Automatic scenario-oriented test case generation from UML activity diagrams: a graph transformation and simulation approach

Abdelkamel Hettab*, Allaoua Chaoui and Mohamed Boubakir

MISC Laboratory,
Department of Computer Science and its Applications,
Faculty of NTIC,
University Constantine 2,
Constantine, Algeria
Email: kamelhettab@yahoo.fr
Email: chaoui2001@yahoo.com
Email: elhillalik@yahoo.fr
*Corresponding author

Elhillali Kerkouche

Department of Computer Science,
University of Jijel, Algeria
and
MISC Laboratory,
University Constantine 2,
Constantine, Algeria
Email: elhillalik@yahoo.fr

Abstract: Model-based testing (MBT) is an activity that allows designing and generating test cases from the initial specification of the system under test (SUT). Unified modelling language (UML) is a standard for model-based specifications, while UML-ADs (UML activity diagrams) are usually used for modelling the overall behaviour of systems. This paper presents a graph transformation-based approach to generate automatically scenario-oriented test cases from UML-ADs. To facilitate the test scenario generation process, an intermediate model called extended activity dependency graph (EADG) is proposed. The approach consists of generating EADG models from UML-ADs. Then, test scenarios are generated from the obtained EADG models. This approach also allows testers to validate their proposed test scenarios by applying them on UML-ADs using a graphical simulation. All ideas presented above are implemented using the graph transformation tool AToM^3. To this end, two meta-models and three graph grammars are proposed for presenting and generating EADG and test scenarios models, and for performing the graphical simulation. The approach is applied on a case study and experimental results show that our approach has a high rate of fault-detection capability. This approach can detect more defects in complex structures of concurrency and nested loops.

Keywords: automatic test case generation; graph transformation; UML activity diagram; model-based testing; test coverage criteria; AToM^3.

Reference to this paper should be made as follows: Hettab, A., Chaoui, A., Boubakir, M. and Kerkouche, E. (2022) ‘Automatic scenario-oriented test case generation from UML activity diagrams: a graph transformation and simulation approach’, *Int. J. Computer Aided Engineering and Technology*, Vol. 16, No. 3, pp.379–415.

Biographical notes: Abdelkamel Hettab is a PhD student at the Department of Computer Science and its Applications, Faculty of New Technologies of Information and Communication, University Abdelhamid Mehri Constantine 2, Algeria. He received his Master degree in Computer Science, Software Engineering Option in 2009 from the University of Constantine 2. He is a member in the Laboratory of Complex Systems Modeling and Implementation (MISC Modsation et Implntation des Systs Complexes), team Formal Methods and Software Engineering (FMGL). He has a study scholarship at the LIG laboratory (Grenoble France) between October 2015 and September 2016. His research fields are testing, formal methods, UML, software engineering, model driven engineering (MDE) and graph transformation systems.

Allaoua Chaoui is with the Department of Computer Science and its Applications, Faculty of NTIC, University Constantine 2, Algeria. He received his Master’s degree in Computer Science in 1992 (in cooperation with the University of Glasgow, Scotland) and PhD in 1998 from the University of Constantine (in cooperation with the CEDRIC Laboratory of CNAM in Paris, France). He has served as an Associate Professor at Philadelphia University in Jordan for five years and University Mentouri Constantine for many years. His research interests include software engineering, formal methods, model driven engineering and graph transformation.

Mohamed Boubakir is a PhD student in the Department of Computer Science and its Applications, Faculty of New Technologies of Information and Communication, University Abdelhamid Mehri Constantine 2, Algeria. He received his Master’s degree in Computer Science, Networks and Distributed Systems Option in 2007 from the University of Bejaia. He is a member of the Laboratory of Complex Systems Modeling and Implementation (MISC Modsation et Implntation des Systs Complexes), team Formal Verification of Software Product Lines. His research fields are software engineering, model driven engineering (MDE) and software product lines (SPL).

Elhillali Kerkouche is an Associate Professor at the Department of Computer Science, Faculty of Science and Technologies, Mohamed Seddik Ben Yahia University, Jijel, Algeria. He received his PhD from the University of Constantine (in cooperation with the LE2I Laboratory, University of Bourgogne, Dijon, France) in 2011. He has published many articles in international journals and conferences and he supervises many master students. He is a member of the Laboratory of Complex Systems Modeling and Implementation (MISC Modsation et Implntation des Systs Complexes), team Formal Methods and Software Engineering. His research fields are formal methods, UML, distributed systems, embedded systems, model driven engineering and graph transformation.

This paper is a revised and expanded version of a paper entitled ‘A graph transformation approach for automatic test cases generation from UML activity diagrams’ presented at C3S2E ‘15 the Eighth International Conference on Computer Science & Software Engineering, Yokohama, Japan, 13–15 July 2015.

1 Introduction

The software testing (Everett and McLeod, 2007) is an important activity for verifying the correctness of a systems implementation and several test case generation tools are developed (https://www.stickyminds.com/). Tests are performed and observations made. The correctness criteria to be tested must be given in the specifications.

Model-based testing (MBT) (Binder, 1979; Utting and Legeard, 2007) is a test strategy that depends on the extraction of test cases from different models. It has many advantages compared with traditional software testing strategies (e.g., manual test case generation). Firstly, the generation of test cases may be provided early in the life cycle of software development. Secondly, the software testers may understand better the system and find all test information by a simple comparison between models and code. Thirdly, by using different test coverage criteria, various test suites can be generated from the same model and the quality of them can be also improved. Furthermore, in the testing process, the test case generation is still the most error-prone, tedious and labour-intensive task, so its automation avoids human errors and reduces significantly the development effort.

Unified modelling language (UML) is a standard of modelling language (Unified Modeling Language Specification version 1.5; http://www.omg.org/spec/UML/2.5/Beta1/) and its diagrams are widely used for modelling the specification and the conception of the future system. UML diagrams model both static and dynamic aspects and represent different points of view of systems. UML activity diagram (UML-AD) is one of the important diagrams and it models the overall behaviour of systems. It is used to represent the control and the data flow of actions and activities in the application. It can be specified sequential treatments and offer a very similar vision of the programming languages (Boghdady et al., 2011). To this end, the description of a use case by an UML-AD corresponds to its algorithmic translation. UML-AD diagrams the global control flow between activities and its relationships between different objects. Furthermore, it can describe, step by step, an operation in the system. Thus, UML-AD is suitable for generating test cases and several works (Arora et al., 2017; Boghdady et al., 2011; Chen et al., 2006, 2008, 2009; Chouhan et al., 2012, 2013; Kundu and Samanta, 2009; Li et al., 2013; Linzhang et al., 2004; Mahali and Acharya, 2013; Malhotra and Bharadwaj, 2012; Ray et al., 2009; Sun, 2008; Sun et al., 2016; Swain et al., 2010; Swain et al., 2010) have used it.

This paper proposes an automatic approach for generating scenario-based test cases from UML-ADs. Furthermore, in order to improve the test quality this approach also gives testers the possibility of validating their proposed test scenarios by applying them on UML-AD using a graphical simulation. In this approach, an intermediate model called extended activity dependency graph (EADG) is proposed in order to facilitate the test scenario generation process. EADG is an extension of activity graphs presented in Boghdady et al. (2011) and Kundu and Samanta (2009). It is considered as an intermediate step in the test scenario generation process and captures all information needed to generate test scenarios. The approach consists of generating EADG model from UML-AD first. Then, test scenarios are generated from the obtained EADG according to an extended version of the basic path coverage criterion presented in Chen et al. (2008) and Linzhang et al. (2004) for non-concurrent activities and the simple path coverage criterion presented in Chen et al. (2006) for concurrent activities. Furthermore, in order to increase the effectiveness of the generated test scenarios, test scenarios proposed by testers can be validated by applying them on the original UML-AD using a graphical simulation. All ideas presented above are implemented using the graph transformation tool AToM^3 (A Tool for Multi-formalism and Meta-Modelling) (http://atom3.cs.mcgill.ca/). To this end, two meta-models and three graph grammars are proposed. The first meta-model is used for modelling UML-AD whereas the second one is used for modelling EADG and test scenarios model. Consequently, modelling tool is generated for each meta-model. The graph grammars are used for generating EADG and test scenarios, and for making the graphical simulation. After that, test data can be generated manually from the test scenarios using the category partition method (CPM) (Ostrand and Blacer, 1988) and then executed on the source code. Finally, the fault-detection capability of our approach is evaluated using the mutation analysis techniques (Jatana et al., 2017).

The rest of the paper is organised as follows. Section 2 introduces the context and the background of this work. Section 3 explains our contribution and its implementation. Section 4 describes a case study. Section 5 represents some similar works while Section 6 concludes the paper and gives some perspectives.

2 Background and context

2.1 UML activity diagram

UML-AD is one of the behavioural UML diagrams. It can be used for modelling the global behaviour of a system and also for describing an operation step by step. UML-ADs are used to model both control and data flows of activities. The control flow represents a sequence of actions with branches and loops. However, the data flow models the movement of data from an action to another. An activity is the implementation of a part of: a use case, a workflow or an algorithm. It has one or several activity nodes. There are three types of activity nodes namely: action nodes (executable nodes), control nodes and object nodes. Action nodes represent a production step in the overall behaviour of the activity. They can consume and produce data through a special kind of object nodes called pins (http://www.omg.org/spec/UML/2.5/Beta1/). Control nodes are usually used to manage the control flow in UML-AD including decision, merge, fork, join, initial and final nodes. Object nodes are used to model data flow in UML-AD including object nodes, input pins and output pins. Edges are used to model the sequence of activities including control flow, data flow and signal flow. Figure 1 shows an illustrate example of UML-AD. In this paper, UML-AD activities correspond to the execution of particular action or control nodes, and each swimming lane corresponds to one object.

2.2 Simple and basic path coverage criteria

A path in an UML-AD is an instance of its behaviour. It starts from the initial state to reach the final state passing by a set of nodes. All paths should be generated for testing an UML-AD, but the existence of loops and concurrency can result in a path explosion (Chen et al., 2006). To avoid such problem, the two concepts of basic path and simple path coverage criteria are introduced. The basic path is used to avoid the problem of the path explosion caused by loops and the simple path is used to avoid the problem of the path explosion caused by concurrency (Chen et al., 2006). A basic path is a path through UML-AD where each activity occurs either zero or one time. This definition ensures that iterations are exercised at most once. However, the problem of path explosion has not been completely avoided by using the concept of basic path because of the existence of competition. Thus, the concept of simple path is introduced to avoid this problem. The simple path is a basic path selected randomly from several basic paths which have the same activities and the same partial order relation is defined between them. Notice that, The partial order relation is reflexive, anti-symmetric and transitive relation between activities. It is denoted by the symbol $\prec$ (e.g., $a-i \prec b-j$ means that the activities $a-i$ are happen before the activities $b-j$) [see Figure 1(b)]. The definition of the simple path ensures the execution of all activities in UML-AD and avoids the repetition of several activities for several times caused by the concurrency. For example there are 33,600 basic paths (Chen et al., 2006) for testing the UML-AD shown in Figure 1(a), such as:

$$start \to a \to b \to c \to d1 \to d2 \to d \to j \to f \to k \to h \to i \to l \to end,$$

$$start \to a \to b \to c \to d1 \to d2 \to d \to j \to f \to k \to i \to h \to l \to end,$$

$$start \to c \to b \to a \to d1 \to d2 \to d \to j \to f \to k \to h \to i \to l \to end,$$

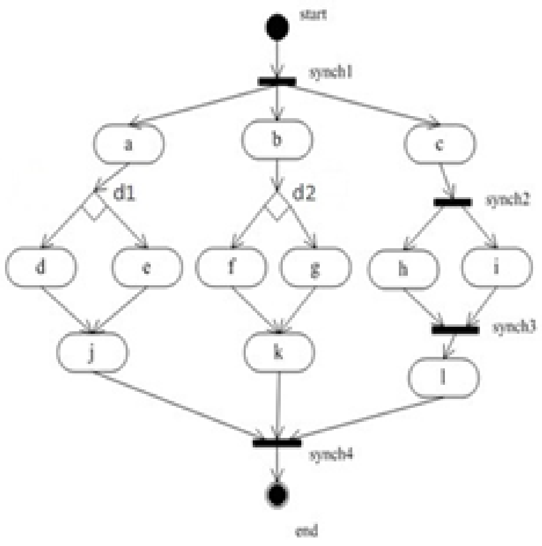
$$start \to a \to b \to c \to d1 \to d2 \to e \to j \to g \to k \to h \to i \to l \to end,$$

However, only two simple paths (Chen et al., 2006) are needed for testing it such as:

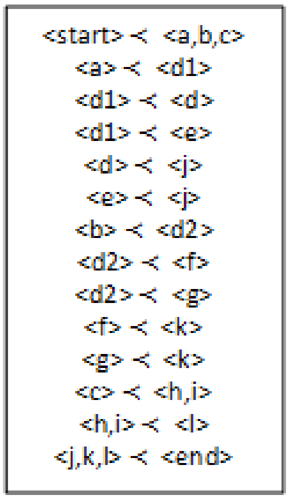
$$start \to a \to b \to c \to d1 \to d2 \to d \to f \to j \to k \to i \to h \to l \to end,$$

$$start \to a \to b \to c \to d1 \to d2 \to e \to g \to j \to k \to i \to h \to l \to end,$$

Figure 1 (a) A UML activity diagram (b) Its partial order relation



(a)



<start> ≺ <a,b,c>
<a> ≺ <d1>
<d1> ≺ <d>
<d1> ≺ <e>
<d> ≺ <j>
<e> ≺ <j>
<b> ≺ <d2>
<d2> ≺ <f>
<d2> ≺ <g>
<f> ≺ <k>
<g> ≺ <k>
<c> ≺ <h,i>
<h,i> ≺ <l>
<j,k,l> ≺ <end>

(b)

2.3 Model-based testing

Model-based testing (MBT) (Utting and Legeard, 2007; Binder, 1979) is an activity for designing and generating (automatically or not) test cases from an abstract model of the system under test (SUT). The model often offers a partial and discrete view of the expected behaviour of the system and captures its important aspects. The generated test cases from abstract models cannot be executed directly on the executable code because of the different abstraction between models and code. This distinction requires often manual intervention by a test engineer who makes a design adaptation to pass from a series of abstract tests to executable tests. This step is usually called concretisation. Finally, during the test cases execution, a comparison is made between the actual behaviour of the software (the executable code) and the expected behaviour which is described in models.

