

UML diagram-driven test scenarios generation based on the temporal graph grammar

Zhan Shi^{1*}, Xiaoqin Zeng², Tingting Zhang^{3,4}, Lei Han¹ and Ying Qian¹

¹ Institute of Computer Engineering, NanJing Institute of Technology, Nanjing, China
[e-mail:smomac@163.com]

² Institute of Intelligence Science and Technology, Hohai University, Nanjing, China
[e-mail:xzeng@hhu.edu.cn]

³ PLA Army Engineering University, Nanjing, China

⁴ Southeast University, Nanjing, China

*Corresponding author: Zhan Shi

*Received February 23, 2021; revised May 31, 2021; accepted June 24, 2021;
published July 31, 2021*

Abstract

Model-based software architecture verification and test scenarios generation are becoming more and more important in the software industry. Based on the existing temporal graph grammar, this paper proposes a new formalization method of the context-sensitive graph grammar for aiming at UML activity diagrams, which is called the UML Activity Graph Grammar, or UAGG. In the UAGG, there are new definitions and parsing algorithms. The proposed mechanisms are able to not only check the structural correctness of the UML activity diagram but also automatically generate the test scenario according to user constraints. Finally, a case study is discussed to illustrate how the UAGG and its algorithms work.

Keywords: Graph grammar, Parsing algorithm, TEGG, UML

1. Introduction

As the complexity and scale of software and hardware systems increase in many fields, such as the IoT(Internet of Things)[1-2], the finance^[3-4] etc., the concurrency of these systems is becoming more and more important. It makes the software architecture essential for describing and testing the software system. There are multiple types of model views to describe the software architecture.

At present, many Architecture Description Languages(ADLs) have been proposed[5-10]. Although the ADLs are specialized formal languages supporting modeling and reasoning on software architectures, their popularity and usage by practitioner is very low[11]. While, the Unified Modelling Language (UML) [12] is a common specification and the standard modeling language[13]. The UML activity diagram is one of UML standard models, which is very popular for describing software requirements and the dynamic behavior of the software systems^[14].

Since the formal method is the application of mathematical logic to describe the model of the software system, it has very high reliability and accuracy. Therefore, many studies use the method of a formal theorem to verify and check the correctness of UML activity diagrams[15-16]. While, a number of researchers use Petri net method to describe and verify the UML activity diagram. Störrle[17] mapped various elements of the UML activity diagram into executable Petri nets. Heuer et al.^[18] extended the dynamic syntax and semantics of the Petri net, and based on this Petri net, the activity diagram was formally described. Tariq et.al^[19] proposed an approach for formal analysis and simulation of the UML behavioral models using Coloured Petri Nets(CPN).

At the same time software testing is closely related to the software development process. Whether it is traditional software testing methods or automated software testing methods, generating test scenarios is one of the most critical issues. At present, many researchers have used UML modeling to represent the system and to test the software. Researchers have done a lot of research on test scenarios generation based on UML models. Yan et al.[20] proposed a test scenarios generation method based on UML sequence diagram models. In the literature [21], there is an approach to generate the test scenarios using UML use case diagrams. Among these studies [22]-[25], UML state diagrams are used, and algorithms for generating test scenarios are also proposed. In the process of generating test scenarios, it is also meaningful research how to use optimization methods^[26-27] to improve efficiency.

In fact, with the wide application of visual graph languages such as UML, BPMN, and control flowcharts, etc., a framework for formal definition and analysis of visual languages plays an increasingly important role. Obviously, for this definition and analysis of visual languages, graph grammars^[28] provide an intuitive but formal method. Just as string grammars are very important to string languages, graph grammars are also an indispensable theoretical tool for visual languages^[29].

Nowadays, there are many research results on graphic grammar and its application. Rekers and Schürr^[30] proposed a context-sensitive graph grammar in 1997. This graph grammar is called Layered Graph Grammar (LGG). Then, based on the LGG, Zhang et al.^[31] proposed another context-sensitive graph grammar. This grammar is called Reserved Graph Grammar (RGG), which not only introduces a marking mechanism in the graph grammar to distinguish different edges, but also proposes selection conditions to restrict the production. The RGG's^[32] application range is wider. There are Visual XML Schemas^[33] and Generic Visual Language Generation Environments^[34].

Zeng et al.^[35] proposed a context-sensitive Edge-based Graph Grammar (EGG). Based on the EGG, a new attempt of transformation between BPMN and BPEL^[36] was provided. For traditional graphic grammars, although they can verify the correctness of the graph structure, they cannot analyze and process temporal semantics. However, graphs with temporal semantics are widely used in many fields. In order to expand the application range of the graph grammar, an attempt of introducing temporal mechanism into graph grammar was proposed. This new graph grammar was called Temporal Edge-based Graph Grammar (TEGG)^[37]. So, we will extend the TEGG, make it more convenient to formally verify the UML activity diagram and further generate test scenarios.

Based on the TEGG, this paper proposes a new context-sensitive graph grammar for UML activity diagrams, called UML Activity Graph Grammar or UAGG in short. In the UAGG, formal definitions are introduced for the characteristics of UML activity diagrams. Similarly, two types of productions are also provided. Then, new algorithms are designed to verify UML activity diagrams and generate test scenarios. Finally, there is a case study to explain how the proposed algorithms work.

The rest of the paper is organized as follows. Firstly, Section 2 briefly introduces the basic concepts of the TEGG. Section 3 elaborates the formal definitions of the UAGG with the introduction of new mechanisms, such as an attribute set, new E-productions, and so on. Based on the UAGG, Section 4 provides steps and algorithms to realize the verification of UML activity diagrams and the generation of test scenarios. Section 5 illustrates a case study on a UML activity diagram. Finally, Section 6 concludes the whole paper.

2. Preliminaries

The TEGG mainly analyzes temporal semantics. In order to better understand the TEGG, the basic concepts are briefly introduced below.

Definition 2.1 A node n based on a label set L_n can be denoted as $n \equiv (lab, Natts)$, and there are the following conditions:

- L_n can be expressed as $L_n = T \cup \bar{T}$. T represents a label set of N_T , and \bar{T} represents a label set of N_N , while N_T represents a set of terminal nodes and N_N represents a set of non-terminal nodes;

- $lab \in L_n$, representing the label of n ;
- $Natts$ is a set of attributes owned by node n .

Definition 2.2 An edge e based on a label set L_e can be denoted as $e \equiv (lab, n_s, n_t, Eatts)$, and there are the following conditions:

- L_e is a set of edge labels;
- $lab \in L_e$, representing the label of e ;
- n_s is a node defined in Definition 2.1, meaning the source node of e ;
- n_t is a node defined in Definition 2.1, representing the target node of e ;
- $Eatts$ is a set of attributes owned by edge e , while $Eatts = Eka \cup \overline{Eka}$.

Usually, there are three graph operations. The first type of graph operation is called L-application, also called deduction. The second one is called R-application, also called reduction. Then, the last one is called T-application. There is the literature [37] for more details about the TEGG.

3. Formalism of the UAGG based on the TEGG

UML activity diagrams are mainly used to describe the workflow and concurrent behavior of the system and show the various sequential activities performed by all participants. Although activity diagrams have an intuitive and straightforward description, it cannot guarantee the correctness and reliability of the established model. It is more challenging to provide a useful basis for future software testing. In order to solve these problems, there are the formal definitions of the UAGG based on the TEGG.

Definition 3.1 A node n based on a label set L_n can be denoted as $n \equiv (id, name, lab, type, Natts)$, and there are the following conditions:

- L_n and $lab \in L_n$ are the same as those defined in Definition 2.1;
- $id \in \{0, 1, 2, \dots, k\}$, representing the identification of the node n to distinguish different nodes;
- $name$, representing the name of the node n ;
- $type$, indicating which categories this node belongs to, while $type = \{Initial, Final, Activity, ComponentActivity, Fork, Join, Branch, Merge, Object\}$. These values represent an initial activity node, a final activity node, a basic activity node, a component activity node, a concurrency fork node, a concurrency join node, a conditional fork node, a conditional join node and an object node in UML activity diagrams respectively;
- $Natts$ is a set of attributes owned by node n , which includes key temporal attributes and non-key attributes for UML activity diagrams.

