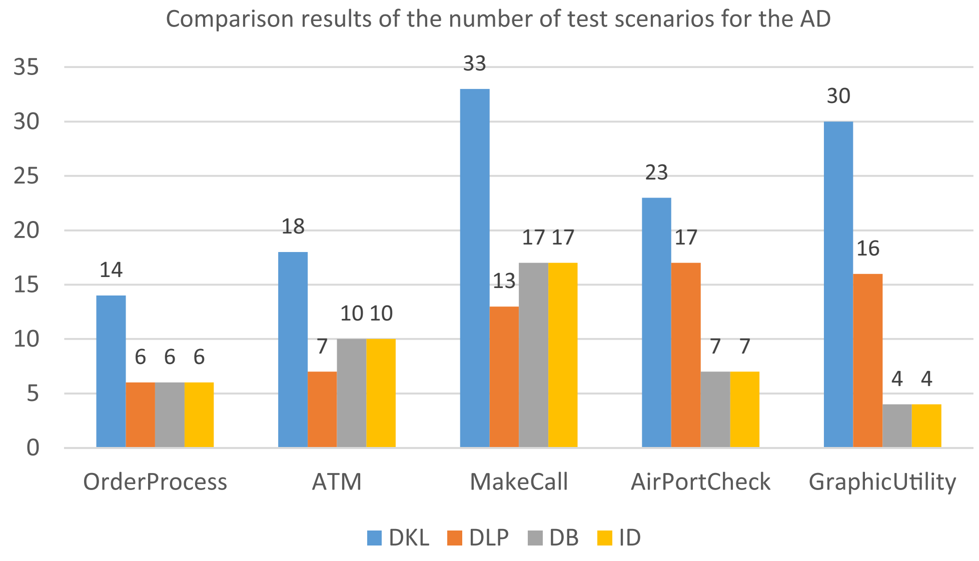


Fig. 10 Comparison results of the number of test scenarios for the AD

Table 4 Test coverage criteria for concurrent active modules

Model name	Test coverage criteria for concurrent active modules											
	Activity coverage(%)				Simple path coverage(%)				Concurrent coverage(%)			
	KLS	LP	DB	ID	KLS	LP	DB	ID	KLS	LP	DB	ID
GraphicsUtility	100	100	100	100	100	100	100	100	50	26.7	6.7	6.7
OrderProcess	100	85.7	100	100	100	50	100	100	100	33.3	33.3	33.3
ATM	100	100	100	100	100	100	100	100	100	66.7	33.3	33.3
MakeCall	100	100	100	100	100	100	100	100	100	66.7	33.3	33.3
AirPortCheck	100	100	100	100	100	100	100	100	68.8	50	18.8	18.8

4.3 Application

Application scenarios containing complex concurrent activity modules are often encountered. But all the approaches shown in the previous section cannot deal with complex concurrent activity modules well. In this section, we will test the practicality of the KeyLevel-Complex method through a real application scenario.

Smart torque wrench system can help customers to achieve smart tightening. It can measure the torque value and angle value of tightening, and then realize the quantitative control of torque. The tightening program is automatically sent down and prevents repeated and leakage of tightening to realize intelligent tightening. It also sets the corresponding tightening program, specifies the target torque value and error range, stores the tightening data, and uploads the tightening structured data automatically to avoid data loss. And the measurement error of the wrench can be controlled within $\pm 5\%$, which has practical use value. The smart torque wrench communicates with the upper computer through a embedded hardware module that can transfer data between them via wireless Bluetooth.

The modeling of the smart wrench hardware module activity is shown in Fig. 11, which contains two parts: the handheld terminal and the smart wrench. The handheld terminal includes operator login and posting tasks. And the smart wrench includes the execution of tasks, data storage and real-time monitoring and uploading sweep information. All of tasks here are fully concurrent.

A complete activity flow is as follows. To connect the hardware module with the mobile phone, we need to turn on the Bluetooth module of the wrench and the app of the mobile phone first. Then we click the adding device button in the app to search and connect the device through wireless Bluetooth. After connection, the mobile app can distribute data which can be obtained through the scanner or selected from the local list. Then the hardware module transmits the data received from the upper terminal to the wrench to complete the corresponding operation and store the operation records. After completing the task, the wrench will real-time upload operation records to hardware module. The mobile phone finally receives operation records from the hardware module. There are two main tasks of the hardware module. One is to receive and analyze the messages from the upper terminal and then carry out corresponding operations according to the message type. The success and failure of the operation will generate corresponding records. The second is to read the wrench operation records and upload them to the upper terminal in real time. In addition, it can also poll whether there is data generated at the code scanner, and analyze the content obtained by scanner and process it according to its type.