2.4 Graph transformation and AToM^3 tool

Graph transformation techniques are used to create a graph from another one using graph rewriting rules (Rosenberg, 1997). Each rule contains two parts namely left hand side (LHS) and right hand side (RHS). The application of a rule involves substituting LHS by RHS if a matching is found between the LHS and a part of the model under transformation. This process is repeated until no rule can be applied. Each rule contains an execution condition and an order number to indicate its priority. A rule is executed if the execution condition is true and all its higher priority rules have been executed.

AToM^3 (http://atom3.cs.mcgill.ca/), which is an acronym for a tool for multi-formalism meta-modelling, is a tool for multi-paradigm modelling. It is developed and written using the Python programming language (htpp://www.python.org). The two main concepts of AToM^3 are meta-modelling and model transformation. Meta-models are created using a modified entity-relationship model and model transformations are produced by graph rewriting rules. The advantage of AToM^3 is that formalisms and models are described as graphs. Furthermore, the model transformations themselves can be declaratively expressed as graph-grammar models. For each proposed meta-model, AToM^3 generates a modelling tool for drawing and modelling its elements.

In AToM^3, there is a relationship between the elements of the source and the target models which is established using traceability links marked by numbers. The matching is not only based on the graphic disposition of the elements of models, but also on the attributes values which can take different values fixed on the graph or any value '⟨ANY⟩'. If an item's label appears in LHS of a rule, and it does not occur in its RHS, then after applying the rule, the item will be removed. However, if an element has a label on RHS, and this label does not appear on LHS then it will be created. The last case is that where the same item is found in both sides of the rule, in such case, that item will be maintain by the rewriting system, and its attributes are manipulated either by a simply copying from LHS (Directive ⟨COPIED⟩), or they will be specified by using a Python code fragment (⟨SPECIFIED⟩).

3 Our approach

This section presents our approach based on Graph transformation for generating test scenarios from UML-ADs. The approach consists of two parts as shown in Figure 2. The first one consists of generating test scenarios from UML-ADs according to the simple path coverage criterion for concurrent UML-AD activities, and an extended basic path coverage criterion for non-concurrent activities.

The first part includes the following two steps:

- generating the intermediate model EADG from UML-AD.
- generating test scenarios from the EADG obtained in the previous step according to the simple path and the extended basic path coverage criteria (see Subsection 3.1.2.1).

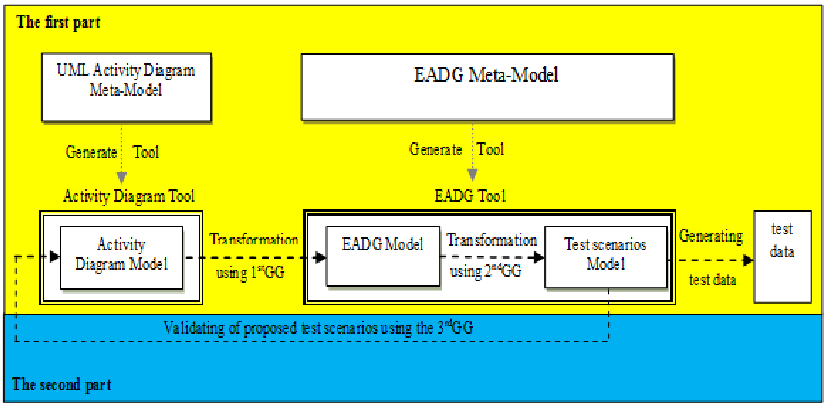
The second part of this approach consists of validating test scenarios proposed by testers using a graphical simulation. To this end, two properties are verified. The coverage property verifies that all test scenarios elements are presented in UML-AD, while the order property verifies that the order of activities is the same in the test scenarios model and UML-AD (see Section 3.2).

Notice that, test data can be generated from test scenarios generated by the first part or proposed by testers using the second part of our approach. These test data are generated manually using the category partition method (CPM) (Ostrand and Blacer, 1988).

3.1 The first part of our approach

The first part of our approach consists of generating test scenarios from UML-ADs according to the simple and the extended basic path coverage criteria. It includes two steps. The first one consists of generating EADG model from UML-AD, and the second step consists of generating test scenarios from the EADG obtained. The previous steps are implemented using the graph transformation tool AToM^3. Thus, two meta-models and two graph grammars are defined in this part. The two meta-models are used to specify UML-AD and EADG concepts. Consequently, a modelling tool is generated for each meta-model. Noting that the concepts of test scenarios are specified using the meta-model of EADG model. Two graph grammars are used for generating EADG and test scenarios; the first graph grammar is used for generating EADG from UML-AD, while the second graph grammar is used for generating test scenarios from the obtained EADG according to the simple path and the extended basic path coverage criteria. After that, test data are generated manually from the test scenarios using the CPM.

Figure 2 The architecture of our approach (see online version for colours)



3.1.1 *Transforming UML-ADs into EADG models*

This subsection presents the first step of the first part of our approach.

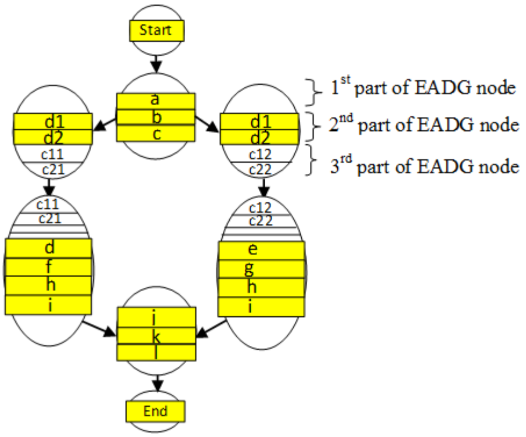
3.1.1.1 *Extended activity dependency graph*

The generation of test scenarios from UML-ADs is a difficult task especially with the existence of concurrent activities. So, in order to facilitate the test scenario generation process, an intermediate model called extended activity dependency graph (EADG) is proposed. EADG is an extension of the activity graphs presented in Boghdady et al. (2011) and Kundu and Samanta (2009). It captures all UML-AD features needed to generate test scenarios according to the simple path and the extended basic path coverage criteria (see Subsection 3.1.2.1). It is used to manage both control flow and data flow of activities. EADG model is composed of nodes interconnected by edges. Each node contains one or several activities. The EADG node which contains only one activity is non-concurrent, otherwise the EADG node which contains several activities is concurrent and its activities can be executed simultaneously. The elements of EADG nodes are modelled using dynamic lists.

Each EADG node consists of three parts: the first one is used for representing input aspects of UML-AD nodes such as incoming data, input pins and/or input guard conditions. The second part is used for representing UML-AD activities. The third part is used for showing output aspects of UML-AD nodes such as outgoing data, output Pins and/or output guard conditions (see Figure 3). There is a relationship between these three parts. If the second part of the EADG node contains one or several activities and each one occupies a position in its associated list, then the input and output aspects of each activity occupy the same positions in the lists associated to the first and the third part of the EADG node. Furthermore, if an UML-AD node has several input and/or output aspects then they are put in the same position of the list separated by the symbol '|'. For example in Figure 3 the condition *d1* and its associated guard condition *c11* occupy the first position in their lists. The condition *d2* and its associated guard condition *c21* occupy the second position in their lists. The EADG nodes are generated from UML-AD nodes by grouping together the nodes which have the same order in the partial order relation (see Figure 1) because they are in concurrency. Although, some or all nodes in the group can be in concurrency with other nodes which are not belonging to the group, only these nodes are grouped together. The reason is that in our approach, which uses the simple path coverage criterion, only some paths are selected from several basic paths and the others are not considered.

The EADG edge connects two EADG nodes (the source node and the target node). The activities of the target node cannot be executed until the execution of all activities of the source node. Figure 3 shows our proposed EADG for UML-AD shown in Figure 1(a). The activities *a*, *b* and *c* are grouped together in the same EADG node because they are in concurrency and they have the same order in the partial order relation. In a similar way, the two decision nodes *d1* and *d2* are grouped together in the same EADG node. The EADG node which has the activities *d1* and *d2* follows the EADG node which has the activities *a*, *b* and *c* because $a \prec d1$ (i.e., the activity a occurs before the activity *d1*) and $b \prec d2$, consequently, $(a, b, c) \prec (d1, d2)$. The activities h and i can be grouped respectively with the two groups of activities (*d*, *f*) and (*e*, *g*) for building the two EADG nodes having respectively the two groups of activities (*d*, *f*, *h*, *i*) and (*e*, *g*, *h*, *i*). Noting that, this example illustrates our way for grouping the UML-AD activities in the EADG model in order to generate some simple paths from several basic paths during the test scenarios generation process.

Figure 3 The EADG generated from the UML-AD of Figure 1 (see online version for colours)

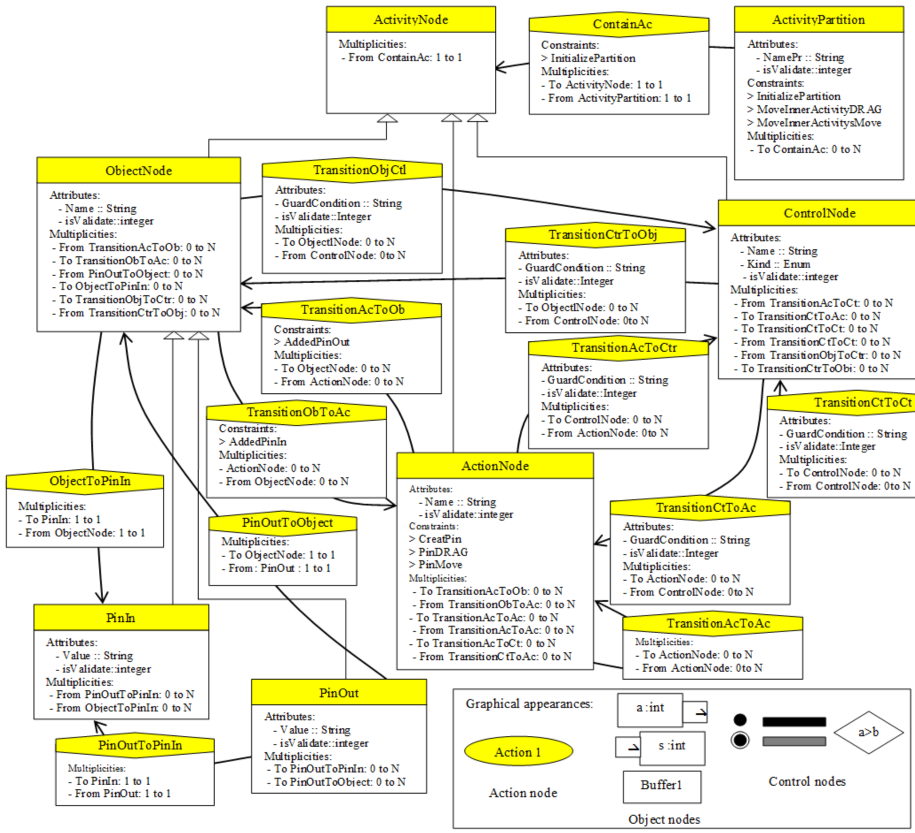


3.1.1.2 *Activity diagram meta-model*

Our proposed meta-model for UML-AD consists of seven classes and 12 associations (see Figure 4). The classes are: *ActivityPartition*, *ActivityNode*, *ActionNode*, *ControlNode*, *ObjectNode*, *PinIn* and *PinOut*. To clarify our meta-model we chose and explain the class *ActionNode* which represents an action node in the UML 2.5 notation. It has two attributes. The first one called *Name* is used to identify the name of the action node. The second attribute called *isValidate* is used to validate this action node with EADG elements during the process of test scenarios validation (see Subsection 3.2.4). The class *ActionNode* is related with itself by the association *TransitionAcToAc*. It is also related with the class *ControlNode* by the associations *TransitionAcToCt* and

TransitionCtToAc, and related with the class *ObjectNode* by the associations *TransitionAcToObj* and *TransitionObjToAc*.

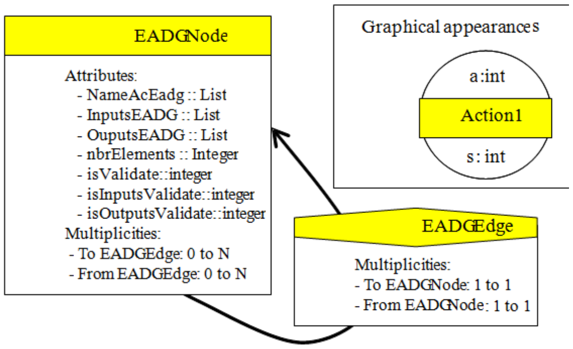
Figure 4 Meta-model of activity diagram and its graphical appearances (see online version for colours)



3.1.1.3 EADG meta-model

Our proposed meta-model for the EADG model consists of a class and an association (see Figure 5). The class contains the attributes: *nameAcEADG*, *InputsEADG* and *OutputEADG* which indicate the list of activities and their input and output aspects respectively. The attribute *nbrElement* of type integer is used to indicate the number of elements of the lists. The EADG node contains also the attributes *isValidate*, *isInputValidate* and *isOutputValidate* of type integer which are used to validate the generated test scenarios (see Subsection 3.2.4). The class EADGNode is related with itself by the association *EADGEdge*. The EADG node is represented graphically by a white circle with a yellow rectangle inside. Each EADG node is divided into three parts: The first and the third parts (the two parts of the white circle) are used to represent the set of input and output aspects respectively. The second part (the yellow rectangle) is used to represent UML-AD activities.

