AUTOMATIC TEST CASES GENERATION FROM UML ACTIVITY DIAGRAMS USING GRAPH TRANSFORMATION

Abdelkamel Hettab[1], Allaoua Chaoui[1], and Ahmad Aldahoud[2]

[1]MISC Laboratory, Department of Computer Science and its Applications, Faculty of NTIC, University Constantine 2, Algeria
[2]Faculty of IT, JUST University, Jordan

E-mail kamelhettab@yahoo.fr , a_chaoui2001@yahoo.com, black4online@yahoo.com

Abstract

Recently Model-based test case generation is attracting more attention of researchers in software development. This is due to the fact that this leads to the early detection of faults reducing software development time and cost. UML 2.0 activity diagram is one of the behavioral models used for test case generation. In this paper we propose an approach based on graph transformation to generate test cases from UML 2.0 activity diagrams. To this end, we propose two graph grammars. The first one transforms a UML activity diagram to an intermediate representation while the second one generates test cases from the intermediate representation. An example illustrates the approach.

Keywords - Test cases, Graph transformation, ATOM3, UML 2.0 Activity Diagrams

1 INTRODUCTION

The test has emerged as an indispensable technique for the validation of software. Indeed, whatever the complexity of developed software, testing is an essential complement to verification techniques. The test process is composed of three parts: the generation of test cases, the execution of these test cases, and test evaluation. The most difficult step is the generation of test cases. UML (Unified Modeling Language) is a language for visualizing, specifying, constructing, and documenting all aspects and artifacts of a software system [11]. Several researchers have used UML models to generate test cases [3, 4, 5, 6, 7, 8, 9]. UML diagrams are used to represent different complementary views of a system and its static and dynamic aspects. UML Activity diagrams are used to represent the workflow of activities and actions step by step in the application. They can be also used for the generation of test cases.

We propose in this paper an approach for automatic test cases generation from activity diagram as initial specification. Our approach is similar to [4] but it is based on graph transformation to generate automatically a set of paths from the UML activity diagram and then compares this set of paths with the program execution traces obtained by running the program under test with random test cases. This comparison allows us to determine the minimum set of test cases according to the coverage criteria for this test. This approach is also used to validate the program against its original specifications. Based on the techniques of graph transformation, we will propose a series of graph transformations using Atom3 [2] tool. There are also similar tools which manipulate models by means of graph grammars, such as PROGRES [21], GReAT [22], FUJABA [23], TIGER [24] and AGG [1].

The remainder of the paper is organised as follows. In section 2 we present related work. In section 3 we present the context of the paper and background. In section 4 we present our contribution consisting of the automatic generation of test cases from activity diagram. In

section 5 we present an example illustrating our approach. Section 6 concludes the paper and gives some perspectives of the work.

2 RELATED WORK

Model-Based Testing (MBT) is a type of test strategy that depends on the extraction of test cases from different models [12]. Unified Modeling Language (UML) is a standard modeling language and its models are best classified between models used in the literature [10, 11]. Many works are proposed for the automatic generation of test cases from UML models [3, 4, 5, 6, 7, 8, 9]. One reason is that UML models vary significantly from one technique development to another. Indeed, there are no standardized methods for the development of UML models. The authors of [3, 7] propose an algorithm that generates a sequence of test cases from state chart diagrams; for example the authors of [3] create a diagram called (IOLTSs) (*labeled transition systems over input / output -pairs*) for a subset of the state chart diagram. Then they generate a set of test cases from IOLTSs appropriate specifications. In [6] the authors propose a method that focuses on the generation of test cases from UML sequence diagram using a genetic algorithm to optimize the set of test cases. The authors of [4, 5, 8, 9] generate test cases from activity diagrams; for example in [4] the authors generate a set of paths from the activity diagram using modified DFS (depth-first search) algorithm. Then a comparison between these paths and the execution traces obtained by running the source program with a random test case they obtain the minimum set of test cases according to the specific coverage criteria. The major problem of this method is that it cannot guarantee that the selected test result can give a good coverage, because of the randomness. In addition, the generation of program execution traces is very long and may be impossible in many scenarios. To avoid this problem we combine the previous approach with the approach presented in [16], so we obtain the program execution traces in a deterministic way to save time and make a very robust test. The method presented in [16] manually identifies a set of test cases from the state chart diagram. In this work we propose an approach to automate it. In [5] the authors proposed an algorithm that automatically creates a table called ADT (Table Activity Dependency), and then this table is used to create a directed graph called ADG (Activity Dependency Graph). The generated ADG covers all the functionalities of the activity diagram. Finally, ADG and ADT are used to generate the final test cases. In [8] the authors proposed a similar method of [5] to generate test cases for mobile agents. In [9] the authors proposed a method as well; the UML activity diagram is translated into a formal model (input NUSMV [13]). Then, the properties in the form of CTL or LTL [14] formulas can be generated from the coverage criteria. Finally, the properties (negative version) are applied to the formal model using the model to produce the required tests (counterexamples).