In order to show more vividly the node status corresponding to different values of the temporal attribute in a node, the following notations are used. In addition, each node also has non-key attributes, including *Test_Path*, *Final_Weight*, and *Rand_Weight*. The *Test_Path* attribute is mainly used to save the generated test path. The *Final_Weight* and *Rand_Weight* attributes are used to save the two weights of the node, which will be described in detail later.



Fig. 1. Nodes with waiting and completed statuses

Definition 3.2 An edge e based on a label set L_e can be denoted as $e \equiv (id, name, lab, type, n_s, n_t, Eatts)$, and there are the following conditions:

- L_e , $lab \in L_e$, and $Eatts$ are the same as those defined in Definition 2.2;
- $id \in \{0, 1, 2, \dots, k\}$, representing the identification of the edge e to distinguish different edges;
- $name$, representing the name of the edge e ;
- $type$, indicating which categories this edge belongs to, while $type = \{Controlflow\}$. The value represents a control edge in UML activity diagrams;
- n_s and n_t are nodes defined in Definition 3.1, representing the source and target node of e respectively.

Definition 3.3 A *uml-activity graph* UAG based on a label set L can be denoted as $UAG \equiv (N, E, l, p, q)$, and there are the following conditions:

- N is a set of nodes defined in Definition 3.1;

- E is a set of edges defined in Definition 3.2;
- $l: N \cup E \rightarrow L$ is a mapping from N and E to L , and L can be expressed as $L = L_n \cup L_e$;
- $p: E \rightarrow N$ and $q: E \rightarrow N$ are mappings from E to N , indicating the source and target node of an edge.

In the UAGG, the concepts of a sub-graph, two isomorphic graphs, redexes, a T-production, and the graph operations R-application and T-application are similar to the corresponding concepts of the TEGG. Then, the introduction of these concepts will not be repeated. For UML activity diagrams, the T-productions shown in Fig. 2.

T-productions			
No.	Productions	Con	Fun
A.		$\forall e(\text{SourceNode}(e) = n_j \wedge e.\text{conds} = \text{'TRUE'})$	$n_j.\text{Final_Weight} = n_j.\text{Final_Weight};$ $n_j.\text{Rand_Weight} = n_j.\text{Rand_Weight}$
B.		$(n_i.\text{lab} \neq T.\text{xor}) \wedge (n_j.\text{lab} \neq T.\text{xor})$	$n_j.\text{Final_Weight} = \max\{\forall e_i(\text{SourceNode}(e_i).\text{Final_Weight} + n_j.\text{Rand_Weight})\}$ $(i = 0, 1, \dots, m);$ $n_i.\text{Final_Weight} = n_j.\text{Final_Weight};$ $n_i.\text{Rand_Weight} = n_j.\text{Rand_Weight}$
C.		$(n_i.\text{lab} = T.\text{xor}) \wedge (n_j.\text{lab} = T.\text{xor})$	$n_j.\text{Final_Weight} = \text{SourceNode}(e).\text{Final_Weight} + n_j.\text{Rand_Weight};$ $n_i.\text{Final_Weight} = n_j.\text{Final_Weight};$ $n_i.\text{Rand_Weight} = n_j.\text{Rand_Weight};$
D.		$(n_i.\text{lab} \neq T.\text{xor}) \wedge (n_j.\text{lab} \neq T.\text{xor}) \wedge (\forall e(\text{SourceNode}(e) = n_j \wedge e.\text{Eatts.conds} = \text{'TRUE'}))$	$n_i.\text{Final_Weight} = n_j.\text{Final_Weight};$ $n_i.\text{Rand_Weight} = n_j.\text{Rand_Weight};$
E.		$(n_i.\text{lab} = T.\text{xor}) \wedge (n_j.\text{lab} = T.\text{xor}) \wedge (\exists e(e.\text{Eatts.cond} = \text{'TRUE'}))$	$n_i.\text{Final_Weight} = n_j.\text{Final_Weight};$ $n_i.\text{Rand_Weight} = n_j.\text{Rand_Weight};$

Fig. 2. A group of T-productions defined for the UML activity diagrams

In order to check the structural correctness of UML activity diagrams and reduce concurrent processes in these diagrams, the concept of a new E-production in the UAGG is given as follows.

Definition 3.4 An E-production p_E is the expression $\overline{UAG}_L \Leftrightarrow \overline{UAG}_R$, denoted as $(\overline{UAG}_L, \overline{UAG}_R, \text{Con}, \text{Fun})$, where

· \overline{UAG}_L and \overline{UAG}_R are uml-activity graphs with dangling edges;

$\cdot \overline{(UAG_L.M = UAG_R.M)} \wedge (M \subset \{0,1,2,\dots\})$;
 \cdot Con is conditions which can be satisfied;
 \cdot Fun is a group of functions that are executed when the Con can be satisfied and this production is used.

According to the above definition, the E-productions for UML activity diagram analysis are shown in Fig. 3.

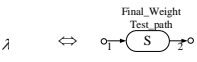
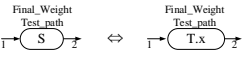
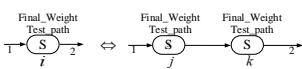
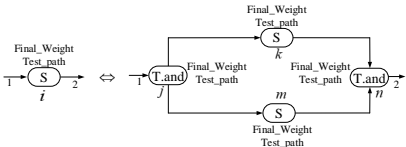
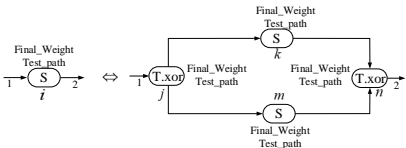
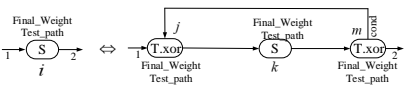
E-productions			
No.	Productions	Con	Fun
1.		$S.Natts.statusflag = COMPLETED$	$\lambda Test_path = S.Test_path$
2.		$S.Natts.statusflag = COMPLETED \ \&\& \ T.x.Natts.statusflag = COMPLETED$	$S.Test_path = T.x.Test_path$
3.		$(i.Natts.statusflag = COMPLETED) \ \&\& \ (j.Natts.statusflag = COMPLETED) \ \&\& \ (k.Natts.statusflag = COMPLETED)$	$i.Test_path = combine(j.Test_path, k.Test_path)$
4.		$(i.Natts.statusflag = COMPLETED) \ \&\& \ (j.Natts.statusflag = COMPLETED) \ \&\& \ (k.Natts.statusflag = COMPLETED) \ \&\& \ (m.Natts.statusflag = COMPLETED) \ \&\& \ (n.Natts.statusflag = COMPLETED)$	$i.Test_path = sequence(j.Test_path, k.Test_path, m.Test_path, n.Test_path)$
5.		$(i.Natts.statusflag = COMPLETED) \ \&\& \ (j.Natts.statusflag = COMPLETED) \ \&\& \ (k.Natts.statusflag = COMPLETED) \ \&\& \ (m.Natts.statusflag = COMPLETED) \ \&\& \ (n.Natts.statusflag = COMPLETED)$	$i.Test_path = sequence(j.Test_path, k.Test_path, m.Test_path, n.Test_path)$
6.		$(i.Natts.statusflag = COMPLETED) \ \&\& \ (j.Natts.statusflag = COMPLETED) \ \&\& \ (k.Natts.statusflag = COMPLETED) \ \&\& \ (m.Natts.statusflag = COMPLETED)$	$i.Test_path = sequence(j.Test_path, k.Test_path, m.Test_path)$

Fig. 3. A group of E-productions defined for the UML activity diagrams

4. A method of UML activity diagrams verification and test scenarios generation based on the UAGG

4.1 Steps of UML activity diagrams verification and test scenarios generation

In order to formally verify UML activity diagrams and further generate test scenarios, this section proposes an algorithm for this purpose based on the UAGG. The main steps of the algorithm are designed as follows, which not only check the structural aspect but also generate test scenarios of UML activity diagrams. The pseudo-codes of the algorithm are shown in Fig. 4.