Figure 5 Meta-model of EADG and its graphical appearances (see online version for colours)



3.1.1.4 First graph grammar which transforms UML-AD into EADG

This subsection presents the first graph grammar that transforms any UML-AD into an EADG model. There is an initial Action associated to this graph grammar. It decorates all nodes and transitions of UML-AD with temporary attributes which are used in the conditions specified in the rules. In the following, some local temporary attributes are presented:

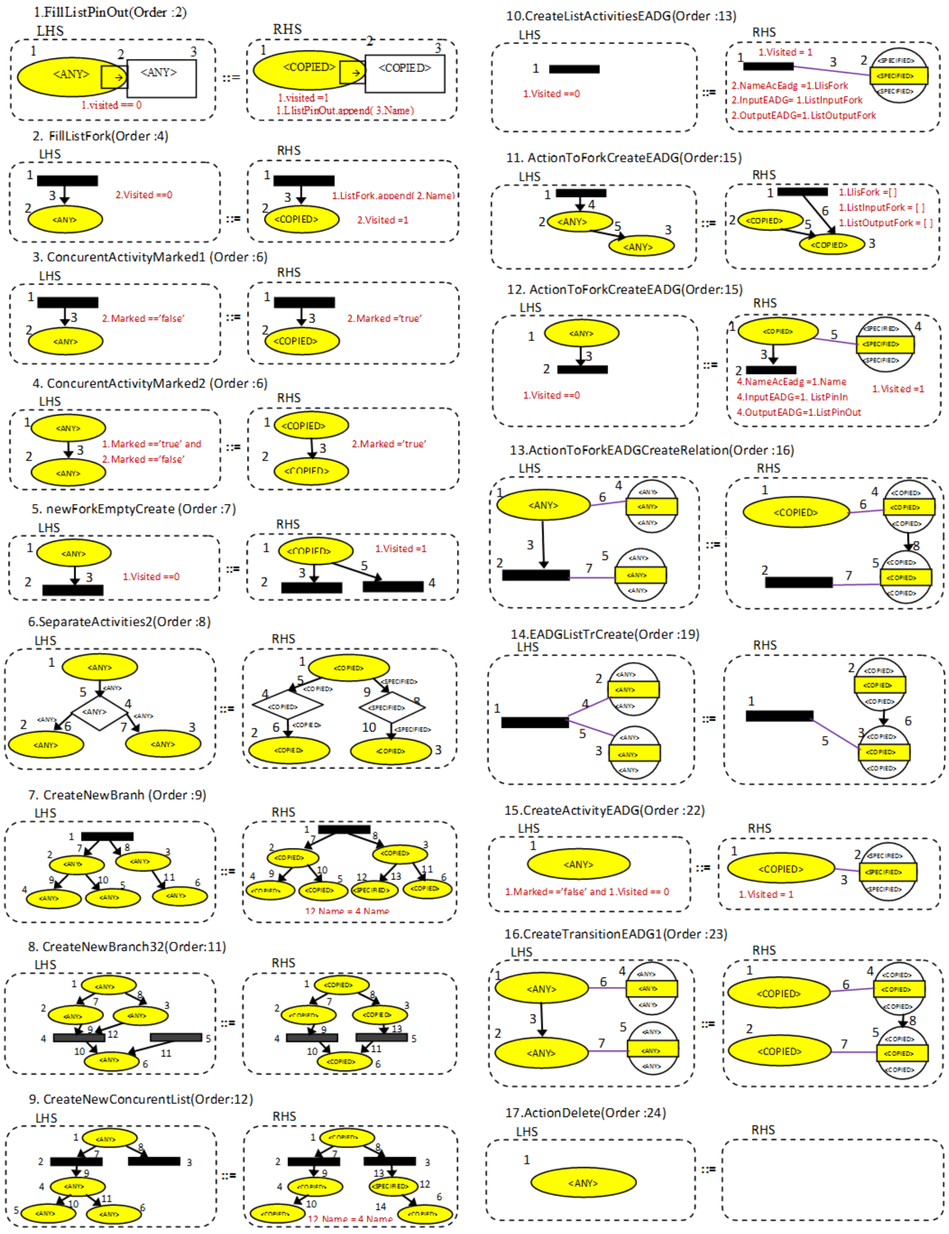
- *Visited:* it is used to count the number of times that an UML-AD node has been processed.
- *Concurrent:* it is used to indicate if a given ULM-AD node is concurrent or not.
- *nbrSucc:* it is used for counting the number of nodes which follow an UML-AD node.
- *ListPinIn*, *ListPinOut:* they are used for collecting information of inputs and outputs aspects for UML-AD Action nodes.
- *ListFork*, *listActInFork*, *listActOutFork:* they are associated to UML-AD fork nodes for constructing lists of UML-AD nodes, which follow the fork node, and their input and output aspects respectively.
- *ListActInDesMrg*, *listActOutDesMrg:* they are used for building lists of guard conditions that follow and precede UML-AD decision and merge nodes, respectively.

The temporary attributes with type integer (*Visited*, *Concurrent*, *nbrSucc*) are initialised to 0 and the temporary attributes with type list are initialised to [] (empty list).

The first graph grammar contains 50 rules divided into five categories. Some rules among them are shown in Figure 6.

The rules of the first category consist of collecting all information about input and output data, guard conditions and partitions names of UML-AD elements. Rule 1 is used for collecting all information about output pins for each action node. Rule 2 is used for collecting all information (e.g., the name of activities, input and output aspects) about UML-AD nodes which follow the fork nodes. Rules 3 and 4 are used for designating all concurrent UML-AD nodes. The value of the temporary attribute *concurrent* becomes 1 for all concurrent nodes.

Figure 6 Some rules for the first graph grammar that generates EADG models from the activity diagrams (see online version for colours)



1.FillListPinOut(Order :2)
1.visited == 0
1.visited =1
1.LlistPinOut.append(3.Name)
2. FillListFork(Order :4)
2.Visited ==0
1.ListFork.append(2.Name)
2.Visited =1
3. ConcurentActivityMarked1 (Order :6)
2.Marked =='false'
2.Marked ='true'
4. ConcurentActivityMarked2 (Order :6)
1.Marked =='true' and
2.Marked =='false'
2.Marked ='true'
5. newForkEmptyCreate (Order :7)
1.Visited ==0
1.Visited =1
6.SeparateActivities2(Order :8)
7. CreateNewBranh (Order :9)
12.Name = 4.Name
8. CreateNewBranch32(Order:11)
9. CreateNewConcurentList(Order:12)
12.Name = 4.Name
10.CreateListActivitiesEADG(Order :13)
1.Visited ==0
1.Visited = 1
2.NameAcEadg =1.LlisFork
2.InputEADG= 1.ListInputFork
2.OutputEADG=1.ListOutputFork
11. ActionToForkCreateEADG(Order:15)
1.LlisFork =[ ]
1.ListInputFork = [ ]
1.ListOutputFork = [ ]
12. ActionToForkCreateEADG(Order:15)
1.Visited ==0
4.NameAcEadg =1.Name
4.InputEADG=1. ListPinIn
4.OutputEADG=1.ListPinOut
1.Visited =1
13.ActionToForkEADGCreateRelation(Order :16)
14.EADGListTrCreate(Order :19)
15.CreateActivityEADG(Order :22)
1.Marked=='false' and 1.Visited == 0
1.Visited = 1
16.CreateTransitionEADG1(Order :23)
17.ActionDelete(Order :24)
LHS
RHS
::=
<ANY>
<COPIED>
<SPECIFIED>

The rules of the second category consist of preparing and facilitating the next steps of the EADG generation process by making some modification on UML-AD. Rule 6 is used to keep guard information about conditions of UML-AD decision and merge nodes during the EADG generation process. So, each decision node and merge node is decomposed into two identical nodes, each one will be followed by one node and one guard condition. Rules 5, 7, 8 and 9 are used for making some modifications on UML-AD to facilitate the selection of one or more simple paths from several basic paths. Notice that, with the presence of branches between the fork and the join nodes, the number of simple paths corresponds to the number of branches. Otherwise, one simple path can be chosen from several basic paths when there are no branches between the fork and the join nodes. Rule 5 is used to create new fork nodes if there are conditional or unconditional branches between a fork node and a join node. For each branch, a new fork node will be created and its local variables will be initialised. Rule 7 is executed if there are one or several UML-AD nodes which follow a fork node and having more than followed nodes. In this case, new followed nodes will be created from UML-AD nodes which follow the fork node and having only one followed node. Rule 8 is used for connecting each two UML-AD nodes belonging to two different branches with two different join nodes. These two nodes are not in concurrency and they will be grouped in two different EADG nodes during the EADG generation process. The same thing is done with join nodes (see Figure 6 rule 9).

The rules of the third category are used for generating EADG nodes and edges from concurrent UML-AD nodes. Noting that, links are created between the generated EADG and UML-AD nodes to use them during the EADG edges generation process. Rule 10 is used for creating concurrent EADG nodes from UML-AD fork nodes. All elements of the created EADG nodes are imported from the temporary attributes associated to the fork nodes. After execution of rule 10, all information about the nodes which follow the fork nodes are kept in the created EADG nodes. These nodes, which follow the fork nodes, form a level of nodes. Thus, rule 11 is applied to create new groups of concurrent UML-AD nodes that follow the fork nodes. The transitions that connect these fork nodes with the nodes of the current level of nodes are moved to the nodes of the next level. After each execution of rule 11, rule 2 is applied to fill in the lists *listFork*, *listPinInFork* and *listPinOutFork*. Then, rule 10 is applied for generating new concurrent EADG nodes. Rule 12 is used for creating non-concurrent EADG nodes from UML-AD nodes which precede the fork nodes. Rules 13 and 14 are used for creating EADG edges between concurrent EADG nodes.

The rules of the fourth category are used for generating EADG nodes and edges from non-concurrent UML-AD nodes. Rule 15 is used for generating EADG nodes from Action nodes of UML-AD, all information associated to the created EADG nodes are imported from the attributes and the temporary attributes associated to the action nodes. Rule 16 is used for creating EADG edges between non-concurrent EADG nodes.

The rules of the fifth category consist of deleting all elements of UML-AD (see Figure 6 rule 17).

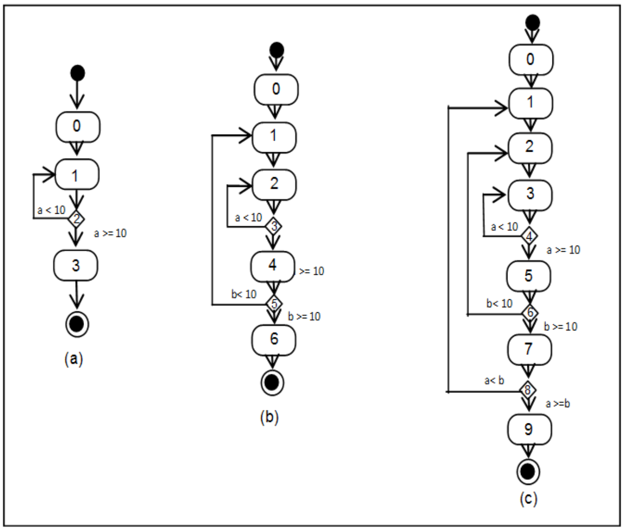
3.1.2 Generating test scenarios from the EADG model

This subsection presents the second step of the first part of our approach which consists of generating test scenarios from the EADG model.

3.1.2.1 The extended basic path coverage criterion

The basic path (Chen et al., 2006, 2008; Linzhang et al., 2004) of an UML-AD is a path that its activities occur only once. However, it does not able to test the loop structure. It does not test the true value of the loop condition. For example, for testing the loop structure shown in Figure 7(a) two paths are needed: *0 → 1 → 2 → 3* and *0 → 1 → 2 → 1 → 2 → 3*. But, the second path is not a basic path because the definition of the basic path is not respected (e.g., the activities 1 and 2 occur more than once). Thus, according to this example, the basic path coverage criterion may be not able to detect faults associated to the true value of the loop condition. So, for solving this problem, a new definition of the basic path is proposed. The extended basic path is a path that its activities occur at most once except for activities involved in loop structures that may occur twice. This definition takes into consideration nested loops. For example three paths can be generated from the nested loop shown in Figure 7(b). The path *0 → 1 → 2 → 3 → 4 → 5 → 6* covers the false value of the two nested loops conditions. The second path *0 → 1 → 2 → 3 → 2 → 3 → 4 → 5 → 6* covers the true value of the inner loop condition and the false value of the outer loop condition. The third path *0 → 1 → 2 → 3 → 4 → 5 → 1 → 2 → 3 → 4 → 5 → 6* covers the false value of the inner loop condition and the true value of the outer loop condition. For the nested loop shown in Figure 7(c), four paths can be generated. The first one *0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9* covers the false value of the three nested loops conditions. The second path *0 → 1 → 2 → 3 → 4 → 3 → 4 → 5 → 6 → 7 → 8 → 9* covers the false value of the intermediate loop and the outer loop conditions, and the true value of the inner loop condition. The third path *0 → 1 → 2 → 3 → 4 → 5 → 6 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9* covers the false value of the inner and the outer loops conditions, and the true value of the intermediate loop condition. The fourth path *0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9* covers the false value of the inner and the intermediate loops conditions, and the true value of the outer loop condition.

Figure 7 Examples of loop structures in the UML activity diagram



3.1.2.2 Simple path coverage criterion

The simple path coverage criterion is used for generating test scenarios from concurrent activities of UML-AD. It is a selected path from several basic paths (see Subsection 2.2). The EADG model offers an easy way for selecting a simple path by taking the elements of each EADG node according to their position in their lists and builds a series of sequential EADG nodes. For example the two following simple paths can be generated from the EADG shown in Figure 3:

$$start \to a \to b \to c \to d1 \to d2 \to d \to f \to h \to i \to j \to k \to l \to end,$$

$$start \to a \to b \to c \to d1 \to d2 \to e \to g \to h \to i \to j \to k \to l \to end.$$

In the first simple path, the activities *a*, *b* and *c* are taken according to their order in the list associated to the EADG node which contains these activities. The same thing is done with the activities (*d1*, *d2*), the activities (*d*, *f*, *h*, *i*) and the activities (*j*, *k*, *l*).