The conversion rules are then applied to convert the AD into a directed graph, as shown in Fig. 12. Two concurrent activity modules are processed using KeyLevel-Complex, and the results are shown in Table 6. Both activity coverage and basic path coverage reach 100%. The concurrency coverage is not high. The simpler concurrent activity module 1X having 40% concurrency coverage, while the more complex

Table 5 Test coverage criteria for AD

Model name	Test coverage criteria for AD											
	Node coverage(%)				Edge coverage(%)				TALPC(%)			
	DKL	DLP	DB	ID	DKL	DLP	DB	ID	DKL	DLP	DB	ID
GraphicsUtility	100	100	100	100	100	100	100	100	50	26.7	6.7	6.7
OrderProcess	100	94.4	100	100	100	90.5	100	100	100	42.9	42.9	42.9
ATM	100	100	100	100	100	95.7	100	100	100	38.9	55.6	55.6
MakeCall	100	100	100	100	100	97.4	100	100	100	39.4	51.6	51.6
AirPortCheck	100	100	100	100	100	100	100	100	70	52	21	21

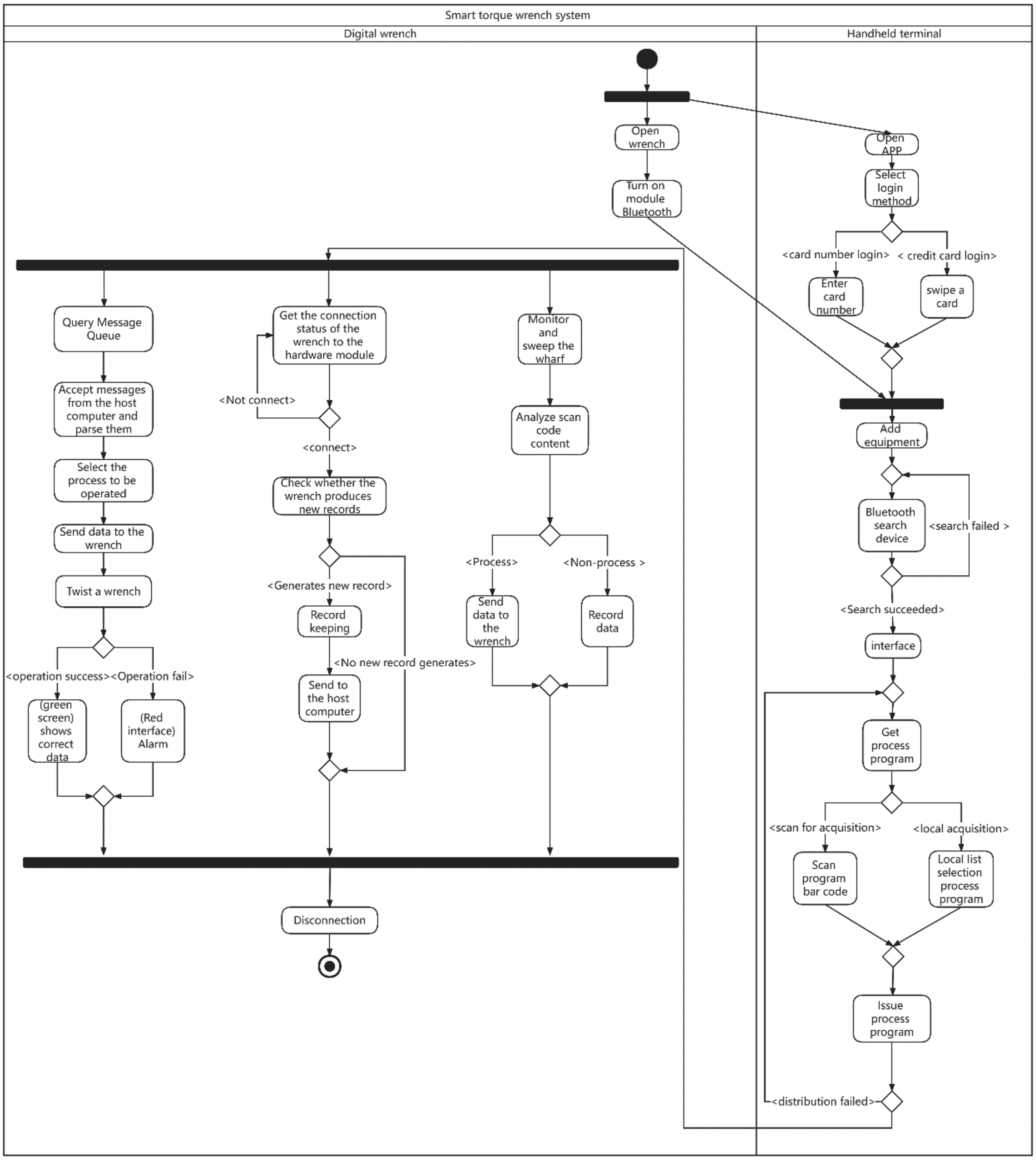


Fig. 11 Smart torque wrench system AD

15X has only about 0.013%. Because the maximum number of test scenarios generated increases rapidly when there are complex structures in the concurrent activity module. The maximum number of test scenarios generated from 15X is 1,630,423,080, while only 217,824 test scenarios are generated by KeyLevel-Complex. This greatly reduces the number of test scenarios generated but ensures enough test scenarios generated to be tested. A comparison between the two concurrent activity modules can also reveal that as the complexity of concurrent activity modules increases, our KeyLevel-Complex method can effectively constrain the growth of the number of test scenarios generation to avoid path explosion.