Algorithm 1. Pseudo-codes for the parsing algorithm	
Graph parsing (UMLActivityDiagram g, Productions P, Constraints userConstraints)	
{	
1:	G ← initialize(g);
2:	if(G ≠ NULL & & P ≠ NULL)
3:	{
4:	if (graphStructureParsing(G, P.Eproductions, FALSE) = λ)
5:	{
6:	result = generateTestScenarios(G, P, userConstraints);
7:	return result;
8:	}
9:	else
10:	return NULL;
11:	}
	}

Fig. 4. A new parsing algorithm

Step 1, initializing. This step mainly transforms a given UML activity diagram g into a host graph of the UAGG, which is done by the function *initialize(UMLActivityDiagram g)*.

Step 2, verifying the structural correctness of the UAGG host graph. This step is mainly implemented by the function *graphStructureParsing()* which has three parameters. The first parameter represents the UAGG graph G , the second one represents the UAGG E-productions, and the last one indicates whether to enable the conditions of E-productions in Fig. 3. If the structure of G is correct, jump to step 3. Otherwise, return the NULL and end the algorithm. A further description of this function will be introduced in Section 4.2.

Step 3, generating test scenarios with the UAGG. This step is mainly implemented by the function *generateTestScenarios()*, which also has three parameters. The first parameter represents the UAGG grammar graph G , the second one represents the productions including E-productions and T-productions, and the last one represents the user constraints on the generation of test scenarios. A further description of this function will be introduced in Section 4.3.

4.2 Steps of UML activity diagrams' structure verification

A UAGG host graph of a UML activity model can be obtained through the first step in Section 4.1. In the algorithm provided in Fig. 4, the function *graphStructureParsing()* completes the function similar to the traditional graph parsing algorithm. It is designed to check the given graph's structure correctness through R-applications with E-productions presented in Fig. 3. The pseudo-codes of the function *graphStructureParsing()* are provided in Fig. 5.

The function *graphStructureParsing()* is to reduce the UAGG host graph g according to the E-productions $ePros$. When selecting the E-production, it will combine the parameter *isCondition* to determine whether to enable the conditions of this E-production in Fig. 3.

In fact, the function *findRedex(Graph g, Productions pros, Boolean isCondition)* is mainly used to realize the process of finding redexes in the host graph g . If the parameter *isCondition* is true, then additional conditions of the productions need to be met when selecting and determining the production. And if the parameter *isCondition* is false, then there is no need to satisfy its conditions when selecting and determining the production additionally.

There are two stacks *GraphStack* and *ERedexStack* in the function *graphStructureParsing()*. The *GraphStack* is used to store graphs, while the *ERedexStack* is used to store corresponding redexes of the graphs respectively.


```

Algorithm 2. Pseudo-codes for the function graphStructureParsing( )
Graph graphStructureParsing( Graph g, E-Productions ePros, Boolean isCondition )
{
1:   if( g!= NULL && ePros!= NULL )
2:     {
3:       while ( g != λ )
4:         {
5:           push( MARK, ERedexStack );
6:           for all p ∈ ePros
7:             {
8:               EredexSet = findRedex ( g, ePros, isCondition);
9:               for all eredex ∈ EredexSet
10:                push( eredex, ERedexStack );
11:             }
12:            eredex ⊖ pop( ERedexStack );
13:            while ( eredex == MARK )
14:              {
15:                g ⊖ pop( GraphStack );
16:                eredex ⊖ pop( ERedexStack );
17:                if ( eredex==NULL )
18:                  {
19:                    if( isCondition== FALSE )
20:                      return NULL;
21:                    else
22:                      return g;
23:                  }
24:              }
25:            push ( g, GraphStack );
26:            g = RApplication ( g, p, eredex );
27:          }
28:        return g;
29:      }
}

```

Fig. 5. The function of *graphStructureParsing()*

4.3 Steps of test scenarios generation

There is a critical step to extract test scenarios from UML activity diagrams in test case generation. However, the concurrency modules in the activity diagram make the problem of extracting the basic test scenarios more complicated. The concurrency module of UML activity diagrams always starts at the concurrent fork node. It ends at the concurrent join node, which meets the characteristic of a single node as input or output.

Therefore, based on the UAGG, this paper proposes the algorithm *generateTestScenarios()* which will analyze the UAGG host graph according to the constraint conditions of the productions and users to generate the test scenarios. The specific steps of the algorithm are as follows:

Step 1, making pretreatment for a given graph of the UAGG. Firstly, the primary implementation is to find all the concurrent modules in the UAGG host graph and then assign an initial value to the state of the first concurrent fork node of each outermost concurrent module.

Step 2, assign a random value to the *Rand_Weight* attribute of each node in each concurrent module. Then, for all concurrent modules, use the *graphTemporalParsing()* function with T-productions, T-applications and other operations to calculate the *Final_Weight* attribute value of each node. Thus, the intermediate graph is generated.

Step 3, further using E-productions to reduce the intermediate graph generated above until all the concurrent modules are reduced to the corresponding single node.

Step 4, further executing the *graphStructureParsing()* function with E-productions to generate a test scenario.

Step 5, determining whether the user constraints are met. If the user has no additional constraints, this algorithm ends and the test scenario is obtained. If the user has additional condition constraints, check whether the test scenario generated in Step 4 meets the conditions. When the conditions are met, the algorithm ends. When the conditions are not met, go to Step

2 and continue execution.

The above process can be expressed with pseudo-codes, as shown in the following figure.

Algorithm 3. Pseudo-codes for generating test scenario

```

generateTestScenarios( Graph g, Productions ps, Constraints userConstraints )
{
1:   Boolean isCycle=TRUE;
2:   finalTestScen=NULL;
3:   if( g !=NULL && ps != NULL )
4:   {
5:     List allConcurrentModel = findAllConcurrentModel( g );
6:     if( allConcurrentModel != NULL )
7:     {
8:       g = initializeFirstNode( g, allConcurrentModel );
9:       while( isCycle )
10:      {
11:        gra = initializeRandWeightOfNode( g, allConcurrentModel );
12:        gra = graphTemporalParsing( gra, ps );
13:        gra = graphStructureParsing( gra, ps.Eproductions, 1 );
14:        basicTestScen = graphStructureParsing( gra, ps.Eproductions, 0 );
15:        if( isSatisfyConstraints( basicTestScen, userConstraints ) == FALSE )
16:        {
17:          isCycle=TRUE;
18:        }
19:        else
20:        {
21:          isCycle=FALSE;
22:          finalTestScen = basicTestScen;
23:        }
24:      }
25:    }
26:    else
27:    {
28:      finalTestScen = graphStructureParsing( gra, ps.Eproductions, 0 );
29:    }
30:    return finalTestScen ;
31:  }
}

```

Fig. 6. The function of *generateTestScenarios()*

4.4 Computational complexity

The computational complexity of the algorithm in Fig. 4 is a critical issue. While, it is analyzed in this section.

Theorem 4.1. For a given UML activity diagram, the time complexity of the algorithm in Fig. 4 is $O(ps^2 \cdot n^{2m} + 2 \cdot (\frac{ps}{m!})^n \cdot (n^n \cdot n!)^m)$, where n is the numbers of nodes in a host graph of the

UAGG, ps is the number of productions, and m is the maximum number of nodes among all productions' right graphs.