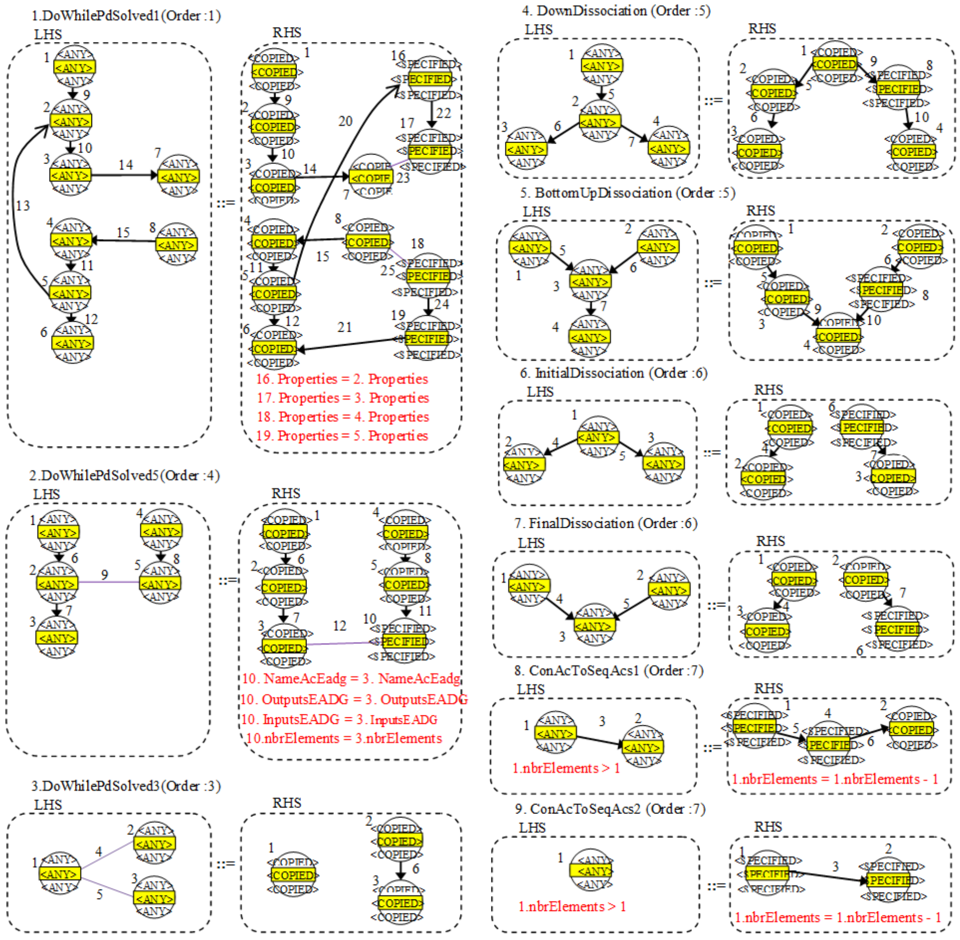
3.1.2.3 Second graph grammar that generates test scenarios from the EADG model

The second graph grammar, which generates test scenarios from EADG according to the simple and the extended path coverage criteria, consists of twelve rules divided into three categories (see Figure 8).

The rules of the first category consist of solving the problem of loop structures. Notice that, the number of generated paths from nested loops equals the number of the nested loops plus one (see Subsection 3.1.2.1). These generated paths cover the true value and/or the false value of each nested loop condition. Rule 1 is used to check the existence of loop structures and start building the path which covers the true value of the loop condition. Notice that, the loop structure in EADG is an EADG node having two branches. The execution flow of the branch, which corresponds to the true value of the loop condition, returns to the first EADG node after running a certain number of EADG nodes, while the execution flow of the other branch continues the execution to reach the final EADG node. Rule 2 is used to continue the construction of the path which covers the true value of the loop condition. It is repeated until a match is obtained with the LHS of rule 3. Rule 3 is used to finishes this process. The two links 4 and 5 , which link the EADG node 1 with the two EADG nodes 2 and 3, are obtained from the execution of rule 2 one or several times, and the rule 1 one time. In this case, these links are deleted, and EADG edge 6, which connects the two EADG nodes 2 and 3, is created. If there are nested loops, then the previous three rules are applied randomly on each loop.

The rules of the second category consist of cloning the EADG nodes for obtaining all test paths (test scenarios). The EADG nodes which compose these paths may be concurrent or not (i.e., they have several activities or only one). Rule 4 consists of cloning each EADG node having *n* (*n* > 1) followed EADG nodes to obtain *n* identical EADG nodes. In rule 5, the same technique of rule 4 is used for each EADG node having several preceded nodes. Rules 6 and 7 are used to clone the initial EADG node and the final EADG node.

Figure 8 Some rules for the second graph grammar that generates test scenarios from the EADG models (see online version for colours)


1.DoWhilePdSolved1(Order:1)
LHS
RHS
16. Properties = 2. Properties
17. Properties = 3. Properties
18. Properties = 4. Properties
19. Properties = 5. Properties
2.DoWhilePdSolved5(Order:4)
10. NameAcEadg = 3. NameAcEadg
10. OutputsEADG = 3. OutputsEADG
10. InputsEADG = 3. InputsEADG
10.nbrElements = 3.nbrElements
3.DoWhilePdSolved3(Order:3)
4. DownDissociation (Order:5)
5. BottomUpDissociation (Order:5)
6. InitialDissociation (Order:6)
7. FinalDissociation (Order:6)
8. ConAcToSeqAcs1 (Order:7)
1.nbrElements > 1
1.nbrElements = 1.nbrElements - 1
9. ConAcToSeqAcs2 (Order:7)
1.nbrElements > 1
1.nbrElements = 1.nbrElements - 1
::=
<ANY>
<COPIED>
<SPECIFIED>

The rules of the third category consist of splitting each concurrent EADG node into several non-concurrent EADG nodes for obtaining the final test scenarios set. To this end, *n* non-concurrent EADG nodes are generated from each concurrent EADG node which has n elements. Rule 8 is used to split a concurrent EADG node to two EADG nodes connecting by an EADG edge. The last element of the associated list of the first EADG node is used for constructing the second EADG node and deleted from the list after that. The same technique is used with rule 9. The execution of the previous two rules are repeated until all EADG nodes become non-concurrent (each EADG node has only one element). After execution of this graph grammar, the test scenarios are generated according to the extended basic and the simple path coverage criteria.

3.2 Validating of proposed test scenarios

The second part of our approach consists of validating any test scenarios set proposed by testers by applying them on the original UML-AD using a graphical simulation. The validation process consists of verifying two properties called coverage property and order property. The coverage property consists of verifying that all UML-AD activities and their input and output aspects are presented in the test scenarios model. The order property is used to verify that the execution order of UML-AD activities is respected in the test scenarios model.

In order to validate the test scenarios, four different colours are used to indicate if a part of UML-AD and/or the test scenarios model is valid or not. The red and the brown colours are used to designate invalid parts of the two models according the coverage and the order properties respectively. The green colour is used to designate the valid parts of models. The original colour of models is used to indicate model parts that have not yet been processed by the validation process.

3.2.1 Verifying the coverage property

The first step in the validation process consists of verifying the coverage property by comparing each EADG node in the test scenarios model with the original UML-AD activities. The test scenarios are validated one by one. Firstly, a test scenario is chosen and its initial EADG node is compared with the initial state of UML-AD. Secondly, the name of the EADG node which follows the initial EADG node is compared with the names of UML-AD nodes. If a matching is found, then these two nodes in the two models are valid according to the coverage property and their colour is changed into green. Secondly, all input and output aspects of these matched nodes are compared one by one. The colour of all valid aspects in the two nodes becomes green, otherwise the colour of invalid aspects becomes red. Finally, in the case where no matching is found between the chosen EADG node and all UML-AD nodes, this EADG node is invalid and its colour becomes red and so as the colour of its inputs and outputs aspects. This comparison process is repeated for each EADG node of each test scenario. After that, the colour of UML-AD nodes which has not changed during this validation process becomes red because they do not have any correspondents EADG node in the test scenarios model.

3.2.2 Verifying the order property

The second step in the validation process consists of verifying the order property. It is applied only on valid nodes of both models according to the coverage property (see the previous subsection). Notice that, invalid parts according to the coverage property are not worth validating by the order property because they are not valid according the complete validation process and they will be changed by testers. The order validation process consists of comparing the order of UML-AD nodes with the order of EADG nodes of the proposed test scenarios model.

For non-concurrent UML-AD nodes the order validation process is simple which consists of comparing the order of each two consecutive UML-AD nodes with their correspondent EADG nodes of the test scenarios model. If the order of nodes is the same, then the order property is verified and the colour of nodes remains green. Otherwise, the order property is not verified and the colour of nodes becomes brown.

For concurrent UML-AD nodes the validation process according to the order property is difficult because of complex structures of concurrency such as the existence of branches and loops between the fork and the join nodes.

The particular structures of concurrency are treated before normal structures. This validation process is applied on branches between the fork and join nodes. Notice that, concurrent UML-AD nodes which belong to different branches must correspond to EADG nodes of the test scenarios model belonging to different branches. In this case, all nodes of the two models are valid according to the order property, otherwise (e.g., two UML-AD nodes belonging to different branches and correspond to consecutive EADG nodes) these nodes are invalid according to the order property. Furthermore, for each branch between the fork and the join nodes, all UML-AD nodes which follow invalid nodes according to the order property are also invalid according to the order property, and so as their correspondent EADG nodes of the test scenarios model.

Then, the validation process is applied on normal structures of concurrency (i.e., no branches and loops between the fork and join nodes). Notice that, UML-AD nodes which follow the fork node must be corresponding to consecutive EADG nodes of the test scenarios model. In the opposite case, all these nodes in the two models are invalid according to the order property. Furthermore, the order of each two consecutive concurrent UML-AD nodes and the order of their correspondent consecutive EADG nodes must be the same. In the opposite case, all these nodes are invalid according to the order property. Moreover, each two concurrent UML-AD nodes which follow two valid concurrent UML-AD nodes according to the order property must correspond to two consecutive EADG nodes of the test scenarios model because in the test scenarios model concurrent EADG nodes form a series of consecutive nodes (see Subsection 3.1.2.3).

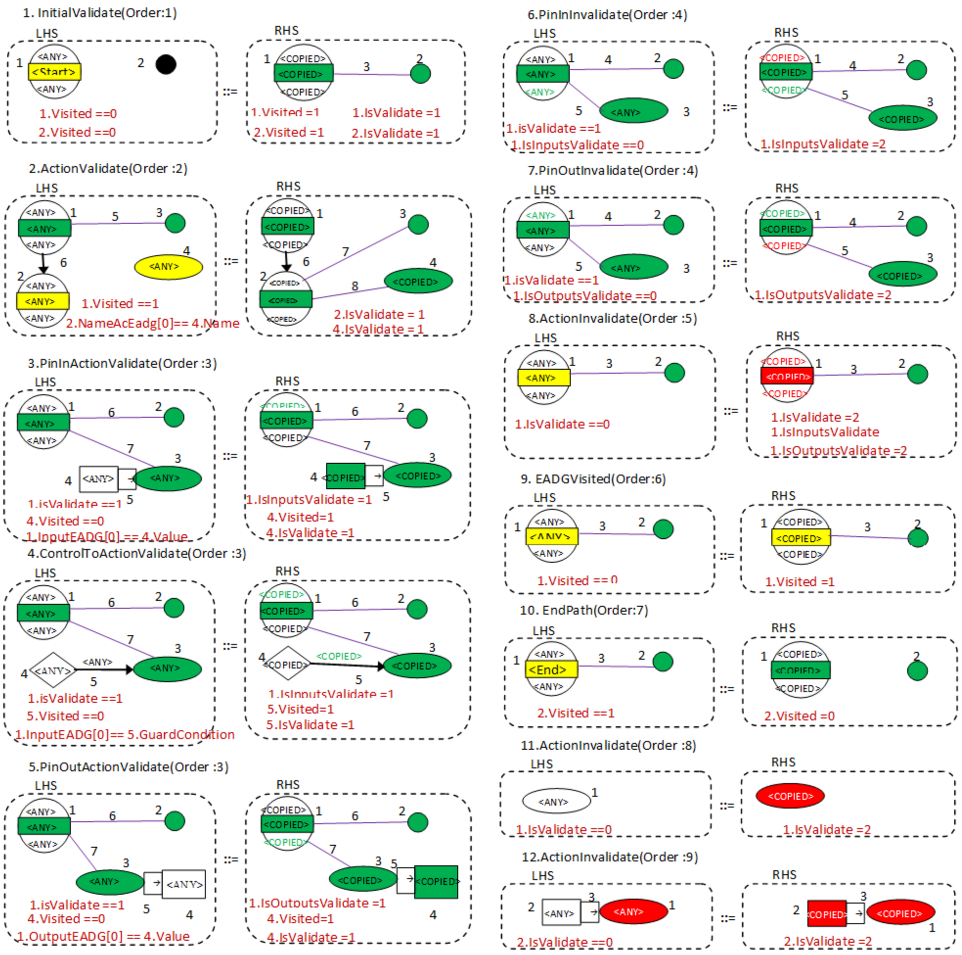
3.2.3 Evaluation of the validation process

The results of the validation process according to the coverage and the order property are evaluated. If the colour of all UML-AD nodes and all EADG nodes of the test scenarios model is green, then the test scenarios are valid according to the two properties. Otherwise, they are not valid and there are some invalid parts in the two models indicated by red and/or brown colours.

3.2.4 Third graph grammar which validate the test scenarios

This section presents the third graph grammar which is used to validate any test scenarios set proposed by testers according to the coverage and the order properties (see the previous subsections). There is an initial Action associated to this graph grammar to decorate all nodes and transitions and/or edges of UML-AD and test scenarios model with temporary attributes which are used in the conditions specified in the rules. There are three temporary attributes which are used to indicate if a part of UML-AD and the test scenarios model is valid or not. The first attribute with type integer is called *isValidate*. It is used for all types of nodes of UML-AD and the test scenarios model. The two other attributes (called *isInputsValidate* and *isOutputsValidate*) are used for the test scenarios model to indicate if the inputs and/or the outputs of the EADG nodes are valid or not. The previous attributes take values between 0 and 3. If the attribute value equals 0, then the validation process has not yet done on this node. If it equals 1, then the node is valid according to the coverage and the order properties. If it equals 2, then the node is not valid according to the coverage property, and if it equals 3, then the node is not valid according to the order property. Initially, these three attributes take the value 0 for each node of the two models.