3 BACKGROUND AND CONTEXT

In the literature, there are several approaches for the automatic test cases generation such as random, path-oriented, goal-oriented and intelligent approaches [15]. Our work belongs to the path-oriented approaches.

Techniques of path-oriented generally use control flow information to identify a set of paths to be covered and generate the appropriate test cases for these paths [15]. Furthermore, the generation of test cases from the specification is a very important work in the test phase. In this work we use UML activity diagram as a functional specification of the future system, and we propose an automatic approach for generating test cases; the principle of this approach is the following:

We generate test cases from the state chart diagram using a method similar to [16]. Then we execute the source program with these test cases to obtain the program execution traces corresponding to these test cases. Next, we compare these traces with the paths identified from the activity diagram and give the minimum set of test cases that satisfies the coverage criteria for this test.

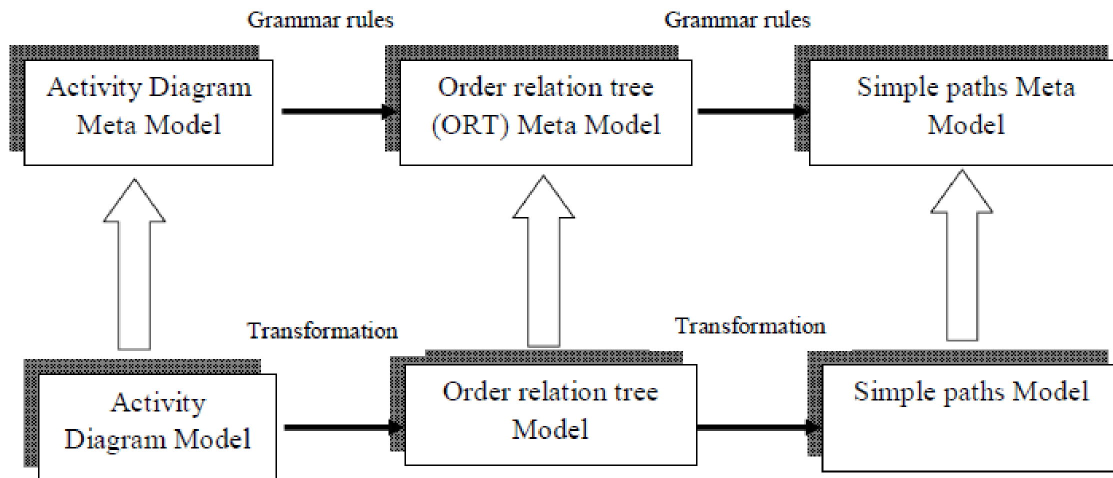


Figure 01: The architecture of our Approach

In this paper, we follow an MDA [18] approach. We present only a part of this work. We are interested in identifying the paths from the activity diagram in the following way:

First, using the technique of graph transformation [17] and Atom3 tool [2] we will make a transformation of the activity diagram into a graph called order relations tree (ORT) (see Figure 03). This model is used as an intermediate model between the activity diagram and the simple paths model. This model consists of nodes and arcs linking these nodes. Each node has two parts: the left side containing one or more actions and the right side containing one or more actions that are directly accessible from the actions of the left side in the execution order. We propose this model to represent the execution order of actions in the form of a tree whose root node contains the initial state of the activity diagram in the left side and actions directly accessible from the initial state on the right side.

Second, we will propose the rules of a graph grammar [17] to transform an ORT graph to simple paths. This method is presented in detail in section 4. This approach is illustrated in ***Figure 01***.

3.1 Activity diagram

The activity diagram is used to model a workflow in a use case or between use cases. It is also used to specify an operation (describe the logic operation), then the activity diagram is more appropriate to model the dynamics of a task, a use case where the class diagram is not yet stabilized.

The vision of the UML2.5 introduced significant changes in the semantics of diagrams; a variety of mechanisms of behavior specification is supported by UML2.5 [10]. The meaning of activity diagrams is explained in terms of concepts of Petri nets as token, flow, etc. By cons

in version UML1.x activity diagrams were defined as a kind of state machine diagrams. In this paper, according to UML2.5 [10], we adopt the semantics of activity diagrams to Petri nets semantics, but the syntax of activity diagrams has remained essentially the same from older versions.

Model elements are composed of nodes, edges and swim lane. Nodes represent processes or process control, including action states, activity states, decisions, forks, joins, objects, signal senders and receivers. The edges represent the sequence of activities, objects involving the activity, including control flow, message flow and signal flow. Activity states and action states are identified by rounded rectangles. Transitions are represented by arrows. Branches are shown as diamonds with an arrow incoming and multiple arrows outgoing; each arrow labeled with a Boolean expression to be satisfied choose the branch. Forks or joins are indicated by multiples arrows entering or leaving the synchronization bar. Swim lanes represent the supplier of activities. ***Figure 02*** shows the notations of elements of activity diagram.

