

# A Graph Transformation of Activity Diagrams into Pi-calculus for Verification Purpose

Aissam Belghiat<sup>1,2</sup>

<sup>1</sup>Department of Computer Science

University of Mohamed Seddik Benyahia-Jijel, Jijel, Algeria

aissam.belghiat@univ-jijel.dz

Allaoua Chaoui<sup>2</sup>

<sup>2</sup>MISC Laboratory

University of Constantine 2-Abdelhamid Mehri, Constantine, Algeria

allaoua.chaoui@univ-constantine2.dz

## Abstract

Activity Diagrams have been used largely in modeling the behavior of control flow and data flow. Unfortunately, they suffer from lack of formal semantics due to its semi-formal nature as all UML diagrams, which prohibits any task of automatic verification. The use of formal methods has been adopted largely, but their interpretation generates another problem. Thus, this paper presents a user-friendly framework that is enabling intuitive visual modeling of systems using UML activity diagrams, and their verification using pi-calculus formal language, without the obligation to master this formal language.

**Keywords:** UML, Activity Diagram, Graph transformation, Verification, Pi-calculus

## 1 Introduction

UML (Unified Modeling Language) [OMG17] is considered as the standard visual modeling language that is used to specify, visualize, construct, and document artifacts in software systems. Its two main objectives are the modeling of systems using object-oriented techniques, from design to maintenance, and the creation

of an abstract language understandable by humans and interpretable by machines. Although UML is a rich language, with an open and widely used notation, its models still need to be checked to ensure that the behavior specified in these models is correct, and exactly meets the functional requirements of the system. This fact is due to the graphic and semi-formal nature of the UML language i.e. to its semantics which is not formally specified.

On the other hand, pi-calculus [Mil99] is a theoretical formal language that provides powerful tools for analysis and verification. It can be used to verify correctness of properties of a model, or check equivalence between two models.

Therefore, UML and pi-calculus have complementary characteristics; UML can be used for intuitive visual modeling while pi-calculus can be used for verification.

In this paper, we propose an approach that automates the mapping between UML activity diagrams towards pi-calculus. This work is inscribed in the context of the MDA (Model Driven Architecture) [OMG04] where model transformation is used and exploited. More precisely, the graph transformation, which is based on meta-modeling and graph grammar, is used to realize the model transformation as activity diagrams are graphs.

Building a modeling tool from scratch has never been a trivial task; on the contrary it was always difficult and hard. Meta-modeling approach has shown positive performances in dealing with this problem, because it gives freedom and easiness in modeling the formalisms themselves. A formalism model that contains enough information allows the automatic gener-

*Copyright © by the paper's authors. Copying permitted for private and academic purposes.*

In: Proceedings of the 3rd Edition of the International Conference on Advanced Aspects of Software Engineering (ICAASE18), Constantine, Algeria, 1,2-December-2018, published at <http://ceur-ws.org>

ation of a tool for constructing models that conforms to the syntax and semantics of the described formalism. The graph grammar is used then to transform the models into pi-calculus and returns an understandable analysis feedback to the user. We have used AToM3 (A Tool for Multi-formalism Meta-Modeling) [ATo02]; which is a tool that provides the mechanisms allowing the realization of these concepts.

The rest of the paper is organized as follows. In Section 2, some related works are exposed. In Section 3, we briefly present UML activity diagram, pi-calculus and graph transformation. In Section 4, the proposed framework is explained. In Section 5, an example is presented. Section 6 concludes the paper and gives some perspectives.

## 2 Related Works

The literature contains a broad range of works regarding giving formal semantics to UML diagrams in general and Activity Diagrams specifically. Among others we choose some works which are very close to ours. In [BCR00], the authors use ASM (Abstract State Machine) for representing behavior semantics of activity diagrams. In [Rod00], an FSP (Finite State Processes) formalism is adopted to formalize activity diagrams. In [BD00a, BD00b], the CSP (Communicating Sequential Processes) formalism is used to specify the execution semantics of activity diagrams. In [EW02] an activity hypergraph and a Kripke structure have been used as intermediate representations when transforming an activity diagram into the model checker NuSMV input language according to STATEMATE semantics. The last work has been enhanced in [Esh06] by adopting both STATEMATE and UML statechart semantics. Other works use Petri-nets and their extensions (Colored Petri-nets, High-level Petri-nets) to formalize activity diagrams such as in [SH05, Sto04a, Sto05, Sto04b, BG03].