Figure 9 Some rules of the first category of the third graph grammar (see online version for colours)

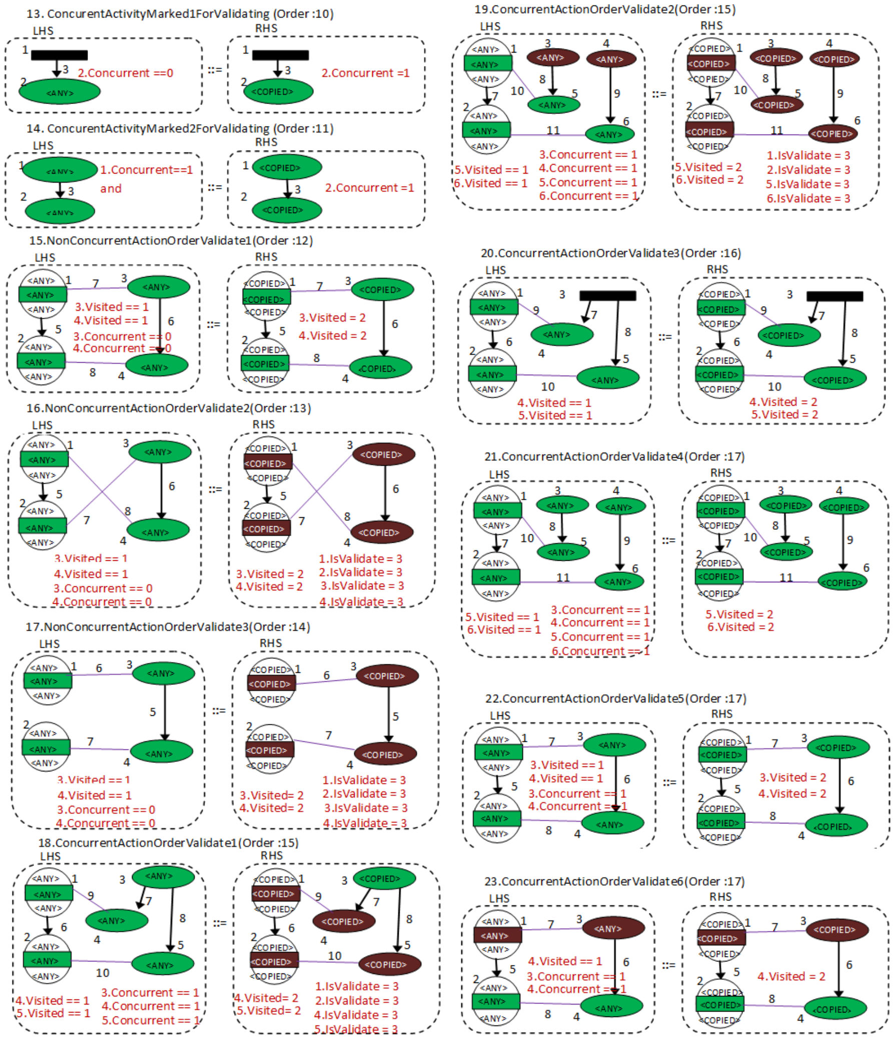


This graph grammar consists of 78 rules divided into two categories.

The rules of the first category are used for verifying the coverage property. They consist of 25 rules. Some rules among them are presented in Figure 9. Rule 1 is used to start the validation process according to the coverage property by comparing an initial EADG node of a test scenario with the UML-AD initial state. Rule 2 is used to verify the coverage property for the EADG node which follows the initial EADG node by comparing its activity name with the UML-AD names. If a matching is found with an UML-AD node, then the value of the temporary attribute *isValidate* becomes 1 for the two nodes of the two models and their colours become green. Rules 3, 4, 5, 6 and 7 are used to verify the coverage property for the input and the output aspects of nodes of the two models. Rules 3, 4 and 5 are used to detect valid parts in the two models. Otherwise, rules 6 and 7 are used to detect invalid parts. Rule 3 is used to verify if the input value of an EADG node of the test scenarios model equals the value of the input Pin of its correspondent UML-AD Action node. In the case where they are equal, the value of the temporary attribute *isInputValidate* of the EADG node becomes 1 so as the value of the temporary attribute *isValidate* of the input pin. Rule 4 is used to verify the input value of an EADG node with the incoming guard condition of its correspondent UML-AD action node. Rule 5 is used to verify the matching between the output value of an EADG node and the output Pin of its correspondent UML-AD action node. Rule 6 is used to detect invalid parts in the two models and executed only on a particular part of models if rules 3 and 4 have not been executed in this part. For example, if the condition *IsInputsValidate* == 0 is true for rule 6, then it is executed and the value of the property *IsInputsValidate* becomes 2; therefore, the colour of the input of the EADG node becomes red. However, if rules 3 and 4 have been executed, then the value of the propriety *IsInputsValidate* became 1 and rule 6 cannot be executed. Similarly, rule 7 is executed, only if rule 5 has not been executed, to detect invalid parts of the two models. Rule 8 is used to indicate invalid EADG nodes. It is executed on an EADG node only if rule 2 has not been executed and consequently rules 3, 4, 5, 6 and 7 have not been executed too. In such case, the value of its temporary attribute *IsValidate* remains 0, so this EADG node is invalid according to the coverage property because it does not have a correspondent UML-AD node. Rule 9 is used only to change the value of the temporary attribute *Visited* for an EADG node from 0 to 1 to move the validation process into the next EADG node in the current test scenario. After executing this rule the validation process can be repeated from rule 2. Rule 10 finishes the validation process according to the coverage property of a test scenario (a path). It changes the value of the temporary attribute *Visited* associated to the UML-AD initial state from 1 to 0 and deletes the link which connects it with the final EADG node of the current test scenario. After that, the previous validation process (rules 1 to 9) can be applied on another test scenario which has not been validated yet. Rules 11 and 12 are used to finish the validation process according to the coverage property of all test scenarios by changing the colour of all UML-AD nodes whose colour has not been changed during the validation process into red. These nodes are invalid because they do not have any correspondent EADG node of the test scenarios model.

The rules of the second category verify the order property. It consists of 53 rules. Figure 10 shows some rules among them. Rules 13 and 14 are used to indicate all concurrent UML-AD nodes. Rules 15, 16 and 17 are used to verify the order property for non-concurrent UML-AD nodes. Rule 15 shows that every two consecutive non-concurrent UML-AD nodes and their correspondent EADG nodes in the test scenarios model are valid according to the order property because the order of nodes is the same in the two models. Rule 16 shows that every two consecutive non-concurrent UML-AD nodes and their correspondent EADG nodes in the test scenarios model are invalid according to the order property because the order of nodes is not the same in the two models. Rule 17 shows that all non-concurrent UML-AD nodes which have not treated by rules 15 and 16 and their correspondent EADG nodes are invalid according to the order property because they are necessarily not in the same order in the two models. Rules 18 and 19 are used to verify the order property for concurrent UML-AD nodes and their correspondent EADG nodes in the test scenarios model. Rule 18 shows that every two concurrent UML-AD nodes which follow an UML-AD choice node (decision node or action node having two followed nodes) are invalid according to the order property if they correspond to two consecutive EADG nodes in the test scenarios model because these EADG nodes are not in concurrency. Rule 19 completes rule 18 and shows that all concurrent UML-AD nodes which follow invalid nodes and their correspondent EADG nodes of the test scenarios model are invalid according to the order property. Rules 20, 21, 22 and 23 are used to verify the order property for normal structures of concurrency in UML-AD. Rule 20 shows that each two UML-AD nodes which follow the fork node and correspond to two consecutive EADG nodes in the test scenarios model are valid according to the order property so as their correspondent EADG nodes. Rule 21 shows that each two correspondent nodes in the two models and they follow valid nodes are also valid according to the order property. Rule 22 shows that each two consecutive UML-AD nodes which correspond to two consecutive EADG nodes in the test scenarios model are valid according to the order property so as their correspondent EADG nodes. Rule 23 shows that two correspondent nodes in the two models which follow invalid nodes are valid according to the order property, although the source nodes in the two models are invalid. This is because the source nodes are invalid only with their predecessor nodes. This invalidity of the two source nodes comes from the execution of rule 18.

Figure 10 Some rules of the second category of the third graph grammar (see online version for colours)



13. ConcurrentActivityMarked1ForValidating (Order :10)
14. ConcurrentActivityMarked2ForValidating (Order :11)
15.NonConcurrentActionOrderValidate1(Order :12)
16.NonConcurrentActionOrderValidate2(Order :13)
17.NonConcurrentActionOrderValidate3(Order :14)
18.ConcurrentActionOrderValidate1(Order :15)
19.ConcurrentActionOrderValidate2(Order :15)
20.ConcurrentActionOrderValidate3(Order :16)
21.ConcurrentActionOrderValidate4(Order :17)
22.ConcurrentActionOrderValidate5(Order :17)
23.ConcurrentActionOrderValidate6(Order :17)
LHS
RHS
::=
<ANY>
<COPIED>
2.Concurrent ==0
2.Concurrent =1
1.Concurrent==1 and
3.Visited == 1
4.Visited == 1
3.Concurrent == 0
4.Concurrent == 0
3.Visited = 2
4.Visited = 2
1.IsValidate = 3
2.IsValidate = 3
3.IsValidate = 3
4.IsValidate = 3
5.Visited == 1
6.Visited == 1
3.Concurrent == 1
4.Concurrent == 1
5.Concurrent == 1
6.Concurrent == 1
5.Visited = 2
6.Visited = 2
5.IsValidate = 3
6.IsValidate = 3
4.Visited == 1
5.Visited == 1
4.Visited = 2
5.Visited = 2

3.3 Extracting of test data

The category partition method (CPM) (Ostrand and Blacer, 1988) is used for creating functional test data suites from the specifications of a system. In CPM, the input domain of function being tested is partitioned on classes, and then test data for each class of the partition is selected. Each partition corresponds to a *test frame* which is designated by a *category* which is a major property or characteristic of an environment or parameter. Test data are taken values within *categories* using *choices* (Chen et al., 2003). In the context of test scenarios, we use the same method presented in Sun (2008) to revise the previous definitions: each test scenario corresponds to a test frame, while test cases which respect a particular test scenario are generated according to the values of a set of choices which can be used to execute this test scenario. The choices values and their relationships are identified by processing guard conditions in the branches which occur in the same scenario paths. Finally, all input and output aspects of EADG nodes are used to generate the final test cases set for each test scenario using the boundary testing method by giving higher priority to boundary values to improve the ability of our approach towards the detection of defects. In this paper, test case selection from test scenarios is performed manually and automatic test case selection is left for our future work. Recall the seventh generated test scenario in Figure 14 (in Section 4), the guard condition of the branch activity *Valid login?* is *[Login failed]*. The guard condition of the branch activity *3ed invalid login?* is *[No]*. The guard condition of the branch activity *Valid login?* this time is *[Login Success]*. The output guard condition of the branch activity *D* is *[Faculty Tasks]*. All previous choices are dependent, for example the choice *[No]* holds according on the choice *[Login failed]*. By the same, the choice *[Login Success]* holds according on the choice *[No]*, etc. This test scenario must satisfy this dependency to be feasible; otherwise, it is infeasible. For example, *[Login failed]* and *[No]* and *[Login Success]* and *[Faculty Tasks]* is a feasible test frame, but, any change on any choice may lead to an infeasible test frame (e.g., the choice *[No]* is changed by the choice *[Yes]*). The other nine test frames are: the first one is *[Login failed]* and *[Yes]*. The second one is *[Login failed]* and *[No]* and *[Login failed]* and *[Yes]*. The third one is *[Login success]* and *[Faculty Tasks]*. The fourth and the fifth ones are *[Login success]* and *[Student Tasks]*. The sixth one is *[Login success]* and *[Admin Tasks]*. The eighth and the ninth ones are *[Login failed]* and *[No]* and *[Login success]* and *[Student Tasks]*. Finally, the tenth one is *[Login failed]* and *[No]* and *[Login success]* and *[Admin Tasks]*.

4 Case study

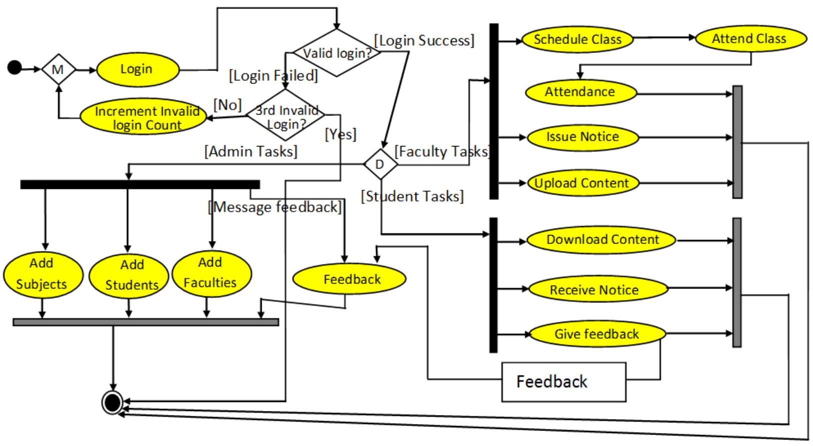
This section presents a case study to examine our proposed approach and evaluate its efficiency and effectiveness. First, in order to answering the question of the ability of our approach to generate test cases from the specification of a program under test, test scenarios are automatically generated from UML-AD of the case study. Then, in order to answering the question of the percentage of the fault-detection capability, the approach is evaluated using the mutation analysis tool MuJava (Ma et al., 2005).

This case study is carried out in four steps which are: generating of EADG model and test scenarios, generating of test data, seeding faults, and executing tests and collecting the results.