Proof. Assuming k_1 , k_2 , and k_3 are time complexity of functions *initialize()*, *graphStructureParsing()*, and *generateTestScenarios()* respectively. According to the algorithm steps, its time complexity k can be calculated by the following equation:

$$k = k_1 + k_2 + k_3$$

It is sure that the complexity of the function *initialize()* is $O(n)$. In addition, for the function *graphStructureParsing()*, assuming l_1 is the maximal iterations number of the lines 3–27, l_2 is that of the lines 6–11, l_3 is that of the lines 9–10, and l_4 is that of the lines 13–24. Then,

assuming k_4 and k_5 are the time complexities of the function $findRedex()$ and $RApplication()$ respectively. So, the complexity of function $graphStructureParsing()$ is as follows^[37]:

$$k_2 = O(l_1 \cdot (l_2 \cdot (k_4 + l_3) + l_4 + k_5))$$

$$= O\left(\left(\frac{ps}{m}\right)^n \cdot (n^n \cdot n!)^m\right)$$

For the function $generateTestScenarios()$, assuming l_5 is the maximal iterations number of the lines 9 – 24, k_6 , k_7 , k_8 , and k_9 are the time complexities of the function $findAllConcurrentModel()$, $initializeFirstNode()$, $initializeRandWeightOfNode()$ and $graphTemporalParsing()$ respectively. In order to find all concurrent subgraphs, it is necessary to traverse all nodes in the UAGG graph. Therefore, the time complexity of $findAllConcurrentModel()$ is $O(n+e)$, while e represents the edges in the UAGG host graph. The function $initializeFirstNode()$ is used to initialize the first node of the concurrent module, so $k_7 \leq O(n+e)$. Similarly, $k_8 \leq O(n+e)$. For the function $graphTemporalParsing()$, there is $k_9 \leq O(ps^2 \cdot n^{2 \cdot m + 1})$ according to the reference [37]. Obviously, for the obtained test scenario containing n nodes, the time complexity of the function $isSatisfyConstraints()$ is $O(n)$, while $l_5 \leq n \cdot (n-1) = n^2 - n$. So, the complexity of the function $generateTestScenarios()$ is:

$$k_3 = O(k_6 + k_7 + l_5 \cdot (k_8 + k_9 + 2 \cdot k_2 + n))$$

$$= O(ps^2 \cdot n^{2 \cdot m} + \left(\frac{ps}{m}\right)^n \cdot (n^n \cdot n!)^m)$$

Combined with the above discussion, the time complexity of the algorithm is the following:

$$k = k_1 + k_2 + k_3$$

$$= O(ps^2 \cdot n^{2 \cdot m} + 2 \cdot \left(\frac{ps}{m}\right)^n \cdot (n^n \cdot n!)^m)$$

5. Case study and Comparison with related work

This section introduces a process model built by UML activity diagrams, and illustrates how this model is verified and test scenarios are generated using the steps presented in Section 4.1.

5.1 Case study

Fig. 7 shows the process of adding product information to the supermarket management system after the administrator logs in. There are ten activity nodes and six gateway nodes. Besides, this process includes two concurrent models.

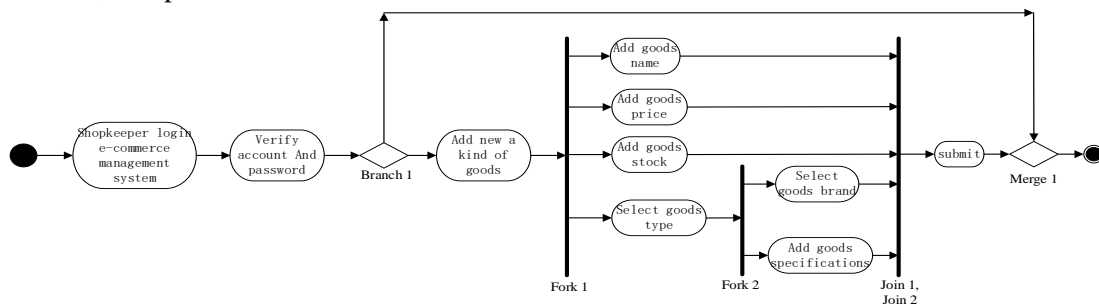


Fig. 7. A business process described by the UML activity diagram

According to the algorithm introduced in Section 4.1, it can analyze this diagram as follows. Step 1, executing the first steps, namely initialization. Through this step, we can get the host UAGG graph g shown in Fig. 8.

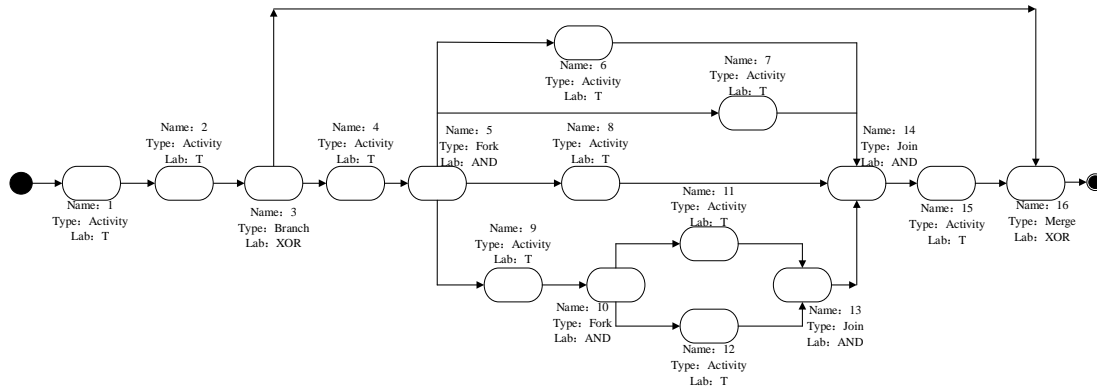


Fig. 8. A UAGG host graph g is corresponding to Fig. 7.

The graph g describes the node name with numbers given to each activity. The two columns in the following table are *Activity Name* and *Node Name* respectively. The *Activity Name* represents the name of the activity in Fig. 7, and the *Node Name* indicates the node in the graph g corresponding to the activity.

Table 1. The UAGG graph of adding product information activity diagram

Activity Name	Node Name	Activity Name	Node Name	Activity Name	Node Name	Activity Name	Node Name
Shopkeeper login e-commerce management system	1	Fork 1	5	Select goods type	9	Join 1	13
Verify account and password	2	Add goods name	6	Fork 2	10	Join 2	14
Branch 1	3	Add goods price	7	Select goods brand	11	Submit	15
Add a new kind of goods	4	Add goods stock	8	Add goods specifications	12	Merge 1	16

Step 2, this step is the process of structural analysis of the above host graph. That is to verify whether the structure is correct by executing the function *graphStructureParsing()* based on E-productions. If the host graph's structure is correct, jump to Step 3. Otherwise, return the error message and end the algorithm.

Step 3, this step is to generate test scenarios. It involves several processes described in Section 4.3, so we can further introduce this step as follows.

Step 3.1, making pretreatment for a given graph of the UAGG. Firstly, there are two concurrent modules in the graph g founded by the function *findAllConcurrentModel(Graph g)*. The first fork node of the outermost concurrent module of these two modules is marked by the red dotted line box in Fig. 9, noted as *Node 5*. Then, assign an initial value to the state of the *Node 5*.

Step 3.2, assigning a random value to the *Rand_Weight* attribute of each node in these two concurrent models, and assigning the value of the *Rand_weight* attribute in the *Node 5* to its *Final_weight* attribute as shown in Fig. 9. Then, for all concurrent models in this graph, use the *graphTemporalParsing()* function with T-productions, T-applications and other operations to calculate the *Final_Weight* attribute value of each node. So, the intermediate graph is generated.

In the following content, C_i^j will be used to represent each redex, where j represents the number of the intermediate graph containing the redex, and i represents the number of the corresponding redex.