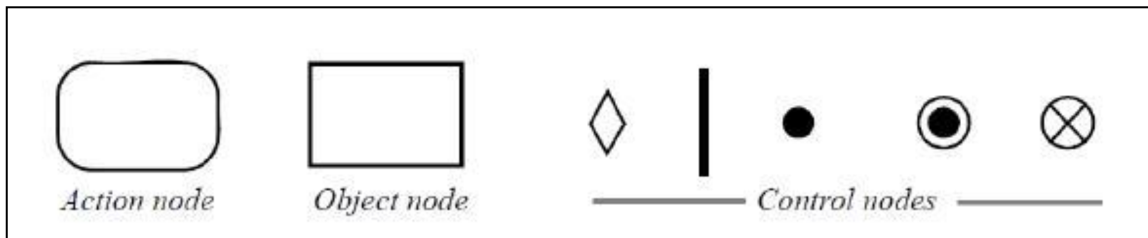


Figure 02 : Activity Nodes notation [10]

In this paper, we are interested by the control flow and data flow of activity diagrams that are relevant for the test cases generation. We call a path [4, 9] of activity diagram an instance of the dynamic behavior of an activity diagram. It can be represented by a sequence of states and concurrent transitions. If each activity in the path of an activity diagram occurs only once, we call it a basic path [4, 9]. There are many paths that have the same basic set of activities and even the partial order relation is used in the model checking. This is the so-called “All from one, one for all” [19]. We select a representative path from the set. The selected path is called a simple path [4, 9] of the activity diagram. Simple path is used for solving the problems of infinite loops and concurrency.

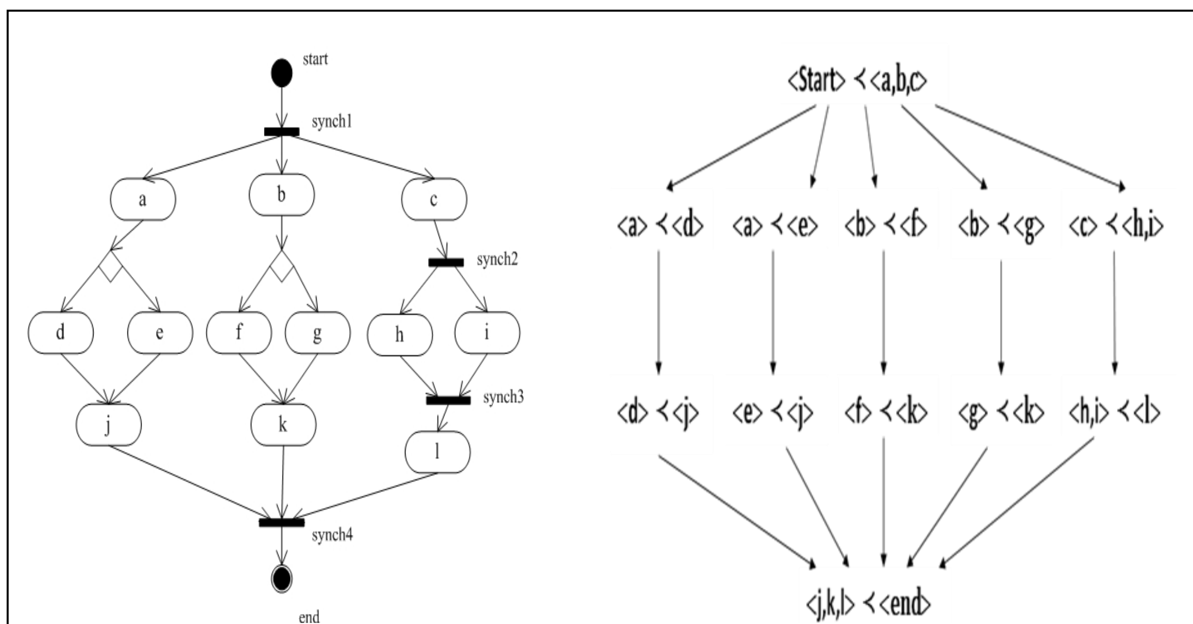


Figure 03: An Activity diagram and its corresponding ORT

3.2 Graph Grammar and AToM^3

AToM^3 [2] is an acronym for: A Tool for Multi-formalism Meta-Modeling, it is a model transformation tool, written in the Python [20] programming language. It uses and implements a set of concepts (modeling multi formalisms, the meta-modeling and graph grammars). The AToM^3 is used for modeling, meta-modeling and model transformation using graph grammars. Moreover, it can be extended to handle the simulation or generation code from models. Models are not simply drawn, but they are built by rules made by specification formalism. In AToM3 meta-models are specified using the UML class diagram or even using the entity-relationship diagram.

4 OUR APPROACH

We first begin with meta modelling the input and output formalisms. Then we propose our two graph grammars that perform the transformations.