4.1 Generating of EADG model and test scenarios

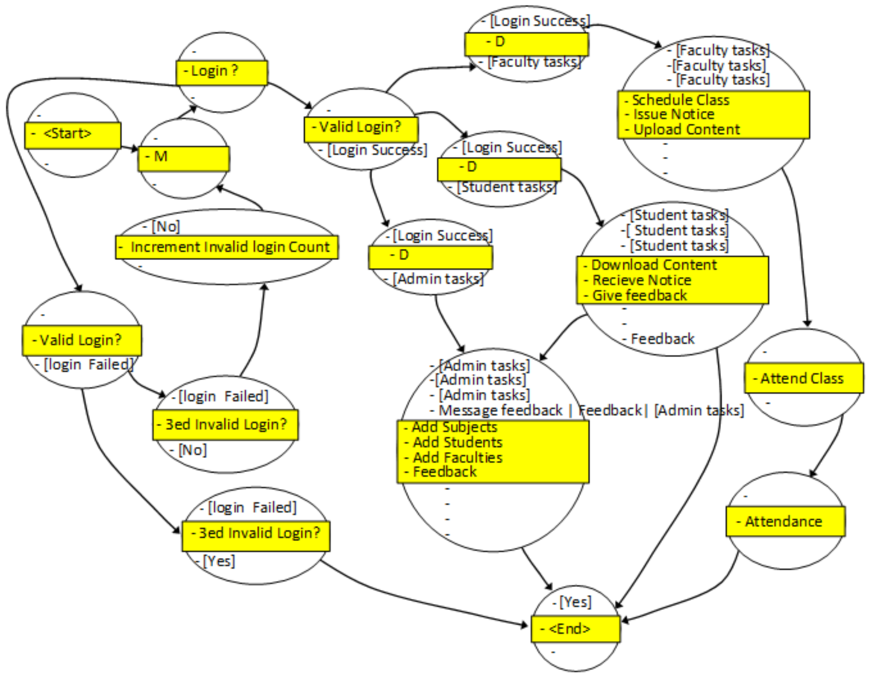
Our proposed approach can be applied in several different domains. In this paper, we have chosen the example of a college management system presented in Makhija et al. (2015). This scenario covers a college that offers online courses via virtual classrooms. This example contains the most important concepts of UML-AD like actions, loops, branches, bifurcations, forks and joins. There are many activities involved in the system. First, the user logs on. Then, the system checks whether the login entered is success or not. If it is failed, the system checks if the failed login is entered for the third time or not. If not, the system increments the count of failed login and returns to the activity *Login*. Otherwise, the system ends. In the case where the login is success, the system chooses a branch among: *Admin Tasks*, *Faculty Tasks* and *Student Tasks*. For each chosen branch the system simultaneously executes some concurrent activities and finishes itself (see Figure 11).

Figure 11 The college management system example (see online version for colours)



There are two steps for generating test scenarios from this example. The first step consists of generating the EADG model shown in Figure 12 from the UML-AD shown in Figure 11 by running the first graph grammar. Some remarks can be seen here: firstly, each set of concurrent activities in UML-AD, which are chosen according to the simple path coverage criterion, corresponds to one EADG node. For example, the concurrent activities: *Download Content*, *Receive Notice* and *Give feedback* are grouped together in the same EADG node (see Figure 12). This group of activities is obtained by applying rule 10 of the first graph grammar (see Figure 6 rule 10). Secondly, in order to keep the guard conditions of conditional branches, *n* EADG nodes are generated from each UML-AD decision node having *n* outgoing transitions ($n > 1$). Each generated EADG node is followed by one EADG node and it has one guard condition. For example, in the generated EADG model of Figure 12 there are three EADG nodes having the name *D*. The first one has the guard condition *[Faculty tasks]* in its third part, the second one has the guard condition *[Student tasks]* and the third one has the guard condition *[Admin tasks]*. The previous EADG nodes are obtained by executing rule 6 of the first graph grammar (see Figure 6 rule 6).

Figure 12 The EADG generated from the previous UML-AD (see online version for colours)



The second step in the test scenario generation process consists of generating the test scenarios model shown in Figure 13, 14 and 15 (i.e., each figure shows some test scenarios) from the EADG model shown in Figure 12 using the second graph grammar. In the final test scenarios set, each EADG node in each test scenario (test path) has one element. For example the EADG node having the activities *Schedule Class*, *Issue Notice* and *Upload Content* in the EADG model shown in Figure 12 is split into three consecutive EADG nodes named: *Schedule Class*, *Issue Notice* and *Upload Content* (see Figure 13 test scenario 3).

This approach also allows validating test scenarios proposed by testers by applying them (one by one) on the original UML-AD using a graphical simulation. We have proposed three test scenarios and put some faults in the second and the third ones in order to show the capability of our approach towards detecting the coverage and the order faults. Figure 16 shows the execution of the third graph grammar on the three test scenarios. The first one is valid according to the coverage and the order properties because all its elements are in green colour. The second one is invalid according to the coverage property because the EADG node which contains the activity *Increment valid login Count* is in red colour. This error is due to the fact that this activity does not have a corresponding UML-AD node. The third test scenario is invalid according to the order property because the colour of the two EADG nodes *Attendance* and *Attend Class* is brown. This fault is happened because the order of these two activities in the test scenarios model is not the same with the original UML-AD activities. Notice that, many UML-AD activities are not concerned here by the validation process and their colour is red because they do not correspond to any EADG activity. Furthermore, the previous faults can be corrected manually by testers and the validation process can be applied again for validating these corrections.

Figure 13 The test scenarios generated from the previous EADG (first part) (see online version for colours)

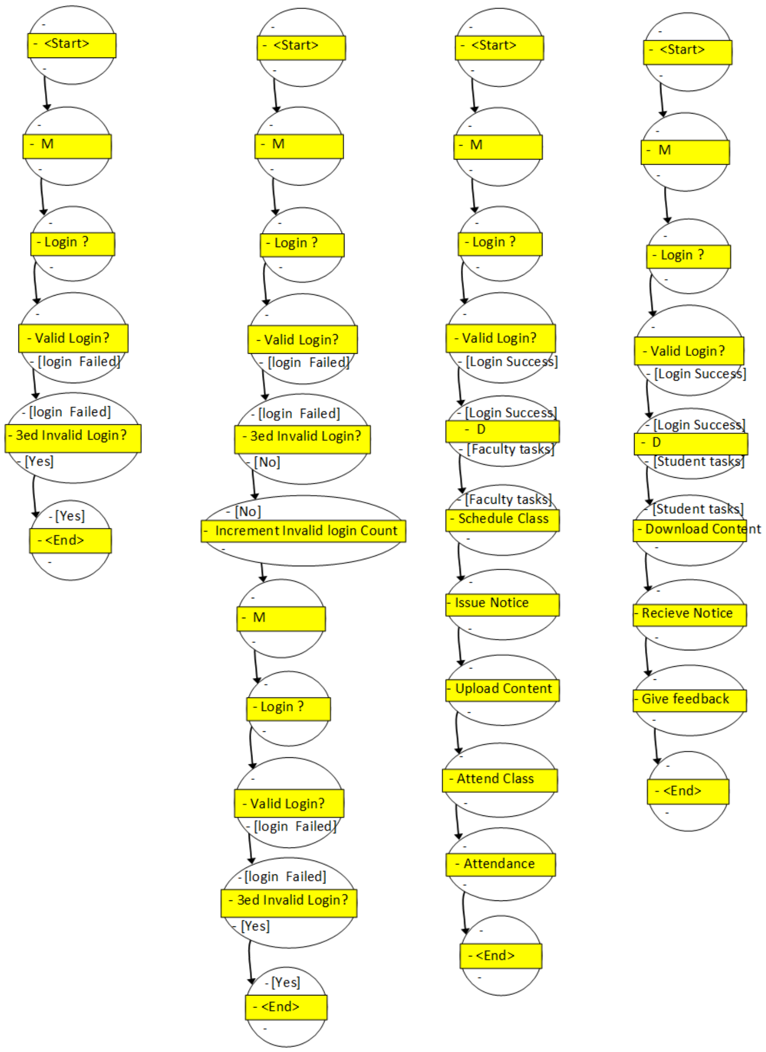


Figure 14 The test scenarios generated from the previous EADG (second part) (see online version for colours)

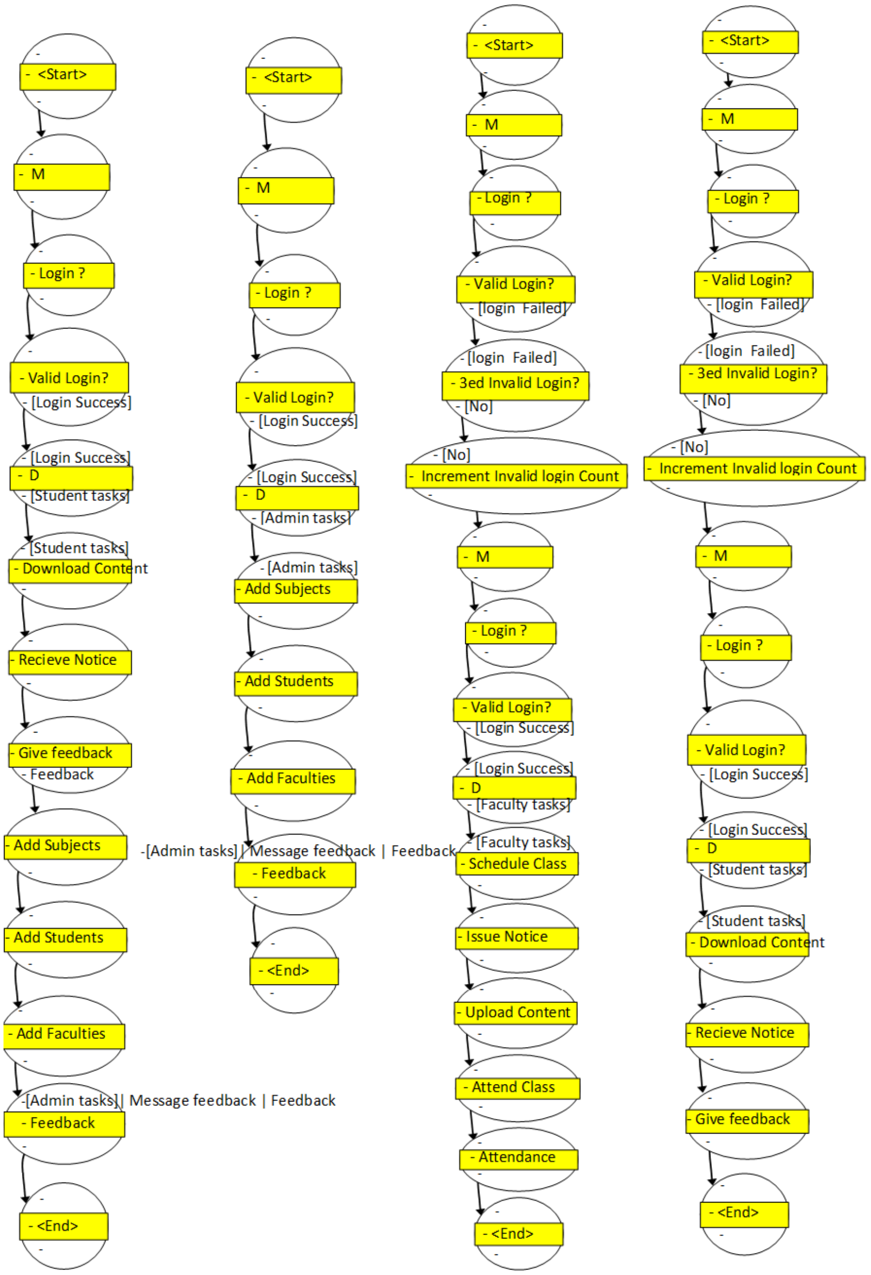


Figure 15 The test scenarios generated from the previous EADG (third part) (see online version for colours)