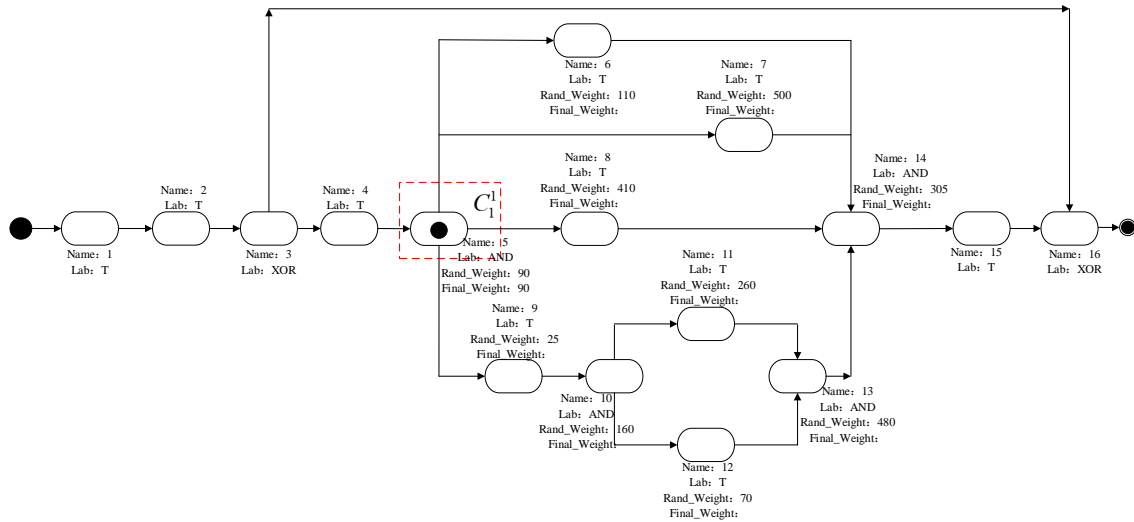


Fig. 9. An intermediate graph

Firstly, the function *graphTemporalParsing()* will search for all redexes in the host graph according to the T-productions given in Fig. 2. In this process, because the condition of T-production A is met, the redex C_1^1 will be selected in Fig. 9 and replaced by T-application with the T-production A. At the same time, the *Fun* function of the T-production A will also be executed. So, Fig. 10 can be obtained.

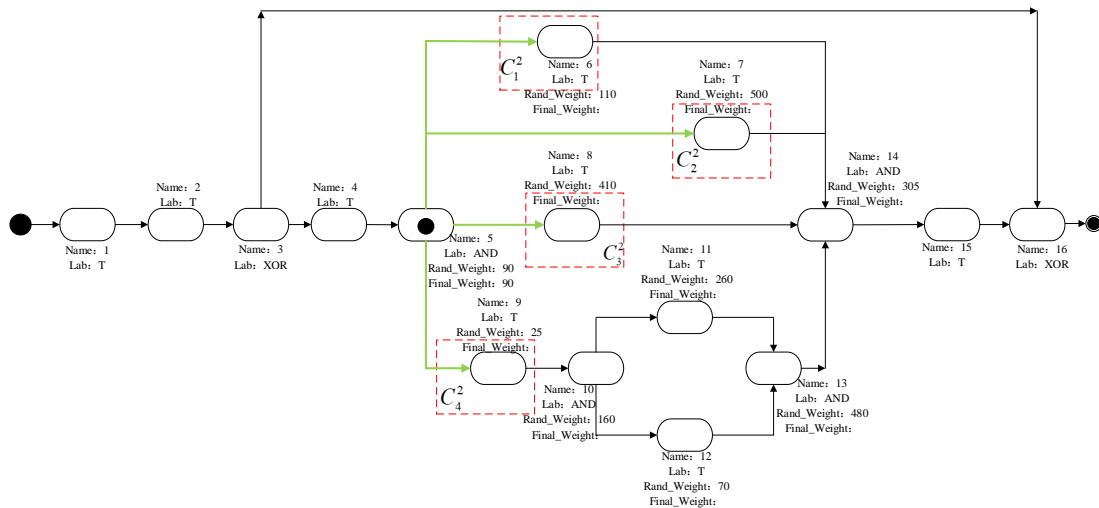


Fig. 10. An intermediate graph after using T-production A

Secondly, the function continues to search for redexes based on Fig. 10. Since conditions of the T-production B are satisfied, the redexes C_1^2 , C_2^2 , C_3^2 , and C_4^2 are selected and replaced by T-application with the T-production B. Then, its *Fun* function is activated and

executed, which calculates the value of *Final_weight* of these redexes. Therefore, the *Final_Weight* attribute value of *Node 6, 7, 8, and 9* is equal to the *final_weight* attribute value of *Node 5* plus the *Rand_weight* attribute value of these nodes themselves respectively. So, **Fig. 11** can be obtained.

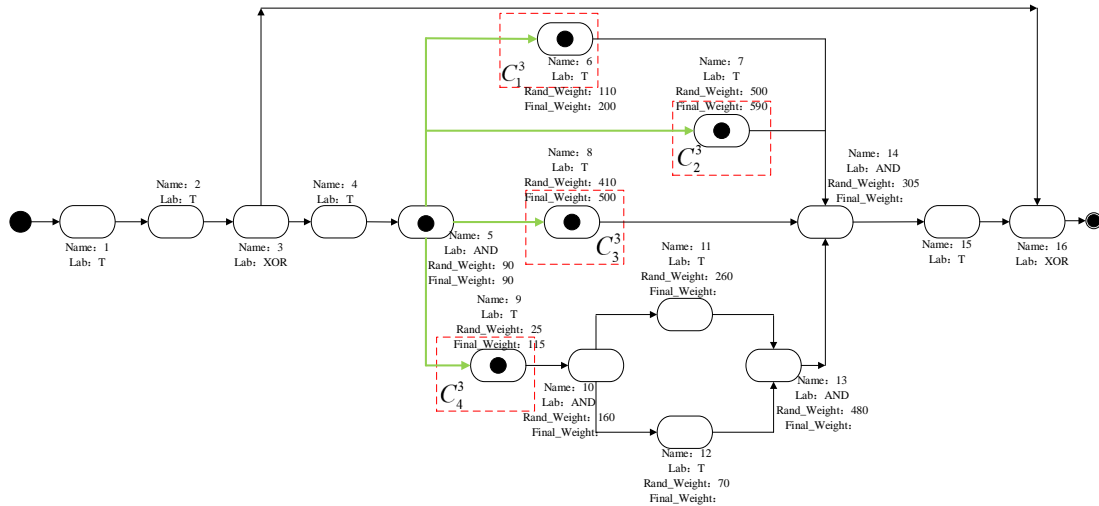


Fig. 11. An intermediate graph after using T-production B

Then, the above process is repeatedly executed until the attribute *Final_Weight* of all nodes in the concurrent modules of the graph can be calculated. So, **Fig. 12** can be obtained.