There is also some works use pi-calculus as semantic domain to give formal semantics to activity diagrams such as in [Lam08] where the authors aim to check a model against its specifications expressed in modal mu-calculus, and also in [YZ04], where a full formal framework is proposed using pi-calculus. The automation of the translations is not provided.

To build automation translations, some works adopted CASE tools such as AToM3 tool which has been used with success in those projects. In [GD03], an AToM3 integrated framework has been developed for the verification of UML models using Petri Nets. They build meta-models for an UML design (composed of Class, Statecharts and Sequence diagrams) in addition to a Petri Nets meta-model. Then, they use graph grammars to translate the former to the later which al-

low their verification by model checking. In [Ker+10], the authors have used an UML design composed of statechart and collaboration diagrams to propose an AToM3 integrated approach for modeling and analysis of such models. They have used graph transformation in mapping the diagrams into Colored Petri net models. In [CEC12], the authors have used a subset of UML diagrams to develop an AToM3 integrated framework for their model checking by transforming them into a rewriting system expressed in the Maude language. Other contributions deserve to be cited such as [Bel+14], [BC16] and [BCB16].

We notice that all previous contributions have not taken into account that the user is not specialized in most cases, and he does not master these formal languages, in addition most of them do not even provide automation, what hinders their usage. Thus, in contrast, we develop in this work an AToM3-based framework that automates the mapping of activity diagrams into pi-calculus. In addition we try to drive away the verification task from users by interpreting feedback analysis results. The AToM3 tool is chosen because it offers the capabilities we need to realize our ideas.

## 3 Background

### 3.1 UML Activity Diagram

The UML activity diagram [OMG17] is used for modeling control flow and data flow. It gives an explanation of the sequence of activities and actions specific to an operation or a use case. It provides a set of elements that allow a very rich expression of any sequence in a system, its notation is relatively close to the state-transition diagram in its presentation, but its interpretation is significantly different. The activity diagram is essentially composed of activities and transitions. An activity specifies a behavior described by an organized sequencing of units whose basic elements are actions. The most common types of actions are: *call operation*, *call behavior*, *send*, *accept event*, *accept call*, *reply*, *create*, *destroy*, and *raise exception*. Each of them is used to represent the adequate behavior. A transition materializes the transition from one activity to another, it is triggered when the source activity is completed and immediately causes the start of the target activity. Therefore, transitions allow specifying sequence of treatments and define the control flow. Activity diagrams provide the mechanism for partitions, called *swimlanes* which allow organizing the nodes of activities by making regroupings.

### 3.2 Pi-calculus

The pi-calculus [Mil99] is a formal language that has solid mathematical bases. It is a computing model

that is used to represent concurrent and mobile systems by the expression of interactions between evolving processes, its two basic concepts are *Names* and *Processes*. A *Name* represents channels(ports), variables, data while a *Process* represents a communicating entity in a system. The syntax of the pi-calculus process expression is given in Table 1:

Type	Representation	Meaning
Input prefix	$a(x).P$	Receives $x$ through $a$ , then behaves as $P$
Output prefix	$\bar{a}\langle x \rangle.P$	Outputs $x$ on $a$ , then behaves as $P$
Summation	$P_1 + P_2$	Behaves as either $P_1$ or $P_2$
Composition	$P_1   P_2$	Behaves as in parallel $P_1$ and $P_2$
Restriction	$\text{new } y (P)$	$y$ can't be used for communicating
Matching	$[x=y].P$	Do $P$ when $x=y$

Table 1: pi-calculus Process Expression

### 3.3 Graph Transformation and AToM3

Graph transformation is one of the approaches used to implement model transformation. It consists on mapping a source graph into another target graph, by using a combined technique of meta-modeling and graph grammar. AToM3 [ATo02] is a powerful model transformation tool that implements the ideas of graph transformation.