4.1 Meta-Modeling of Used UML Diagrams and Formalisms

4.1.1 Activity diagram metamodel

Our proposed meta-model for UML Activity diagrams is shown in ***Figure 04.***

Figure 04: Meta Model of Activity Diagram

4.1.2 Order Relation Tree (ORT) Meta-Model

Our proposed meta-model for ORT model is shown in ***Figure 05.***

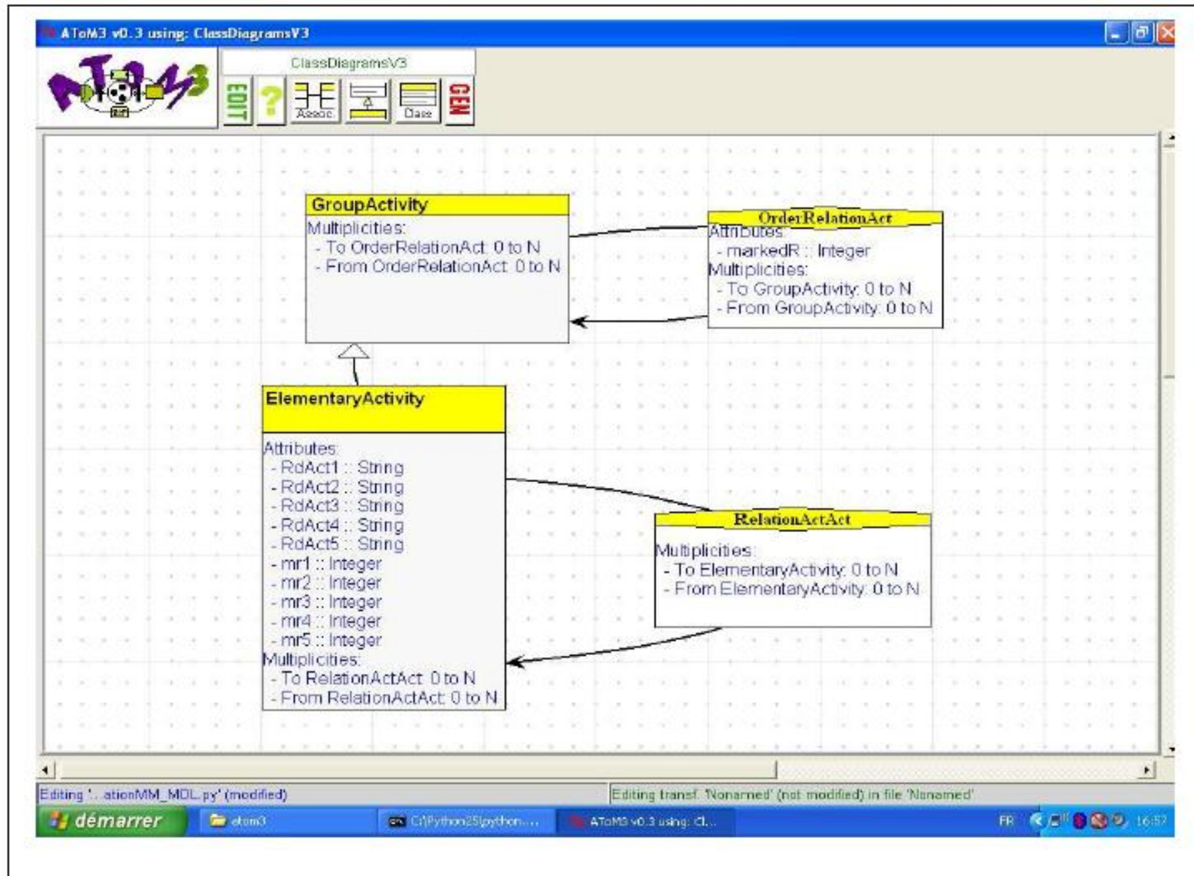


Figure 05 : Meta Model of ORT

4.1.3 Simple Path Meta-Model

Our proposed meta-model for Simple paths model is shown in ***Figure 06.***

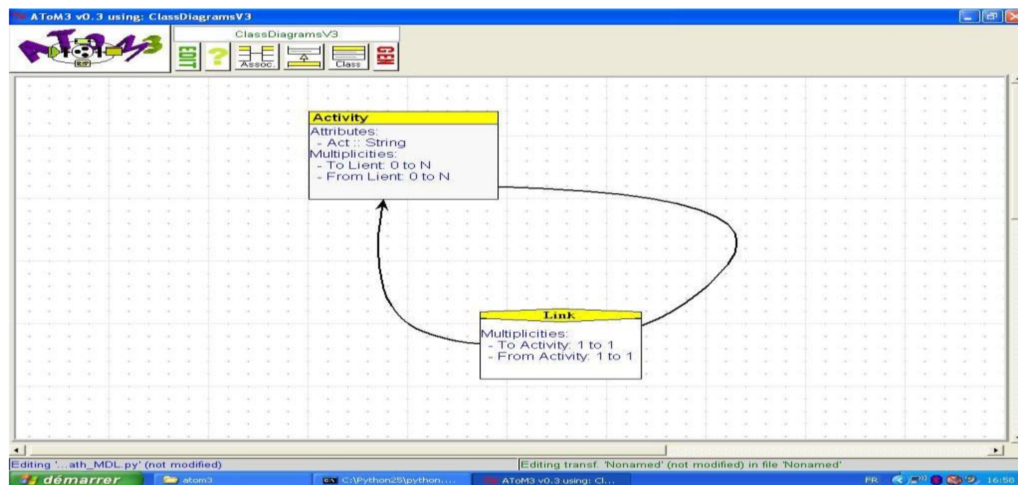


Figure 06 Meta-model for Simple paths model

4.2 Graph Grammars

In this section, we propose a graph transformation approach to transform the activity diagram into simple paths through the ORT model. To this end, we have defined two graph grammars. The first one converts each activity diagram to an ORT model, whereas the second graph grammar constructs simple paths from the obtained ORT model. In the following, we describe these graph grammars.

4.2.1 1st GG: Converting Activity Diagrams into ORT Models

We call this graph grammar ActivityDiagramToORT. It is used to convert any activity diagram having at most five concurrent activities in the system to its equivalent ORT model. This grammar is composed of sixty-one rules (see ***Figures 8***). The rules are applied in ascending order. Note that each rule has a priority. The idea of the grammar transformation ActivityDiagramToORT can be summarized in the following main steps:

The first step is to construct the root of the tree of ORT model from the initial node of the activity diagram and the actions that are directly related to this initial state or through one or more control nodes (Rules 1 - 6).

The second step is to construct the nodes of ORT model. So for every two actions of Activity diagram that follow directly or through one or more control nodes, we will create both left side and right side for a node of ORT model (Rules 7 - 25).