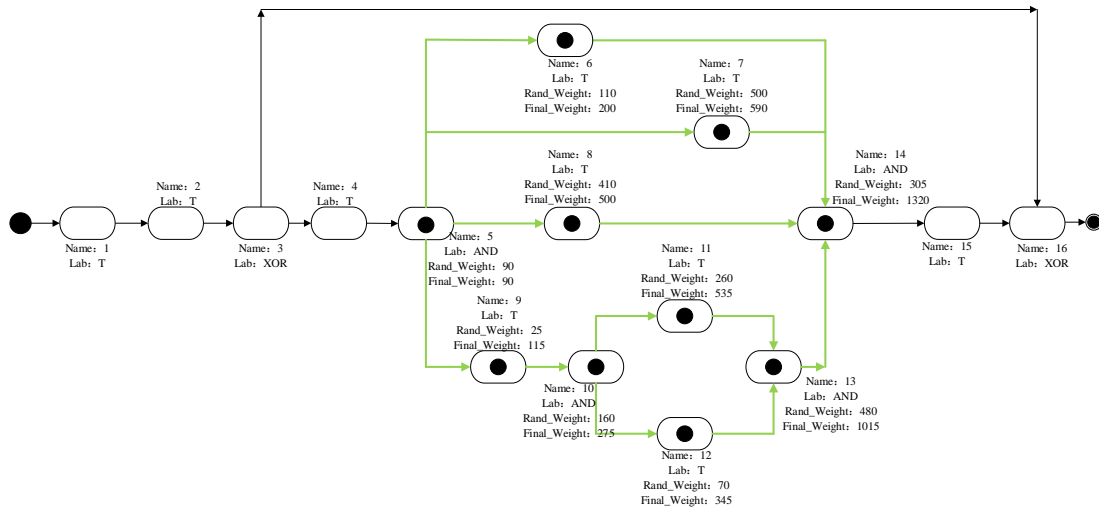


Fig. 12. An intermediate graph

Step 3.3, in this step the main realization uses E-productions to reduce the intermediate graph generated in the above Step 3.2, until the concurrent modules are reduced to a single node. More detailed analysis processes are given below.

(1) The function *graphStructureParsing()* first tries to search for all redexes in **Fig. 12** with the E-productions given in **Fig. 3**. Considering *Con* of the E-production 2, it can be used to perform R-application operations, so that the value of nodes' *Lab* in the concurrent modules

can be changed from “T” to “S”. Then, Fig. 13(a) can be obtained.

Considering *Con* of the E-production 4, the redex C_1^5 is selected and replaced by R-application, which is shown in Fig. 13(a). Its *Fun* is activated and executed, which calculates the value of *Test_path* of the redex. So, Fig. 13(b) can be obtained.

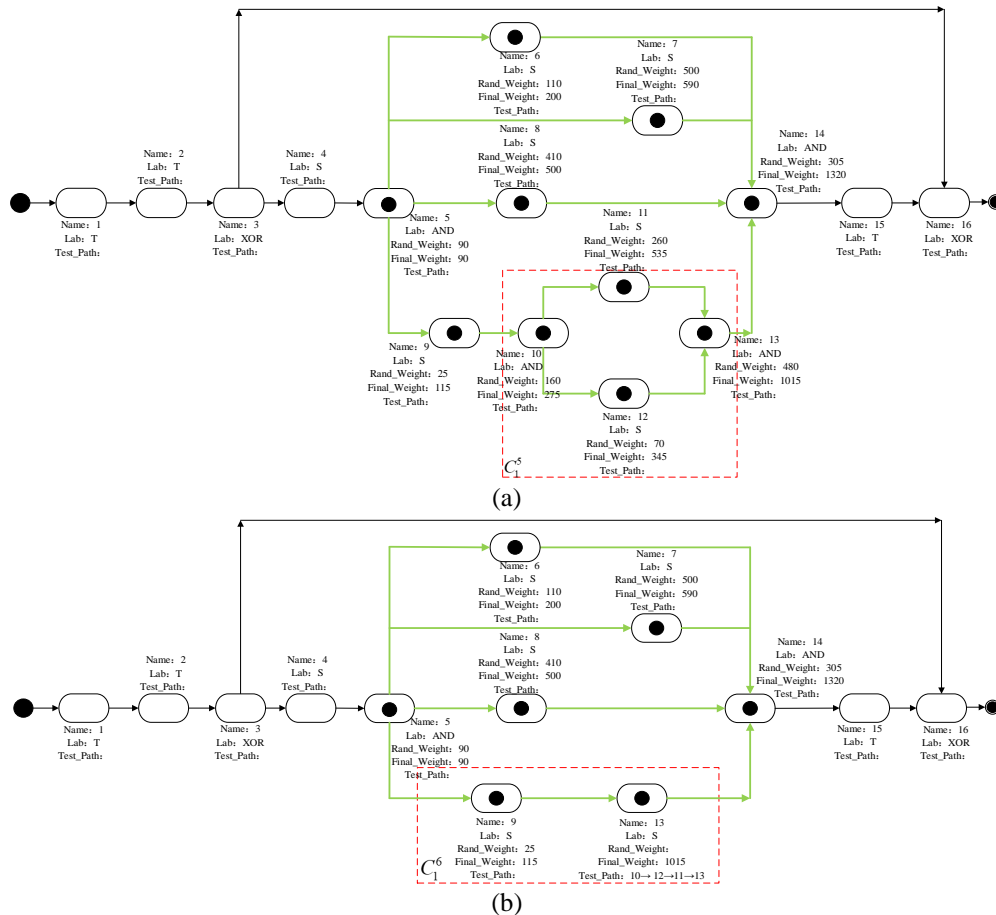


Fig. 13. (a). An intermediate graph with C_1^5 ; (b). An intermediate graph after using E-production 4

(2) The above process is repeatedly executed until all the concurrent modules are reduced to the corresponding single node. And, the attribute *Test_path* of the corresponding single node is calculated. So, Fig. 14 can be obtained.

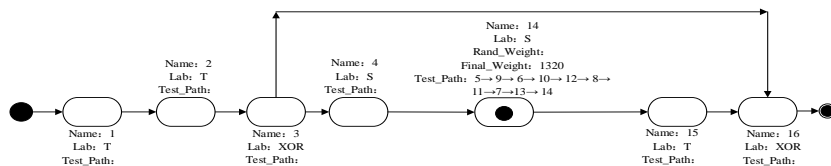


Fig. 14. An intermediate graph

Step 3.4, executing the *graphStructureParsing()* function without considering *Con* of the E-productions to generate a test scenario. More detailed analysis processes are given below.

(1) The function *graphStructureParsing*() first tries to search for all redexes without considering *Con* of the E-productions given in Fig. 3. Obviously, E-production 2 and 3 are satisfied and there are the redexes C_1^7 , C_2^7 , C_3^7 , and C_4^7 which is shown in Fig. 15(a). Then, the redexes C_1^7 , C_2^7 , and C_4^7 are replaced by R-application with E-production 2, while the redex C_3^7 are replaced with E-production 3. Its *Fun* is activated and executed, which calculates the value of *Test_path* of the redexes. So, Fig. 15(b) can be obtained.

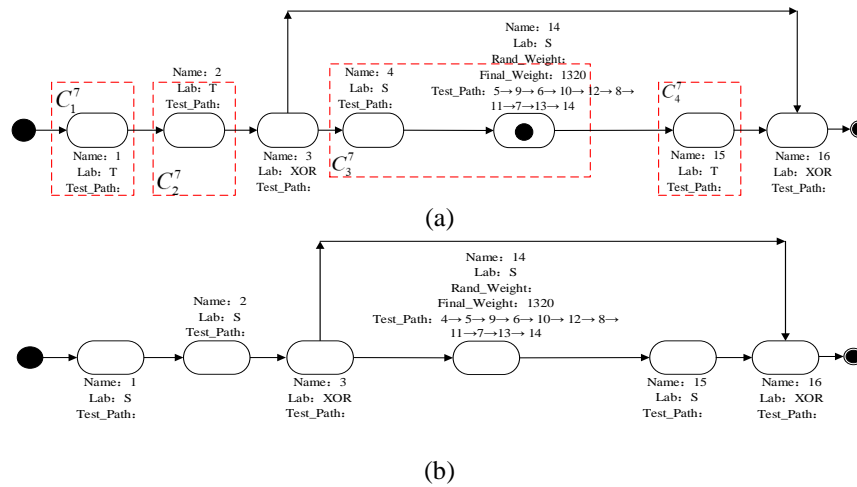


Fig. 15. (a). A graph with C_1^7 , C_2^7 , C_3^7 , and C_4^7 ; **(b).** A graph after using E-production 2 and 3

(2) The above process is repeatedly executed until a test scenario is generated. That is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 6 \rightarrow 10 \rightarrow 12 \rightarrow 8 \rightarrow 11 \rightarrow 7 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16$.