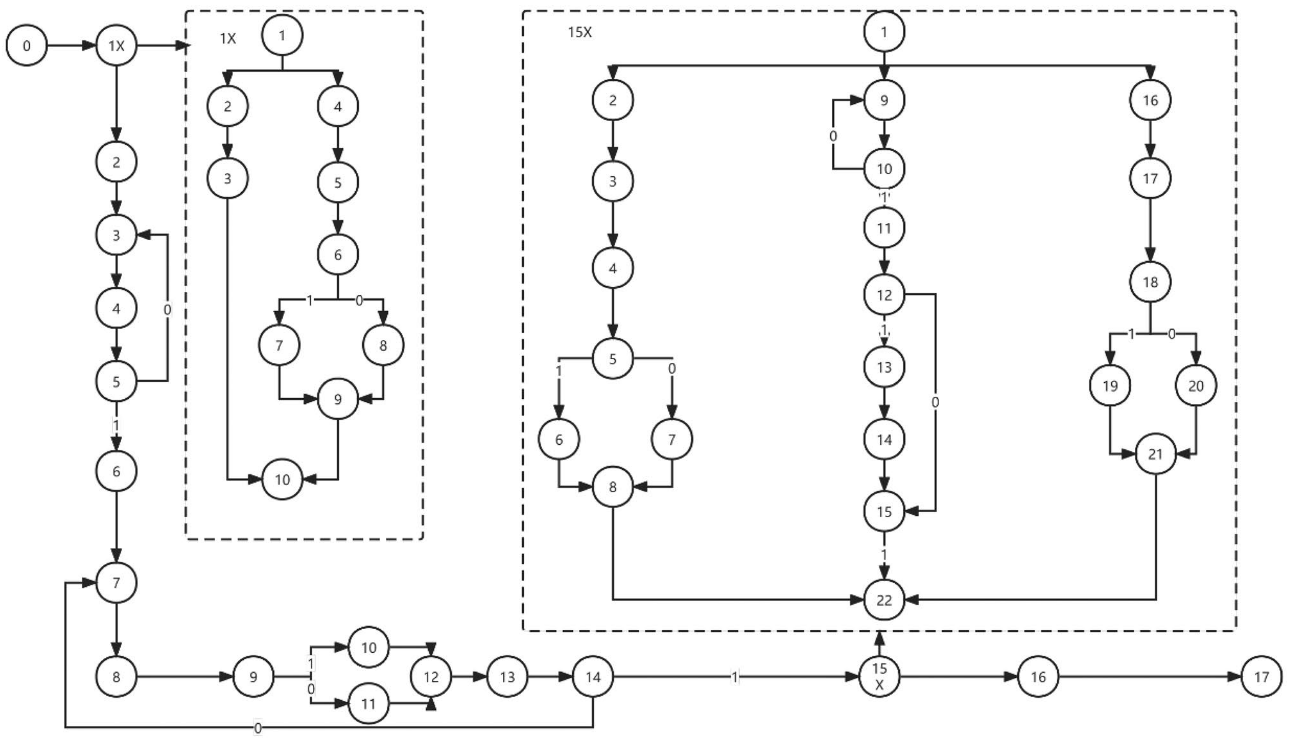


Fig. 12 Simplified directed graph

Table 6 Test result for concurrent active modules

Composite node	Test result for concurrent active modules				
	Activity coverage(%)	Simple path coverage(%)	Number of generated paths	Theoretical Number of paths	Concurrent coverage(%)
1X	100	100	8	20	40
15X	100	100	217,824	1,630,423,080	0.013

5 Conclusion

To generate the test scenarios efficiently, this paper proposes a two-layer test scenarios generation approach for AD. First, the system is modeled, preprocessed and converted into control flow graph (CFG). Next, to generate test scenarios from concurrent activities, this paper proposes Key-Level method in the primary test scenarios generation, and according to different application scenarios, it is detailed as follows: KeyLevel-Simple and KeyLevel-Complex method. Then improve DFS algorithm to deal with simplified AD in high-layer test scenarios generation. The experiment results on five different types of software cases showed that our DFS-KeyLevel is superior to the previous approaches. The number of test scenarios generated by DFS-KeyLevel approach in this paper is 1.13 times higher than that of DFS-LevelPermutes algorithm and 2.37 times higher than that of DFS-BFS algorithm and IDFS algorithm. From the coverage point of view, the average coverage rate of staggered activities is 83.76%, and the average coverage rate of TALPC of this algorithm 84%, which is significantly higher than DFS-LevelPermutes algorithm, DFS-BFS algorithm and IDFS algorithm. The experimental results of the embedded intelligent wrench tightening system showed that our KeyLevel-Complex method can deal with complex actual situations, and can effectively constrain the generation test scenarios in complex concurrent activity module, but can ensure generate enough test scenarios to be tested.

Although our approach can effectively generate the test scenarios based on UML AD, we did not verify the effectiveness of our approach on other models, such as state diagrams, which will be addressed in future. Manually abstracting activity diagrams from source code is a complex and time-consuming task when the source code size is large, and how to efficiently generate activity diagrams from source code will be a research focus for our future work. In addition, introducing relevant content in the OS may help to improve generation efficiency and handle more complex concurrent activity module. This is the part we will work on in the future.

Funding The work described in this paper is partially supported by the Chinese National Natural Science Foundation (grant number 11975182).

Data Availability Some or all data, models, code generated or used during the study are available from the corresponding author by request.

Declarations

Conflict of Interest/Competing Interest The authors have no conflicts of interest to declare relevant to this article's content.