The third step is to construct the end node or end nodes of the ORT model using the same ways of previous steps (Rules 26-32).

The fourth step is to eliminate double definition nodes of ORT model as follows: for each action of the activity diagram that is related to two nodes of ORT model, we will separate this action into two actions respecting all the links between the two models (rules 32-48).

In the fifth step we will create relations between nodes of ORT model (Rules 48 - 55).

In the last step we will remove any elements of activity diagram and we obtain our ORT model (Rules 55 - 61).

4.2 2nd GG: Generation of paths from ORT

We call this graph grammar ORTToSimplePaths. It allows us to generate a set of simple paths from the ORT model. We proposed fifty-five rules (see ***Figures 9***) to be applied in ascending order. The idea of this graph grammar can be summarized in the following main steps:

The first step is to construct a set of actions that follow, from the root of ORT model (Rules 1-5).

The second step consists of constructing paths from each of the two nodes of ORT model having the root as the first node (Rules 6-27).

The third step continue the construction of paths in the same way as the previous steps from each of the two nodes of ORT model which are not roots (rules 28-51).

The fourth step consists of eliminating branching in paths. This rule has priority over all other rules because the branching can occur as soon as the first rule, then you must remove these branching early to keep the right sequence of actions in paths (Rule 52).

In the previous steps we marked all nodes of ORT model that are transformed into paths, and we marked also all links between these nodes. In the fifth step we will remove any marked link in the ORT model, and will also remove the links between elements of two formalisms ORT model and simple paths model. Whenever we apply these rules we obtain a simple path (rules 53-54).

In the last step we will delete all elements of ORT model, and gets all the simple paths (rule 55).