Step 3.5, determining whether the user constraints are met. If the user has no additional constraints, the algorithm ends and the test scenario is obtained. If the user has additional condition constraints, check whether the test scenario generated in Step 3.4 meets the conditions. When the additional condition constraints are met, the algorithm ends; when the additional condition constraints are not met, go to Step 3.2 and continue execution.

In Table 2, there are some test scenarios generated by the above steps. This table contains three columns. The first column represents the ID number of the test scenario, the second column represents user constraints, and the third column represents a generated test scenario.

Table 2. Test scenarios from the UAGG graph

Test Scenarios ID	User Constraints	Generated Test Sequences
TS-1	9, 6, 8, 7	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 6 \rightarrow 10 \rightarrow 12 \rightarrow 8 \rightarrow 11 \rightarrow 7 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16$
TS-2	9, 6, 8, 7	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16$
TS-3	9, 6, 8, 7	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16$
TS-4	9, 7, 6, 8	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 14 \rightarrow 15 \rightarrow 16$
TS-5	9, 7, 6, 8	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 7 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 6 \rightarrow 8 \rightarrow 14 \rightarrow 15 \rightarrow 16$
TS-6	9, 7, 6, 8	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 7 \rightarrow 13 \rightarrow 6 \rightarrow 8 \rightarrow 14 \rightarrow 15 \rightarrow 16$

5.2 Comparison with related work

Many researchers used UML diagrams to generate test scenarios in the literatures [38-42]. This section provides the comparison with these related works. Kundu et al.^[38] presented an approach of generating test cases from UML activity diagrams. They considered a test coverage criterion, called activity path coverage criterion. Pechtunun et al.^[39] proposed a method with AC grammar to generate test cases from UML activity diagram. In addition, this literature further illustrated the effectiveness of the method through four case studies. Patel et al.^[40] focused on two approaches to generate test cases form UML activity diagram. Jena et al.^[41] introduced a method to transform from a AFT generated by UML activity diagram to AFG. Then, the test paths are generated by traversing the AFG. By building an IOAD, Mahali et al.^[42] presented an approach to generate a minimum test suite with maximum coverage.

Table 3 compares some related works, which illustrates the approach proposed in this paper has several features. On the one hand, all test paths can be automatically generated through this approach. On the other hand, this approach is able to check the structural correctness of the UML activity diagrams.

Table 3. Comparison on the related work

Reference and Proposed work	Generation Algorithm	Mechanism	Structure Verification	Coverage Criteria	Case Study
[38]	DFS,BFS	Manually	No	Activity path	Registration Cancellation
[39]	AC Grammar	Automatically	No	All paths	ATM withdrawing
[40]	Two approaches	Automatically	No	Path,Activity path	Login system
[41]	DFS , Genetic	Automatically	No	Path,Transition, Decision	ATM cash withdrawal system
[42]	BFS	Automatically	No	All paths	Order system
Proposed work	UAGG	Automatically	Yes	All paths	Supermarket management system

6. Conclusions

This paper proposes a new graph grammar UAGG to verify and analyze UML activity diagrams based on the TEGG. In the UAGG, there are some new mechanisms which include new E-productions and a parsing algorithm and so on. According to the designed productions set, the algorithm proposed in this paper can not only verify the correctness of the UML activity diagrams' structure, but also automatically generate test scenarios that meet user constraints. Finally, there is a specific case to explain how the proposed UAGG and its mechanisms works.

Our future research goal is to extend the application of generating test cases by the UAGG and develop the algorithms with a wider range of applications and higher efficiency.

Acknowledgement

This work is supported by the Natural Science Foundation of Jiangsu Province(CN) under grant BK20181019, National Natural Science Foundation of China under grant 61802428, and Science Foundation of Nanjing Institute of Technology under grant YKJ201723.

References

- [1] D. Park, H. Bang, C. S. Pyo and S. Kang, "Semantic open IoT service platform technology," in *Proc. of IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, Korea, pp. 85-88, March 2014. [Article \(CrossRef Link\)](#).
- [2] S. K. Vishwakarma, P. Upadhyaya, B. Kumari and A. K. Mishra, "Smart Energy Efficient Home Automation System Using IoT," in *Proc. of the 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU)*, pp. 1-4, April 2019. [Article \(CrossRef Link\)](#)
- [3] T. Jin, X. Yang, H. Xia and H. Ding, "Reliability index and option pricing formulas of the first hitting time model based on the uncertain fractional-order differential equation with Caputo type," *Fractals Complex Geometry, Patterns, and Scaling in Nature and Society*, vol.29, no.1, January 2021. [Article \(CrossRef Link\)](#)
- [4] T. Jin, H. Ding, H. Xia and J. Bao, "Reliability index and Asian barrier option pricing formulas of the uncertain fractional first-hitting time model with Caputo type," *Chaos, Solitons & Fractals*, vol.142, January 2021. [Article \(CrossRef Link\)](#).
- [5] D.C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol.21, pp. 717-734, September 1995. [Article \(CrossRef Link\)](#).
- [6] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol.6, no.3, pp.213-249, July 1997. [Article \(CrossRef Link\)](#)
- [7] G. Georg and S. Seidman, "The use of architecture description languages to describe a distributed measurement system," in *Proc. of the 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, Edinburgh, UK, April 2000. [Article \(CrossRef Link\)](#).
- [8] D.Garlan, R.Monroe and D.Wile, "Acme: an architecture description interchange language," in *Proc. of CASCON*, 1997.
- [9] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol.21, no.4, pp. 314-335, April 1995. [Article \(CrossRef Link\)](#).
- [10] S. Rigo, G. Araujo, M. Bartholomeu and R. Azevedo, "UML as an Architecture Description Language," in *Proc. of the 16th Symposium on Computer Architecture and High Performance Computing*, Foz do Iguacu, Brazil, October 2004.
- [11] B.Bharathi and D.Sridharan, "UML as an Architecture Description Language," *International Journal of Recent Trends in Engineering*, vol. 1, no. 2, pp.230-232, May 2009.
- [12] B. Selic, S. Cook, E. Seidewitz and D. Tolbert, "OMG Unified Modeling Language (Version 2.5)," Object Management Group, MA, USA, March 2015.
- [13] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham and R. Pretorius, "Empirical evidence about the UML: a systematic literature review," *Software Practice & Experience*, vol. 41, no. 4, pp. 363-392, April 2011. [Article \(CrossRef Link\)](#)
- [14] N. Maneerat and W. Vatanawood, "Translation UML Activity Diagram into Colored Petri Net with inscription," in *Proc. of the 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, Khon Kaen, Thailand, November 2016. [Article \(CrossRef Link\)](#)
- [15] L. Baresi, A. C Morzenti, A. Motta and M. G Rossi, "A logic-based semantics for the verification of multi-diagram UML models," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no.4, pp. 1-8, July 2012. [Article \(CrossRef Link\)](#)
- [16] R. Eshuis, "Symbolic model checking of uml activity diagrams," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp.1-38, January 2006. [Article \(CrossRef Link\)](#)
- [17] H.Störrle, "Semantics of control-flow in UML 2.0 activities," in *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Rome, Italy, 2004.
- [18] A.Heuer, V.Stricker, C.J. Budnik, S.Konrad, K.Lauenroth and K.Pohl, "Defining variability in activity diagrams and Petri nets," *Science of Computer Programming*, vol.78, no.12, pp. 2414-2432, December 2013. [Article \(CrossRef Link\)](#)