The meta-modeling allows specifying the abstract syntax (the relationships between the elements) of any formalism, and its concrete syntax (the graphical notation that must be respected by models).

Graph grammars [Roz97] are generalization of Chomsky grammars for graphs. In AToM3, a graph grammar is composed of an initial action, multiple rules and a final action. Initial and final actions are used to provide necessary information before and after the execution of the rules. Each rule has two graphs; one on its left called the LHS (Left Hand Side) and another one on its right called the RHS (Right Hand Side). A rule is evaluated by comparing its LHS with a zone in an input graph (called host graph). If a matching is found, the rule will be executed and the corresponding matching sub-graph in the host graph will be replaced by the rule RHS. Furthermore, a rule may also have a condition that must be satisfied to apply the rule, as well as actions to be performed when the rule is carried out. The tool has a rewriting system that iterates applying the matching rules in the graph grammar to the host graph, until no rule is applicable. The rules are also ordered in this tool by a user-assigned priority (higher to lower).

## 4 The framework

### 4.1 Overview

In our framework, a combined meta-modelling and graph grammar approach is adopted to build an integrated tool using the AToM3 (see Figure 1). The approach consists of two essential tasks; firstly, we needed to propose a meta-model for activity diagrams to generate an AToM3 integrated environment which supports the visual modeling of these diagrams. Secondly, we have proposed and developed a graph grammar which gives us for each activity diagram modeled in the tool its corresponding pi-calculus process expression. This last is uploaded immediately in the mobility workbench (MWB) [VM94] to start the verification task. Thirdly, the results of the verification will be exported to the graph grammar (a feedback) that try to identify the problem found by interpreting these results and makes a sign on the diagram itself (the deadlock as example).

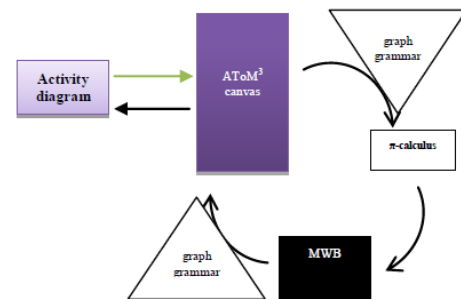


Figure 1: The framework architecture

### 4.2 Formalization of activity diagrams using pi-calculus

Activity diagrams must be provided with formal semantics in order to be able to verify any aspect of the behavior [Rod00]. To overcome this problem the pi-calculus can be used for the verification by a transformation approach of the activity diagrams into pi-calculus (Table 2)

### 4.3 Activity diagram Meta-model

Our meta-model is composed of 8 classes and 7 associations developed by the meta-formalism (CD-classDiagramsV3) in order to have an AToM3-based tool which offers the necessary tools to model the activity diagrams (see Figure 2).

After we have come to model our meta-model, only its generation remains. The generated meta-model contains the set of classes modeled as buttons that are ready to be used for a possible modeling of an ac-