References

1. Ahmad T, Iqbal J, Ashraf A (2019) Model-based testing using UML activity diagrams: a systematic mapping study. Comput Sci Rev 33:98–112
2. Anbunathan R, Basu A (2019) Combining genetic algorithm and pairwise testing for optimised test generation from UML ADs. Softw IET 13(5):423–433
3. Arora V, Singh M, Bhatia R (2020) Orientation-based ant colony algorithm for synthesizing the test scenarios in UML activity diagram. Inf Softw Technol 123:106292
4. Clarisó R, González CA, Cabot J (2019) Smart bound selection for the Verification of UML/OCL class diagrams. IEEE Trans Softw Eng 45(4):412–426
5. Fan LL, Wang Y, Liu T (2021) Automatic test path generation and prioritization using UML activity diagram. In: Proc. of 8th International Conference on Dependable Systems and Their Applications (DSA). pp 484–490
6. Hamza ZA, Hammad M (2019) Generating test sequences from UML use-case diagrams. In: Proc. of International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), pp 1–6
7. Jahan H, Feng Z, Mahmud S (2020) Risk-based test case prioritization by correlating system methods and their associated risks. Arab J Sci Eng 45:6125–6138
8. Jain P, Soni D (2020) A survey on generation of test cases using UML diagrams. In: Proc. of International Conference on Emerging Trends in Information Technology and Engineering (IC-ETITE). IEEE, pp 1–6
9. Jena AK, Swain SK, Mohapatra DP (2014) A novel approach for test case generation from UML activity diagram. In: Proc. of International Conference on Issues & Challenges in Intelligent Computing Techniques. IEEE, pp 621–629
10. Kamonsantiroj S, Pipanmaekaporn L, Lorpunmanee S (2019) A memorization approach for test case generation in concurrent UML activity diagram. In: Proc. of the 2nd International Conference on Geoinformatics and Data Analysis, pp 20–25
11. Kundu D, Samanta D (2009) A novel approach to generate test cases from UML activity diagrams. J Object Technol 8(3):65–83
12. Lafi M, Alrawashed T, Hammad AM (2021) Automated test cases generation from requirements specification. In: Proc. of International Conference on Information Technology (ICIT). IEEE, pp 852–857
13. Li H, Lam CP (2005) Using anti-ant-like agents to generate test threads from the UML diagrams. In: Proc. of International Conference on Testing of Communicating Systems. Springer-Verlag, Berlin, pp 69–80
14. Lima L, Tavares A (2019) Verifying deadlock and nondeterminism in activity diagrams. In: Proc. of ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp 764–768
15. Mahali P, Arabinda S, Acharya AA (2016) Test case generation for concurrent systems using UML activity diagram. In: Proc. of IEEE Region 10 Conference (TENCON), pp 428–435
16. Mahanto P, Barisal SK, Mohapatra DP (2018) Achieving MC/DC using UML communication diagram. In: Proc. of International Conference on Information Technology (ICIT). IEEE, pp 73–78