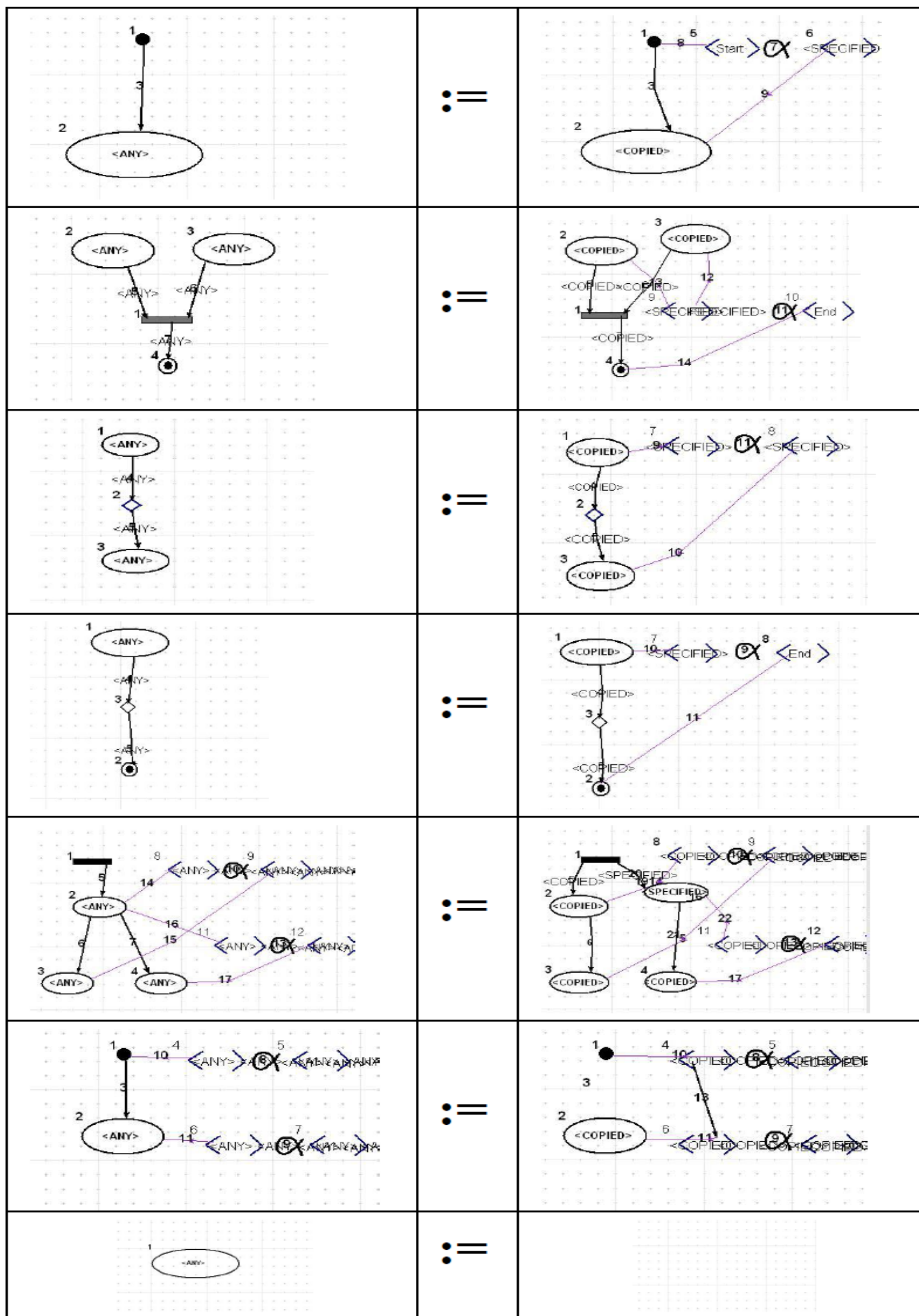


Figure 08: **Some rules of the 1st Graph Grammar**

	:=	
	:=	
	:=	
	:=	
	:=	
	:=	
	:=	

Figure 09: **2nd Graph Grammar some Rules**

5 Example

We have applied our two proposed grammars on the activity diagram borrowed from [4] (see ***Figure 10***).