Activity diagram	$\pi$ -calculus
<b>Initial node</b>	
	$IN_1(startIN_1, x_1) \stackrel{def}{=} startIN_1.\overline{x_1}.IN_1(startIN_1, x_1).$
The process corresponding to the initial state expects a signal from the environment to start execution by sending the $x_1$ channel.	
<b>Final node</b>	
	$AFN_1(x_1) \stackrel{def}{=} x_1.AFN_1(x_1)$
The process corresponding to the final state awaits receipt of the $x_1$ token to complete the execution.	
<b>Final flow</b>	
	$FFN_1(x_1) \stackrel{def}{=} x_1.FFN_1(x_1)$
The process corresponding to the final state awaits receipt of the $x_1$ token to complete the execution.	
<b>Action</b>	
	$A_1(x_1, x_2) \stackrel{def}{=} x_1.\tau.\overline{x_2}.A_1(x_1, x_2)$
The process corresponding to the action waits for the receipt of the token, to execute, after it passes the token to the next.	
<b>Fork node</b>	
	$FN_1(x_1, x_2, \dots, x_n) \stackrel{def}{=} x_1(\nu traversed) \left( \prod_{i=2}^n \overline{x_i}.traversed.0 \right) \overline{traversed}.\dots.\overline{traversed}.FN_1(x_1, x_2, \dots, x_n)$
The process corresponding to the fork node $FN_1$ waits to receive the token $x_1$ , it then behaves as multiple parallel sub-processes.	
<b>Join node</b>	
	$JN_1(x_1, x_2, \dots, x_n) \stackrel{def}{=} (\nu received) \left( \prod_{i=1}^{n-1} x_i.\overline{received}.0 \right) \overline{received}.\dots.\overline{received}.x_n.JN_1(x_1, x_2, \dots, x_n)$
The process corresponding to the join node $JN_1$ sends the $x_n$ token when it receives all the complete signals from sub-processes.	
<b>Merge node</b>	
	$MN_1(x_1, x_2, \dots, x_n) \stackrel{def}{=} \sum_{i=1}^{n-1} x_i.\overline{x_n}.MN_1(x_1, x_2, \dots, x_n)$
The process corresponding to the merge node $MN_1$ sends the $x_n$ token when it receives all the complete signals from sub-processes.	
<b>Decision node</b>	
	$DN_1(x_1, x_2, \dots, x_n, c_1, c_2, \dots, c_n) \stackrel{def}{=} x_1.(\nu x)\overline{c_n}(x).x(y). \left( \sum_{i=1}^{n-1} [y = c_i]\overline{x_{i+1}}.DN_1(x_1, x_2, \dots, x_n, c_1, c_2, \dots, c_n) \right).$
The process corresponding to the decision node $DN_1$ makes a non-deterministic choice to continue the execution, according to condition evaluation using $c_n$ channel.	

Table 2: Formalization of Activity Diagram

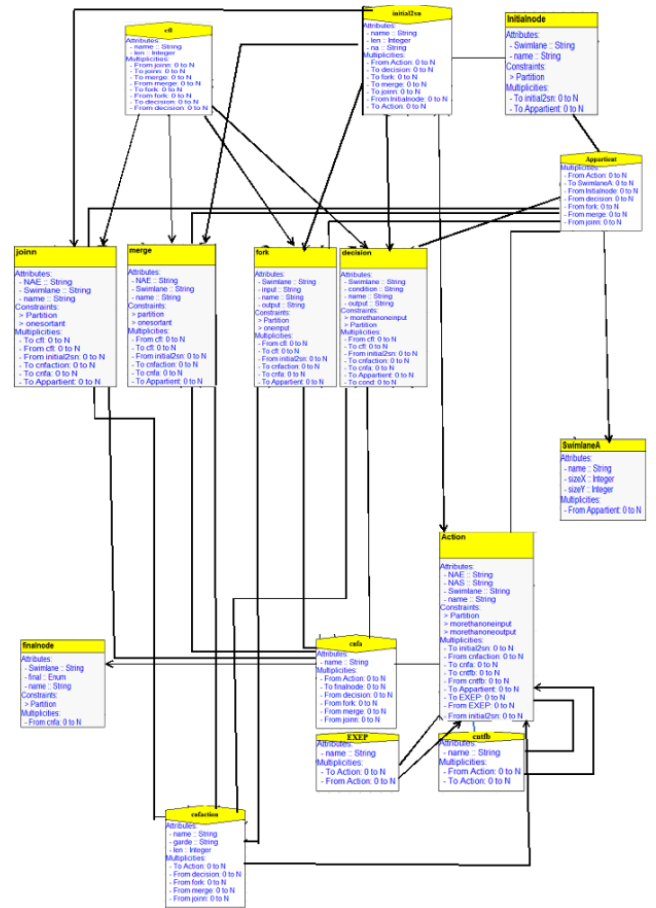


Figure 2: Activity diagrams meta-model under ATOM3

activity diagram. The generated environment is shown in Figure 3:

#### 4.4 The Graph grammar

The construction of graph grammar rules is an important step in the process of implementing a graph transformation. Indeed, it requires a good understanding of both languages. Thus, we have inspired the semantic rules from [Lam08].