17. OMG. Foundational Unified Modeling Language v1.5. http://www.omg.org/spec/FUML/1.5/. Retrieved June 2021
18. Panthi V, Tripathi A, Mohapatra DP (2022) Software validation based on prioritization using concurrent activity diagram. Int J Syst Assur Eng Manage 13:1801–1816
19. Qian Z, Zhu J, Zhu Y (2022) Multi-path coverage strategy combining key point probability and path similarity. J Softw 33:434–454
20. Salman, Yasir D (2017) Coverage criteria for test case generation using UML state chart diagram. AIP Conf Proc 1891(1):1–6
21. Shi Z, Zeng XQ, Zhang TT, Han L (2021) UML diagram-driven test scenarios generation based on the temporal graph grammar. KSII Trans Internet Inf Syst 15(7):2476–2495
22. Shirole M, Kumar R (2012) Testing for concurrency in UML diagrams. ACM SIGSOFT Softw Eng Notes 37(5):1–8
23. Shirole M, Kumar R (2021) Concurrency coverage criteria for activity diagrams. IET Softw 15(4):43–54
24. Shirole M, Kumar R (2021) Constrained permutation-based test scenario generation from concurrent activity diagrams. Innov Syst Softw Eng 17:345–353
25. Sypsas A, Kalles D (2020) Using UML AD for adapting experiments under a virtual laboratory environment. In: Proc of 24th Pan-Hellenic Conference on Informatics. Association for Computing Machinery, New York, pp 27–30
26. Thanakorncharuwit W, Kamonsantiroj S, Pipanmaekaporn L (2016) Generating test cases from UML activity diagram based on business flow constraints. In: Proc. of the Fifth International Conference on Network, Communication and Computing. pp 155–160
27. Tiwari RG, Srivastava AP, Bhardwaj G (2021) Exploiting UML diagrams for test case generation: a review. In: Proc. of the 2nd International Conference on Intelligent Engineering and Management (ICIEM), pp 457–460
28. Yed. https://www.yworks.com/products/yed. Accessed on 2022/9/2
29. Yimman S, Kamonsantiroj S, Pipanmaekaporn L (2017) Concurrent test case generation from UML activity diagram based on dynamic programming. In: Proc. of the 6th international conference on software and computer applications. pp 33–38

Xiaozhi Du received the M.S. degree in Control Theory and Control Engineering in 2004 and the Ph.D. in Computer Science and Technology in 2010, both from Xi'an Jiaotong University, China. He is Assistant Professor in the Software Engineering Department of the same university. His research interests mainly include fault tolerance, software reliability, and SoC testing.

Jinjin Zhang received the M.S. degree in Software Engineering in 2022 from Xi'an Jiaotong University, China. Her research interests include fault recovery and software testing.

Kai Chen did postgraduate study of Software Engineering at Xi'an Jiaotong University in 2021. His research interests include software testing and embedded testing.

Yanrong Zhou did postgraduate study of Software Engineering at Xi'an Jiaotong University in 2021. Her research interests include software testing and embedded testing.