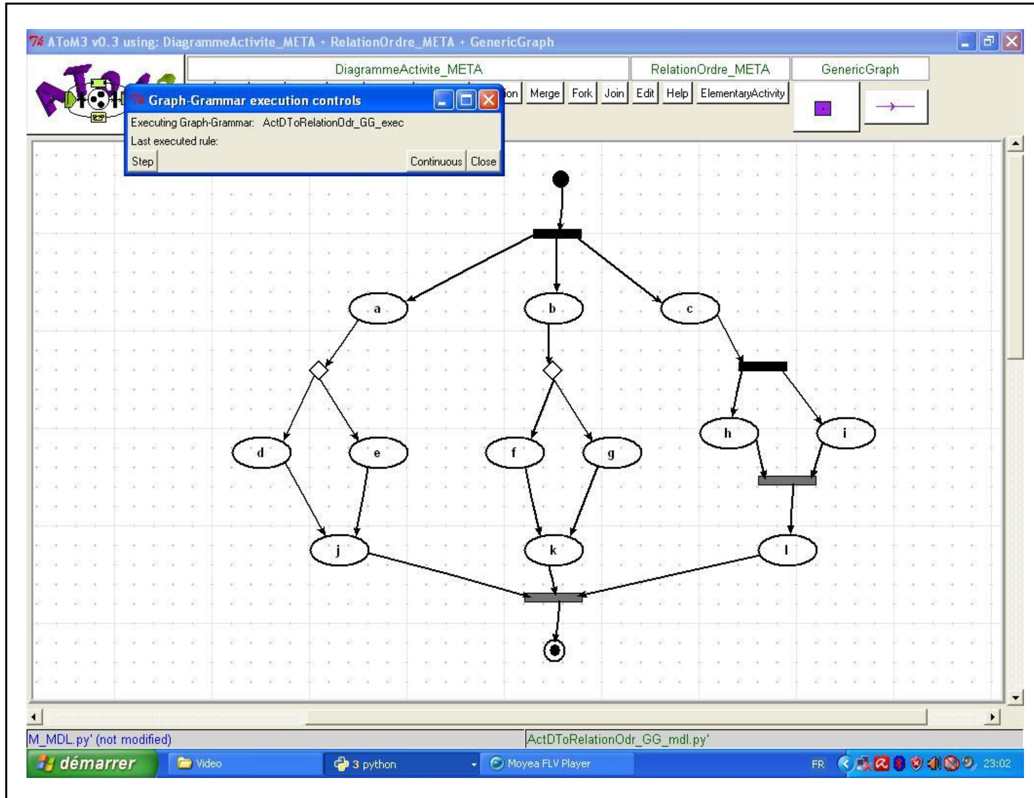
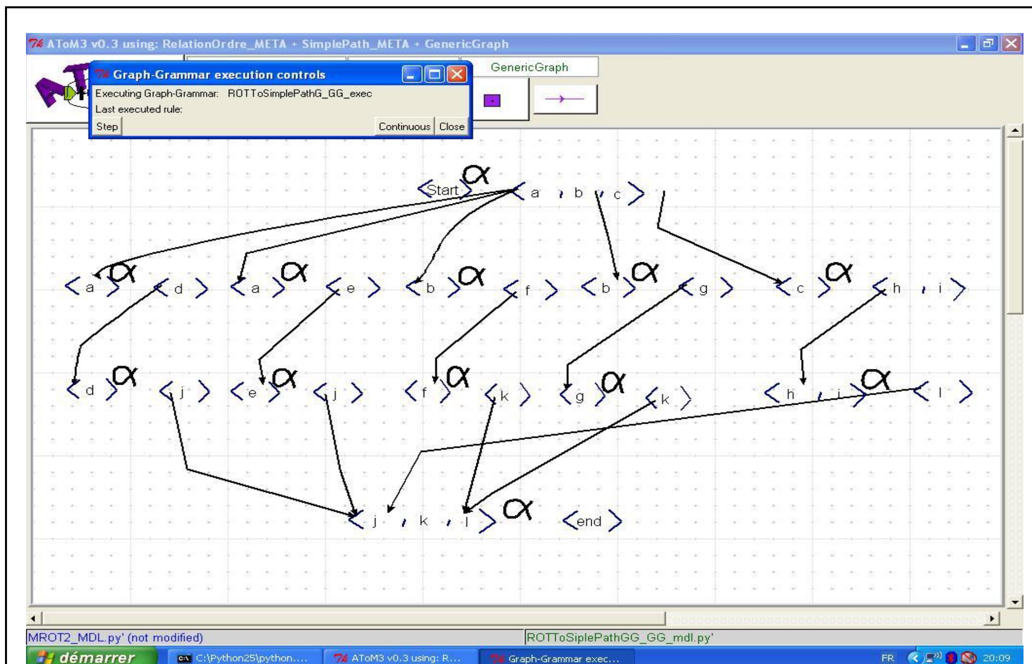


Figure 10: Activity Diagram

First we have applied ActivityDiagramToORT grammar on this activity diagram and obtained the ORT model of ***Figure 11***.



Then we have applied ORTToSimplePaths grammar on the ORT model of figure 11 and we have obtained all the simple paths of ***Figure 12***.