- [19] O. Tariq, J. Sang, K. Gulzar and H. Xiang, "Automated analysis of UML activity diagram using CPNs," in *Proc. of the 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, China, pp. 134-138, November 2017. [Article \(CrossRef Link\)](#)
- [20] Yan Li and Li Jiang, "The research on test case generation technology of UML sequence diagram," in *Proc. of the 9th International Conference on Computer Science & Education (ICCSE)*, Vancouver, Canada, pp. 1067-1069, August 2014. [Article \(CrossRef Link\)](#)
- [21] Z. A. Hamza and M. Hammad, "Generating Test Sequences from UML Use Case Diagram: A Case Study," in *Proc. of the 2nd International Sustainability and Resilience Conference: Technology and Innovation in Building Designs*, pp. 1-6, November 2020. [Article \(CrossRef Link\)](#)
- [22] M. A. Ali, K. Shaik and S. Kumar, "Test case generation using UML state diagram and OCL expression," *International Journal of Computer Applications*, vol. 95, no. 12, pp. 7-11, June 2014. [Article \(CrossRef Link\)](#).
- [23] V. Chimisliu and F. Wotawa, "Improving test case generation from UML statecharts by using control, data and communication dependencies," in *Proc. of the 13th International Conference on Quality Software*, Naging, China, pp. 125-134, July 2013. [Article \(CrossRef Link\)](#).
- [24] P. Gulia and R. S. Chillar, "A new approach to generate and optimize test cases for UML state diagram using genetic algorithm," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1-5, May 2012. [Article \(CrossRef Link\)](#)
- [25] R. Swain, V. Panthi, P. K. Behera, and D. P. Mohapatra, "Automatic test case generation from UML state chart diagram," *International Journal of Computer Applications*, vol. 42, no. 7, pp. 26-36, March 2012. [Article \(CrossRef Link\)](#)
- [26] W. Deng, S. Shang, X. Cai, H. Zhao, Y. Song and J. Xu, "An improved differential evolution algorithm and its application in optimization problem," *Soft Computing*, vol. 25, pp. 5277-5298, January 2021. [Article \(CrossRef Link\)](#)
- [27] X. Cai, H. Zhao, S. Shang, Y. Zhou, W. Deng, H. Chen and W. Deng, "An improved quantum-inspired cooperative co-evolution algorithm with multi-strategy and its application," *Expert Systems with Applications*, vol. 171, June 2021. [Article \(CrossRef Link\)](#)
- [28] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*, Singapore: World Scientific Publishing Co., Inc. February 1997. [Article \(CrossRef Link\)](#)
- [29] C. Ermel, M. Rudolf and G. Taentzer, "The AGG Approach: Language and Environment," *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 2, pp. 551-603, January 1999. [Article \(CrossRef Link\)](#).
- [30] J. Rekers and A. Schürr, "Defining and parsing visual languages with layered graph grammars," *Journal of Visual Languages & Computing*, vol. 8, no. 1, pp. 27-55, February 1997. [Article \(CrossRef Link\)](#)
- [31] D. Zhang, K. Zhang and J. Cao, "A context-sensitive graph grammar formalism for the specification of visual languages," *The Computer Journal*, vol. 44, no. 3, pp.186-200, January 2001. [Article \(CrossRef Link\)](#)
- [32] X. Zeng, K. Zhang, J. Kong and G. Song, "RGG+: An enhancement to the reserved graph grammar formalism," in *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, Dallas, TX, USA, pp. 272-274, September 2005. [Article \(CrossRef Link\)](#).
- [33] G. Song and K. Zhang, "Visual XML schemas based on reserved graph grammars," in *Proc. of the International Conference on Information Technology: Coding and Computing*, vol.1, pp. 687-691, April 2004. [Article \(CrossRef Link\)](#).
- [34] K. Zhang, D. Zhang and J. Cao, "Design, construction, and application of a generic visual language generation environment," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 289-307, April 2001. [Article \(CrossRef Link\)](#)
- [35] X. Zeng, X. Han and Y. Zou, "An edge-based context-sensitive graph grammar formalism," *Journal of Software*, vol. 19, no. 8, pp.1893-1901, 2008. [Article \(CrossRef Link\)](#).
- [36] Z. Shi, X. Zeng, T. Zhang, S. Huang, Z. Qi, H. Li, B. Hu, Y. Yao and S. Zhong, "Bidirectional transformation between BPMN and BPEL with graph grammar," *Computers & Electrical Engineering*, vol. 51, pp. 304-319, April 2016. [Article \(CrossRef Link\)](#)

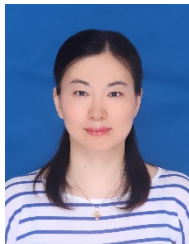
- [37] Z. Shi, X. Zeng, Z. Yang, S. Huang, H. Li, B. Hu and Y. Yao, "A temporal graph grammar formalism," *Journal of Visual Languages and Computing*, vol. 47, pp.62-76, August 2018.
[Article \(CrossRef Link\)](#)
- [38] D. Kundu and D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams," *Journal of Object Technology*, vol. 8, no.3, pp.65-83, May 2009.
[Article \(CrossRef Link\)](#)
- [39] K. Pechtanun and S. Kansomkeat, "Generation test case from UML activity diagram based on AC grammar," in *Proc. of the International Conference on Computer & Information Science (ICIS)*, Kuala Lumpur, Malaysia, pp. 895-899, June 2012. [Article \(CrossRef Link\)](#)
- [40] P. E. Patel and N. N. Patil, "Testcases Formation Using UML Activity Diagram," in *Proc. of the International Conference on Communication Systems and Network Technologies*, Gwalior, pp. 884-889, April 2013. [Article \(CrossRef Link\)](#)
- [41] A. K. Jena, S. K. Swain and D. P. Mohapatra, "A novel approach for test case generation from UML activity diagram," in *Proc. of the International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Ghaziabad, India, pp. 621-629, February 2014.
[Article \(CrossRef Link\)](#)
- [42] P. Mahali, S. Arabinda, A. A. Acharya and D. P. Mohapatra, "Test case generation for concurrent systems using UML activity diagram," in *Proc. of IEEE Region 10 Conference (TENCON)*, Singapore, pp. 428-435, November 2016. [Article \(CrossRef Link\)](#).



Zhan Shi received the Ph.D. and M.S. degree from Hohai University, and the B.S. degree from ZhengZhou University, China. He currently is a lecturer and teacher of the Institute of Computer Engineering, NanJing Institute of Technology. His research interests include artificial intelligence, machine learning, graph grammars, and software test.



Xiaoqin Zeng received the Ph.D. degree from the Hong Kong Polytechnic University, the M.S. degree from Southeast University, and the B.S. degree from Nanjing University, all in Computer Science. He currently is a professor and Ph.D. student supervisor at Hohai University, China. Prof. Zeng, as a principal investigator, has taken charge of several research projects awarded by Natural Science Foundation of China. His current research interests include Computational Intelligence, machine learning, pattern recognition, and graph grammars.



Tingting Zhang is an associate professor of computer and military software engineering at the Army Engineering University, Nanjing, China, where she received B.S. and M.S. degree in computer science and information Technology in 2003 and 2007 respectively, and the Ph.D. degree in Communication, in 2016. She is currently a post Ph.D. student at Southeast University, and the 28th Research Institute of China Electronics Technology Group Corporation. Her major field of evolutionary computing, intelligent computing, swarm unmanned system, software engineering, system and systems engineering, and system engineering.



Lei Han received the B.S. and M.S. degrees computer science and technology from China University of Mining and Technology, China, in 2004 and 2007, respectively, and Ph.D. degree in computerscience and technology from Hohai University, China, in 2018. From March to September 2019, he has served as Visiting Scholar at University of Oulu, Finland. He is now an associate professor at Nanjing Institute of Technology, China since 2016. He was a lecture and an assistant professor at Nanjing Institute of Technology from 2010 to 2016 and from 2007 to 2010, respectively. His research interest includes computer vision, artificial intelligence, computational imaging.



Ying Qian is a lecturer and teacher of Computer Engineering, Nanjing Institute of Technology (NJIT), where she teaches and conducts research in the field of Software Engineering and System Engineering. Her research interests include software engineering, object detection, artificial intelligence, image classification.