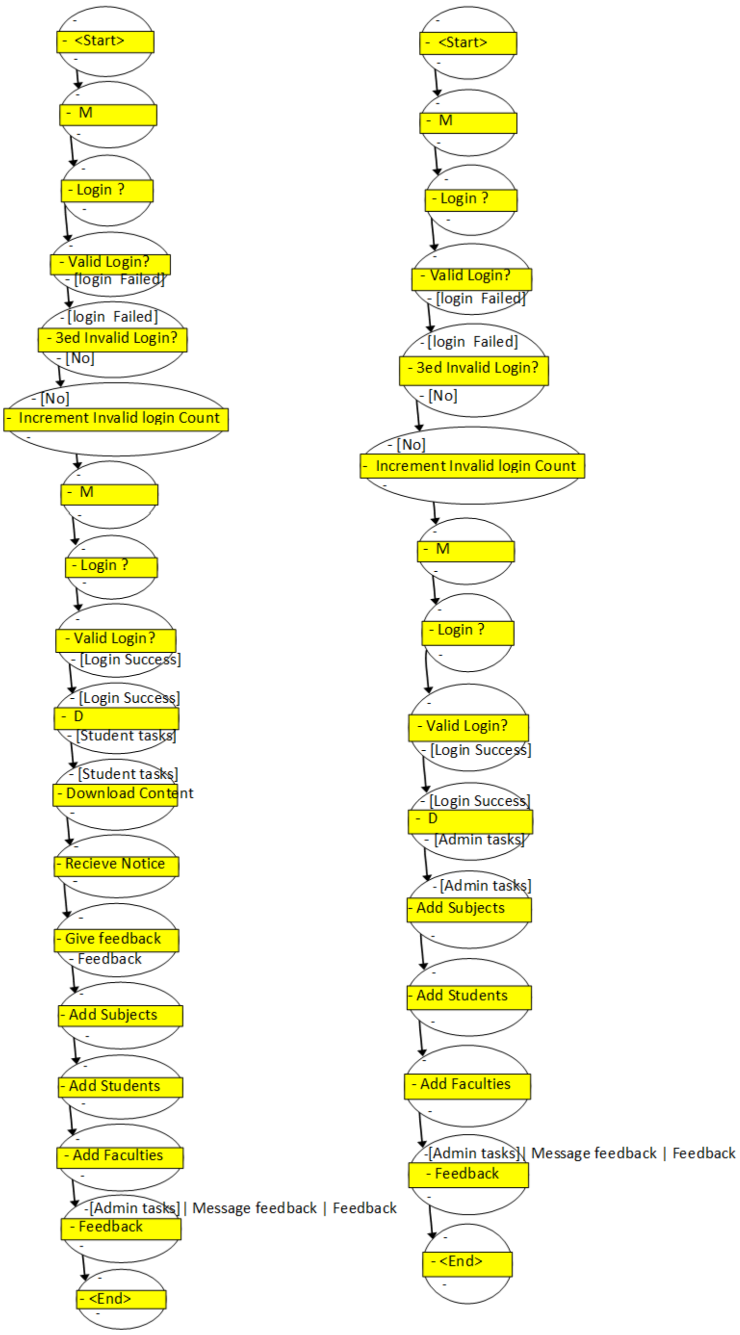
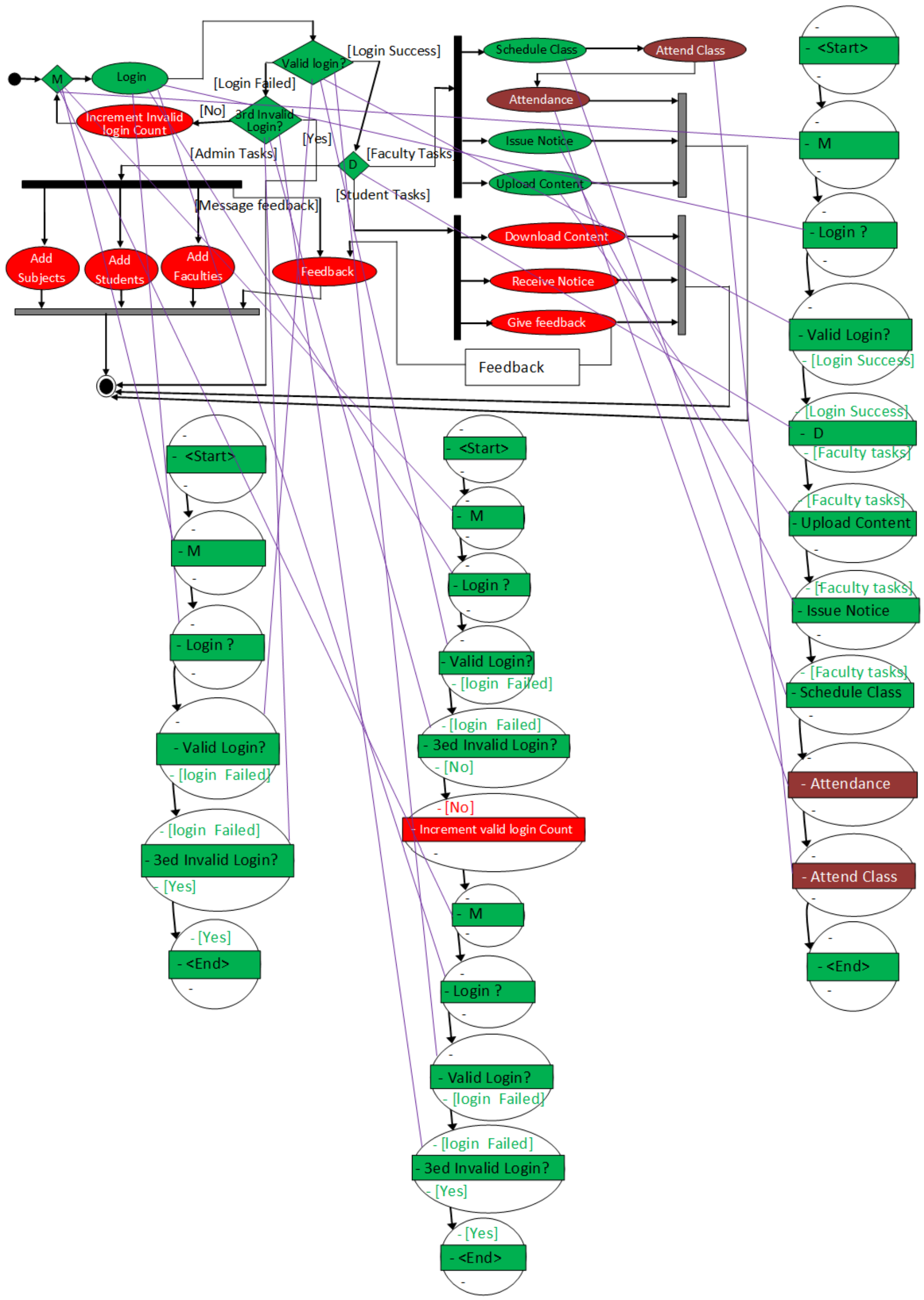


Figure 16 The process of test scenarios validation during its execution (see online version for colours)



4.2 Test data generation

A suite of test data which satisfies the generated test scenarios are generated using the category partition method (CPM). A number of test cases are generated for each test scenario using higher priority to boundary values to improve the fault-detection capability of our approach.

4.3 Seeding faults

Mutation testing consists of selecting and applying a set of mutation operators to each part of the source code of the program under test. The application of a mutation operator on a program results to one mutant (i.e., a program which is modified by a mutation operator). If these changes in the program are detected by at least one of the test suite, then the mutant is called: killed mutant. MuJava (Ma et al., 2005) is an open-source mutation system for Java programs. It is used for seeding faults into Java programs. There are 16 types of operators for the method-level mutation testing and 29 types of operators for the class-level mutation testing. In this work MuJava is used to automatically generate mutants for both method-level (traditional) mutation and class-level mutation testing of the college management system. Nine method-level operators and five class-level operators are applicable in this case study. At a results 229 method-level and 243 class-level mutants are generated.

4.4 Executing tests and collecting the results

The proposed approach is evaluated for indicating a test suite adequacy of the program under test by calculating the mutation score (MS). The mutation score is defined as the percentage of killed mutants (i.e., the mutants whose seeded faults are detected by the test suite) with the total number of mutants.(mutation score = (number of mutants killed/total number of mutants) $*$ 100). The fault-detection capability of the generated test suite using our approach is evaluated as follows: First, the impact of the test suite size on the effectiveness of the proposed approach is evaluated by changing the number of test cases per each test scenario. Next, the impact of the number of test scenarios is evaluated by adding some test scenarios which have been validated by the second part of our approach using the coverage and the order properties.

4.4.1 Fault-detection capability using test suite size

The number of test cases for each test scenario is changed from 1 to 20. First, the original program and the adequate generated MuJava mutants are executed using 1 test case for each test scenario. Then, they are executed using 5, 10 and 20 test cases for each test scenario. Tables 1 and 2 present the *MS* score for each operator and the overall *MS* score of the method-level and the class-level mutation testing respectively.

Table 1 The mutation score (MS) for the method-level mutants using our approach

Mutant type	*Number of mutants*	*Number of test cases per test scenario*	*Number of killed mutants*	*Mutation score (MS)*
AORS	6	1	4	67%
		5/10/20	6	100%
AOIU	14	1	7	50%
		5/10/20	11	79%
AOIS	40	1	26	65%
		5/10/20	29	73%
ROR	27	1	19	70%
		5/10/20	19	70%
COI	7	1	7	100%
		5/10/20	7	100%
LOI	19	1	14	74%
		5/10/20	14	74%
VDL	5	1	3	60%
		5/10/20	5	100%
CDL	27	1	27	100%
		5/10/20	27	100%
ODL	84	1	84	100%
		5/10/20	84	100%
Total	229	1	191	83%
		5/10/20	202	88%

From Tables 1 and 2, in general, we can notice that the fault-detection capability of the test suite is good for both method-level and class-level mutation testing with 88% for the method-level and the class-level mutation testing. The second remark is that the *MS* score of the class-level mutation and the method-level mutation is almost the same. The third remark is that, when the size of test suite is 5 and more, the test suite size has no impact on the fault-detection capability for the two mutation levels. The fourth remark is that, the test suite size has a big impact for class-level mutation when the size of the test suite is changed from 1 to 5 while it has a small impact with the method level. As results, our approach needs only a small number of test cases for each scenario.

In the following, more details about the results which shown in the two tables 1 and 2 are presented. Table 1 shows the following observations: The *MS* score is 100% for the operators: conditional operator insertion (COI), constant deletion (CDL) and operator deletion (ODL). That means that our approach can detect all these types of faults. The *MS* score is less than 100% but it still very high for the operators: arithmetic operator replacement replace shortcut arithmetic operators with other unary arithmetic operators (AORS) and variable deletion (VDL). That means that our approach can detect the majority of these types of faults. The *MS* score is not very high for the operators: logical operator insertion (LOI), arithmetic operator insertion insert basic unary arithmetic operators (AOIU), arithmetic operator insertion insert shortcut arithmetic operators

(AOIS) and relational operator replacement (ROR). That means that our approach cannot detect a lot of these types of faults.

Table 2 The mutation score (MS) for the class-level mutants using our approach

Mutant type	*Number of mutants*	*Number of test cases per test scenario*	*Number of killed mutants*	*Mutation score (MS)*
PRV	86	1	46	53%
		5/10/20	77	89%
JTI	24	1	20	83%
		5/10/20	21	87%
JTD	25	1	21	84%
		5/10/20	22	88%
JSI	26	1	11	42%
		5/10/20	11	42%
EAM	82	1	25	30%
		5/10/20	82	100%
Total	243	1	123	51%
		5/10/20	213	88%

Table 2 shows the following observations: the *MS* score is almost 100% for the operator accessor method change (EAM) that means that our approach can detect almost all these types of faults. The *MS* score is less than 100% but it still very high for the operators: this keyword deletion (JTD), this keyword insertion (JTI) and reference assignment with other comparable variable (PRV). That means that our approach can detect the majority of these types of faults. The *MS* score is low for the operator JSI (static modifier insertion) that means that our approach cannot detect a lot of these types of faults. In general, our approach detects about 88% of seeded faults according to the both levels of mutation testing where the size of test cases for each test scenario is 5 and more. This rate is considerable compared with random approaches because of to the best of our knowledge, studies show that the random test can quickly reach around 50% of the test goal but tends to peak afterwards (Arora et al., 2017; Thevenod-Fosse et al., 1991; Ntafos, 2001; Chen et al., 2010).

4.4.2 Fault-detection capability using more test scenarios

The fault-detection capability of the generated test suite using the simple and the basic path coverage criteria has been reported in the previous subsection. In this subsection we present the fault-detection capability of the generated test suite using more test scenarios which are proposed by testers and validated by our approach (see Section 3.2). When we use a sufficient number of added test scenarios (50 different test scenarios validated using the third graph grammar for the college management system example), the *MS* score where the size of test cases for each test scenario is 1 equals the *MS* score where the size is 5 and more. Furthermore, the previous *MS* score is the same where we use the simple and the basic path coverage criteria where the size of test cases for each test scenario is 5 and more. For example the previous *MS* score is 88% according to the both method-level and the class-level mutation. These equalities are happened for the following: test suites generated using the simple and the basic path coverage criteria and those using more test scenarios, which are validated by the third graph grammar, cover the same basic paths. Although, using more test scenarios increases the capability of detecting more concurrency-specific faults, the MS score is the same because of the absence of mutation operators specific for the concurrency in this version of MuJava. Thus, in this paper, we only interest by the fault-detection capability of the test suite generated using the simple and the basic path coverage criteria. In the future we aim to use other mutation tools which have specific concurrency operators for evaluating the concurrency fault-detection capability of the generated test suite.

5 Related work

Many studies (Arora et al., 2017; Boghdady et al., 2011; Chen et al., 2006, 2008; Chouhan et al., 2012, 2013; Kundu and Samanta, 2009; Li et al., 2013; Linzhang et al., 2004; Mahali and Acharya, 2013; Malhotra and Bharadwaj, 2012; Ray et al., 2009; Sun, 2008; Sun et al., 2016; Swain et al., 2010; Zhang et al., 2015) for the topic of test case generation from UML-ADs have been proposed. Chen et al. (2006, 2009) introduced the notion of simple path coverage criterion to generate test cases from UML-ADs using a modified version of depth first search (DFS) algorithm. First, they executed the program under test with random test cases to obtain several execution traces. Next, in order to obtain the reduced set of test cases according to the simple path coverage criterion, the obtained execution traces have been compared with the simple paths obtained by the execution of the improved DFS algorithm. In Boghdady et al. (2011), the authors proposed an approach for generating test cases from UML-ADs. First, a table called activity dependency table (ADT) has been generated automatically from UML-ADs and then transformed into a directed graph called activity dependency graph (ADG). Finally, the final test cases have been generated using ADG and ADT. Similar approaches to Boghdady et al. (2011) are proposed for generating test cases for mobile agents (Chouhan et al., 2012, 2013). In Chen et al. (2008), the authors proposed a method for generating test cases from UML-ADs using formal methods. First, a formal model called NuSMV (Cimatti et al., 2002) input has been generated from UML-AD. Then, coverage criteria have been selected and the properties in the form of LTL or CTL formulas have been generated. Next, the negated version of these properties has been applied on the obtained formal model using the model checking for generating the required tests (counterexamples). In Mahali and Acharya (2013) and Malhotra and Bharadwaj (2012), the authors used genetic algorithms for generating test cases from UML-ADs based on the prioritisation of test cases. In Zhang et al. (2015), the authors used the single population genetic algorithm and the data flow testing coverage criterion for generating test cases from UML-ADs. In Kundu and Samanta (2009), the authors used the activity path coverage criterion for generating test cases from UML-ADs. First, UML-AD has been augmented by the necessary test information. Then, UML-AD has been converted into an activity graph. Finally, test cases have been generated from the obtained activity graph. In Swain et al. (2010), the authors proposed an approach for generating test cases from UML state and activity diagrams for integration testing. They generated their intermediate model called state-activity diagram (SAD) and then generated test cases from SAD using the state-activity coverage criterion (Swain et al., 2010). In Ray et al. (2009), the authors proposed an approach for generating test cases from UML-ADs based on a conditioned slicing. First, a flow dependency graph has been built from UML-AD. Then, a conditioned slicing has been applied on all predicate nodes of the obtained flow dependency graph. Next, a suitable set of test cases has been generated for each slice previously generated. Finally, by pulling all practically useful test cases, the number of the generated test cases has been reduced. In Linzhang et al. (2004), the authors proposed an approach for generating test cases from UML-ADs based on the Gray-Box method. In Li et al. (2013), the authors proposed an approach for generating test cases from UML-ADs. First, a directed graph has been generated from UML-AD. Then, the Euler circuit has been constructed from the directed graph satisfying the transition coverage criterion. Finally, the test cases have been generated and minimised using the Euler circuit algorithm. In Sun et al., (2016) and Sun (2008), the authors proposed an approach and a tool based on Graph transformation for generating scenario oriented test cases from concurrent UML-ADs. They generated an intermediate model from UML-AD and then test scenarios have been generated from it. Finally, test data have been generated manually from the generated test scenarios and applied on the software application. In Arora et al. (2017), the authors applied a bio-inspired algorithm to generate test scenarios from the concurrent sections in UML-ADs.