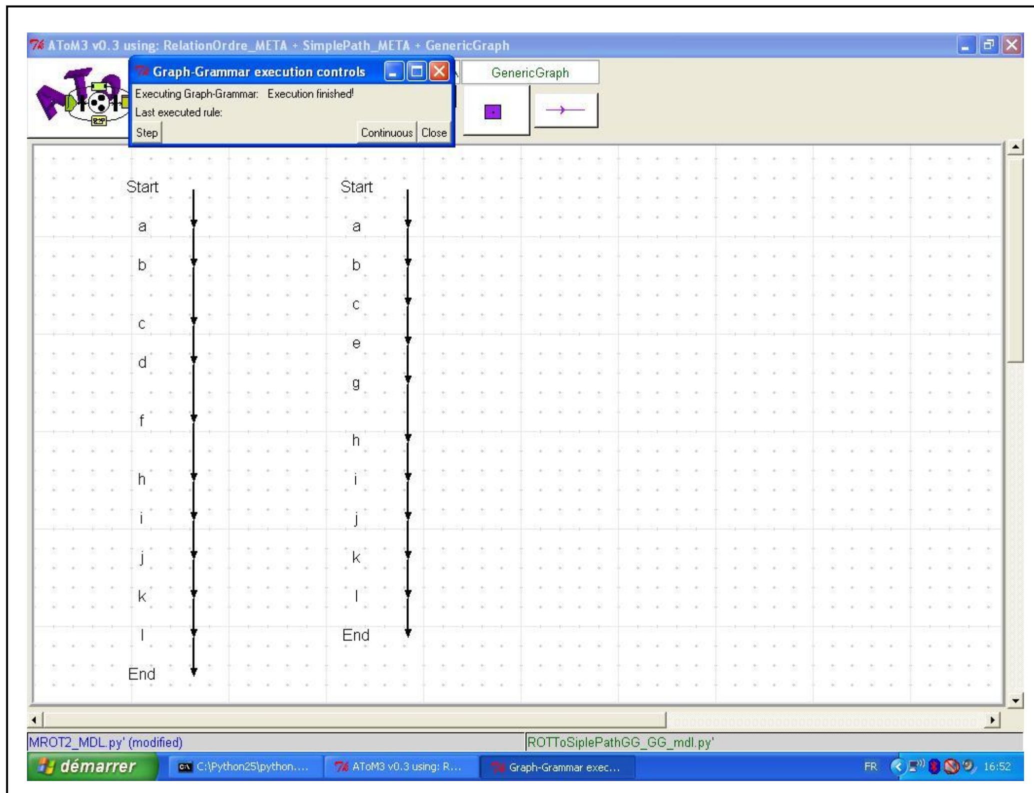


Figure 12: All simple paths obtained from previous ORT model

6 Conclusion

In this paper, we have proposed an approach to generate simple paths from UML activity diagram. This transformation aims to bridge the gap between two different notations; UML activity diagram and simple paths. It allows us to generate the minimum set of test cases that satisfy the coverage criteria. The approach is based on graph transformation since the input and the output of the transformation process are graphs. The Meta-modeling tool ATOM3 is used. An example illustrates our approach. This document is a step in the project to generate test cases from UML activity diagram including the state chart diagram. In a future work, we plan to generate test cases from the state chart diagram for the execution traces of program analyzed in a deterministic way, and a comparison between these execution traces and simple paths, we'll get verification and validation tests.

References

[1] AGG Home page, http://tfs.cs.tu-berlin.de/agg/

[2] AToM3 Home page, version 3.00, http://atom3.cs.mcgill.ca/

[3] Stefania Gnesi, Diego Latella and Mieke Massink Formal Test-case Generation for UML Statecharts Proc. 9th IEEE Int. Conf. on Engineering of Complex Computer Systems 2004.

[4] C. Mingsong, Q. Xiaokang, L. Xuandong. “Automatic Test Case Generation for UML Activity Diagrams”, Proceedings of the international workshop on Automation of software test, New York, NY,USA, 2006.

[5] Pakinam N. Boghdady, Nagwa L. Badr, Mohamed Hashem and Mohamed F.Tolba “Proposed Test Case Generation Technique Based on Activity Diagrams” Computer Engineering & Systems (ICCES), 2011 International Conference on.

[6] A.V.K. Shanthi and G. Mohan Kumar Automated Test Cases Generation from UML Sequence Diagram *2012 International Conference on Software and Computer Applications (ICSCA 2012) IPCSIT vol. 41 (2012) © (2012) IACSIT Press, Singapore*

[7] Ranjita Swain, Vikas Panthi, Prafulla Kumar Behera and Durga Prasad Mohapatra Automatic Test case Generation From UML State Chart Diagram International Journal of Computer Applications (0975 – 8887) Volume 42– No.7, March 2012

[8] Chanda Chouhan, Vivek Shrivastava and Parminder S Sodhi Test Case Generation based on Activity Diagram for Mobile Application *International Journal of Computer Applications (0975 – 8887) Volume 57– No.23, November 2012*

[9] M. Chen, P. Mishra, D. Kalita. “Coverage-driven Automatic Test Generation for UML Activity Diagrams”, Proceedings of the 18th ACM Great Lakes symposium on VLSI, Orlando, Florida, USA, 2008.

[10] OMG Unified Modeling Language TM (OMG UML) *Version 2.5 FTF – Beta 1*. October 2012 Available at *http://www.omg.org/spec/UML/2.5/Beta1/*.

[11] Object Management Group: OMG Unified Modeling Language Specification, Version 1.5, mars 2003.

[12] G.D. Everett, R. McLeod, Jr.” Software Testing: Testing across the Entire Software Development Life Cycle”, IEEE press, John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[13] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Roveri. NuSMV 2: An OpenSource Tool for Symbolic Model Checking, 2002.

[14] Christel Baier and Joost-Pieter Katoen Principles of Model Checking The MIT Press Cambridge, Massachusetts London, England 2007.

[15] M.Prasanna, S.N. Sivanandam, R.Venkatesan and R.Sundarrajan A SURVEY ON AUTOMATIC TEST CASE GENERATION Academic Open Internet Journal Volume 15, 2005.

[16] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae and S.D. Cha “test case generation from UML state diagrams”. IEE proceedings software, Vol. 146, No. 4, pp. 187-192, Aug. 1999.

[17] G. Engels, H.-J. Kreowski, G. Rosenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1, World Scientific, Singapore, 1999, pp. 551–604

[18] Object Management Group (OMG): MDA Guide Version 1.0.1, copyright 2003.

[19] D. A. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proc. of the* 5*th* *International Conference on Computer Aided Verification* (*CAV* _1993), pages 24–31. LNCS 697, Springer, 1993.

[20] Python Home page, htpp://www.python.org.

[21] PROGRES Home page, http://www-i3.informatik.rwth-aachen.de / research /projects/ progress / main.html.

[22] GReAT http://www.escherinstitute.org/Plone/tools/.

[23] FUJABA Home page, http://www.fujaba.de/

[24] TIGER Home page http://tfs.cs.tu-berlin.de/tigerprj/