Our approach is compared to research studies presented in (Chen et al., 2006; Boghdady et al., 2011; Kundu and Samanta, 2009; Sun et al., 2016; Sun, 2008). Boghdady et al. (2011), Chen et al. (2008), Sun et al. (2016) and Sun (2008) proposed intermediate models between UML-AD and test cases such as ADG (Activity Dependency Graph) presented in Boghdady et al. (2011), AG (Activity Graph) presented in Kundu and Samanta (2009) and EBTs (Extended ANDOR Binary Trees) presented in Sun et al. (2016) and Sun (2008). These intermediate models are used to facilitate the process of test case generation. However, they are insufficient to capture all test information in UML-AD; several aspects of UML-AD like the data flow, the guard conditions and the concurrency are not expressed in Boghdady et al. (2011) and Kundu and Samanta (2009). In order to be able to represent all test information, these intermediate models have been reinforced using data structures like activity dependency table (ADT) and node description table (NDT) presented in Boghdady et al. (2011) and Kundu and Samanta (2009) respectively. Furthermore, the way to generate test cases from particular structures in UML-AD like nested loops, complex structures of the concurrency and the synchronisation was not discussed in Sun et al., (2016) and Sun (2008). In Chen et al. (2006), the authors proposed the concept of simple path coverage criterion, which is used to generate test cases from concurrent activities of UML-ADs. The simple path is a very good way to avoid basic paths explosion with the existence of concurrency. However, the way of choosing a simple path among several basic paths has not been discussed. Furthermore, Chen et al. (2006) used the basic path coverage criterion with non-concurrent activities. But, the minimal loop testing is not ensured (see Subsection 3.1.2.1).

Our approach is different from related work presented in Chen et al. (2006), Boghdady et al. (2011), Kundu and Samanta (2009), Sun et al. (2016) and Sun (2008), mainly in our proposal intermediate model (EADG). EADG covers all information of UML-AD needed to generate test cases. It models data flow, branches, concurrency and loops. Any particular structure in UML-AD can be modelled using EADG and test scenario-oriented test cases can be generated from any complex structures of concurrency and loops such as forks and/or decisions within the concurrency and nested loops. In this work, using EADG, a simple and an effective method for choosing a simple path from several basic paths is proposed by taking the concurrent activities of each EADG node in their order during the test scenario generation process (see Subsection 3.1.1.1). Furthermore, the problem of the basic path coverage criterion with loops and nested loops has been solved by extending the definition of the basic path (see Subsection 3.1.2.1). Furthermore, the use of graph transformation with EADG has several advantageous. Graph transformation is defined using formal, graphical and high-level fashion formalism. Furthermore, with the flexibility of graph grammars, the maintenance of our test scenario generation tool becomes easier. Whenever one or several rules can be added, modified and/or deleted. This property also allows dealing with particular structures such as the problems of loop and nested loop structures in EADG and the complex structures of synchronisation and concurrency in UML-AD. For each particular structure one or several new graph grammar rules are defined. Another distinction between our work and related work is the automatic validation of test scenarios proposed by testers using a graphical simulation. This property increases the effectiveness of test scenarios set by giving testers the possibility to apply any particular test coverage criterion and validates it using our tool (see Section 3.2).

6 Conclusions

This paper have been dealt with software testing that is used to verify the correctness of a systems implementation by carrying out tests and making observations. The correctness criteria can be given in the specifications and UML is used often as model-based specifications. UML-AD is one of the important UML diagrams and it is used for modelling the global behaviour of a system. In this work, we have proposed an approach and a tool based on Graph transformation for generating automatically test scenarios from UML-ADs using the two coverage criteria: simple path and extended basic path. This approach also gives testers the possibility of validating their proposed test scenarios by applying them on UML-AD using a graphical simulation. In this approach we have proposed EADG as an intermediate model between UML-AD and test scenarios model in order to facilitate the transformation process. Our approach has been realised by proposing of two meta-models and three graph grammars. Meta-models are used for modelling UML-AD and EADG respectively. Graph grammars are used for performing the transformation of UML-AD diagrams into EADG models, for generating test scenarios from the obtained EADG model, and for validating test scenarios proposed by testers. All ideas presented above have been implemented using the graph transformation tool AToM3. Our approach and tool can detect more defects than existing approaches such as defects of loops and synchronisation errors. It has been applied to a college management system case study. The fault-detection capability of the generated test suite using our approach has been evaluated by MuJava mutation tool and given good results. As future work, we aim to generate test data automatically using the results of the work presented in Kerkouche et al. (in press), and to verify some properties of systems before testing them. We also plan to evaluate the capability of the concurrency fault-detection of our approach using other mutation tools.

Acknowledgements

The authors would like to thank Mr. Rachid Echahed (CNRS and university of Grenoble Alpes, France) for his words of advice and comments regarding this work.

References

Arora, V., Bhatia, R. and Singh, M. (2017) 'Synthesizing test scenarios in UML activity diagram using a bio-inspired approach', *Computer Languages, Systems Structures*, doi: 10.1016/j.cl.2017.05.002.

AToM^3 Home Page, version 3.00 [online] http://atom3.cs.mcgill.ca/ (accessed January 2019).

Binder, R.V. (1979) *Testing Object-Oriented Systems Models, Patterns, and Tools*, Addison Wesley, Reading, Massachusetts, ISBN 978-0321700674.

Boghdady, P.N., Badr, N.L., Hashem, M. and Tolba, M.F. (2011) 'Proposed test case generation technique based on activity diagrams', *International Journal of Engineering Technology IJET-IJENS*, Vol. 11, No. 3, pp.35–52.

Chen, M., Qiu, X. and Li, X. (2006) 'Automatic test case generation for UML activity diagrams', *AST6 Proceedings of the International Workshop on Automation of Software Test*, pp.2–8, New York, NY, USA, ISBN: 1-59593-408-1.

Chen, M., Mishra, P. and Kalita, D. (2008) 'Coverage-driven automatic test generation for UML activity diagrams', *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, Orlando, Florida, USA, ISBN: 978-1-59593-999-9.

Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J. and Li, X. (2009) 'UML activity diagram-based automatic test case generation for Java programs', *The Computer Journal*, Vol. 52, pp.545–556, DOI: 10.1093/comjnl/bxm057.

Chen, T.Y., Poon, P.L. and Tse, T.H. (2003) 'A choice relation framework for supporting category-partition test case generation', *IEEE Transactions on Software Engineering*, Vol. 29, No. 7, pp.577–593, DOI: 10.1109/TSE.2003.1214323.

Chen, T.Y., Kuo, F.C., Merkel, R.G. and Tse, T.H. (2010) 'Adaptive random testing: the art of test case diversity', *Journal of Systems and Software*, Vol. 83, No. 1, pp.60–66, DOI: 10.1016/j.jss.2009.02.022.

Chouhan, C., Shrivastava, V. and Sodhi, P.S. (2012) 'Test case generation based on activity diagram for mobile application', *International Journal of Computer Applications*, Vol. 57, No. 23, ISSN 0975-8887, DOI: 10.5120/9436-3563.

Chouhan, C., Shrivastava, V., Sodhi, P.S. and Soni, P. (2013) 'Test case generation on the origin of activity diagram for navigational mobiles', *International Journal of Advanced Computational Engineering and Networking*, Vol. 1, No. 2, pp.32–36, ISSN(p): 2320-2106.

Cimatti, A., Clarke, E.M., Giunchiglia, E. and Roveri, M. (2002) 'NuSMV 2: an opensource tool for symbolic model checking', in Brinksma, E. and Larsen, K.G. (Eds.): *Computer Aided Verification. CAV 2002. Lecture Notes in Computer Science*, Vol. 2404, Springer, Berlin, Heidelberg, Online ISBN: 978-3-540-45657-5.

Everett, G.D. and McLeod, R.J.R. (2007) *Software Testing: Testing across the Entire Software Development Life Cycle*, IEEE Press, John Wiley Sons, Inc., Hoboken, New Jersey, ISBN: 0470146346, 9780470146347.

Jatana, N., Suri, B. and Rani, S. (2017) 'Systematic literature review on search based mutation testing', *e-Informatica Software Engineering Journal*, Vol. 11, No. 1, pp.59–76, DOI: 10.5277/e-Inf170103.

Kerkouche, E., Khalfaoui, K. and Chaoui, A. (in press) 'A rewriting logic based semantics and analysis of UML activity diagrams: a graph transformation approach', *International Journal of Computer Aided Engineering and Technology*.

Kundu, D. and Samanta, D. (2009) ‘A novel approach to generate test cases from UML activity diagrams’, *Journal of Object Technology*, Vol. 8, No. 3, pp.65–83, DOI: 10.5381/jot.2009.8.3.a1.

Li, L., Li, X., He, T. and Xiong, J. (2013) ‘Extenics-based test case generation for UML activity diagram’, *Information Technology and Quantitative Management (ITQM2013), Procedia Computer Science*, Vol. 17, pp.1186–1193, doi: 10.1016/j.procs.2013.05.151.

Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L. and Guoliang, Z. (2004) ‘Generating test cases from UML activity diagram based on gray-box method’, *11th Asian Pacific Software Engineering Conference (APSEC04)*, pp.284–291, DOI: 10.1109/APSEC.2004.55.

Ma, Y.S., Offutt, J. and Kwon, Y.R. (2005) ‘MuJava: an automated class mutation system’, *Software Testing, Verification and Reliability*, Vol. 15, No. 2, pp.97–133, Doi: 10.1002/stvr.v15:2.

Mahali, P. and Acharya, A.A. (2013) ‘Model based test case prioritization using UML activity diagram and evolutionary algorithm’, *International Journal of Computer Science and Informatics*, Vol. 3, No. 2, pp.42–47, ISSN (PRINT): 2231-5292.

Malhotra, R. and Bharadwaj, A. (2012) ‘Test case prioritization using genetic algorithm’, *International Journal of Computer Science and Informatics*, Vol. 2, No. 3, pp.63–66, ISSN (PRINT): 2231-5292.

Makhija, J., Makhija, S. and Gulrajani, R. (2015) IICT Department Website [online] https://www.slideshare.net/jigarmakhija/college-department-management-system (accessed January 2019).

Ntafos, S.C. (2001) ‘On comparisons of random, partition, and proportional partition testing’, *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, pp.949–960, DOI: 10.1109/32.962563.

Object Management Group (OMG) Unified Modeling Language Specification, Version 1.5 [online] https://www.omg.org/spec/UML/1.5/About-UML/.(accessed January 2019).

OMG Unified Modeling Language TM (OMG UML), Version 2.5, FTF Beta 1 [online] http://www.omg.org/spec/UML/2.5/Beta1/ (accessed January 2019).

Ostrand, T.J. and Blacer, M.J. (1988) ‘The category-partition method for specifying and generating functional tests’, *Communications of the ACM*, Vol. 31, No. 6, pp.676–686, DOI: 10.1145/62959.62964.

Python Home page [online] htpp://www.python.org (accessed January 2019).

Ray, M., Barpanda, S.S. and Mohapatra, D.P. (2009) ‘Test case design using conditioned slicing of activity diagram’, *International Journal of Recent Trends in Engineering*, Vol. 1, No. 2, pp.117–120.

Rosenberg, G. (1997) *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, World Scientific Publishing Co., Inc. River Edge, NJ, USA, ISBN: 98-102288-48.

Stickyminds TechWell-Corporation [online] https://www.stickyminds.com/ (accessed January 2019).

Sun, C.A. (2008) ‘A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications’, *32nd Annual IEEE International Computer Software and Applications, COMPSAC’08*, pp.160–167, IEEE, DOI: 10.1109/COMPSAC.2008.74.

Sun, C.A., Zhao, Y., Pan, L., He, X. and Towey, D. (2016) ‘A transformation-based approach to testing concurrent programs using UML activity diagrams’, *Softw. Pract. Exper.*, Vol. 46, No. 4, pp.551–576, Doi: 10.1002/spe.2324.

Swain, S.K., Mohapatra, D.P. and Mall, R. (2010) ‘Test case generation based on state and activity models’, *Journal of Object Technology*, Published by ETH Zurich, Chair of Software Engineering, DOI: 10.5381/jot.2010.9.5.a1 [online] http://www.jot.fm.

Thevenod-Fosse, P., Waeselynck, H. and Crouzet, Y. (1991) 'An experimental study on software structural testing: deterministic versus random input generation', *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS91)*, 25–27 June, Montreal, Canada, pp.410–417, DOI: 10.1109/FTCS.1991.146694.

Utting, M. and Legeard, B. (2007) *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, ISBN: 0123725011; 9780080466484.

Zhang, N., Wu, B. and Bao, X. (2015) 'Automatic generation of test cases based on multi-population genetic algorithm', *International Journal of Multimedia and Ubiquitous Engineering*, Vol. 10, No. 6, pp.113–122, DOI: 10.14257/ijmue.2015.10.6.11.