Thus we propose the graph grammar (AD2Picalculus) composed of an initial action, 30 rules, and a final action. It should be noted that due to space constraints we cannot present all the rules, so we choose some rules among others and we join them with the Python code used for generating automatically pi-calculus code.

- *Initial action:*

**Role:** In the initial action of the graph grammar we have created a file with sequential access named "pi-calculus.txt" to store the generated pi-calculus code.

- *Final action:*

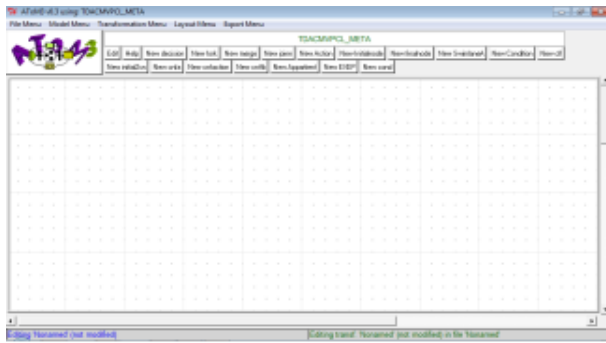


Figure 3: Activity diagrams environment under ATOM3

**Role:** In the final action of the graph grammar, we close the file "picalculus.txt".

- *Rules:*

**Rule 1 : Initial Node mapping**

**Name :** Initial2Initial

**Priority :** 1

**Role:** This rule makes it possible to transform an initial node that links by an action to the pi-calculus, in this rule we return the name of the initial node and the name of the outgoing arc, In the condition of the rule we test if the node is already transformed, otherwise the action of the rule opens the file "picalculus.txt" and adds the class in the pi-calculus code (see Figure 4).

**Condition :**

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "rule1_executed")
```

**Action :**

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
Aname = node.name.getValue()
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
transname = node1.name.getValue()
stateName = node.name.getValue()
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
transname = node2.name.getValue()
node.rule1_executed = True
obfichier = open("picalculus.txt", "a")
obfichier.write("agent "+stateName+"(start "+stateName+", "+transname+") "+
"start "+stateName+", "+transname+", "+stateName+"(start "+stateName+", "+transname+
") "+stateName+"")
```

Figure 4: Initial node mapping in the graph grammar

Similar rules are *initial2decision*, *initial2fork*, *initial2merge*, they are described in Figure 5. The difference is in the code python used to generate the pi-calculus code as already seen in table2.

**Rule 2 : Action mapping**

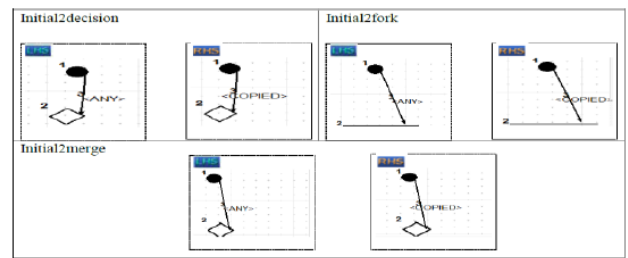


Figure 5: The rules *initial2decision*, *initial2fork*, *initial2merge*

**Name :** Action2picalcul

**Priority :** 2

**Role:**This rule allows transforming an Action with the incoming arc from an action and the outgoing arc that ends to an action, in this rule we return the name of the action and the name of the incoming and outgoing arc (see Figure 6).

**Condition :**

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "rule2_executed")
```

**Action :**

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
Aname = node.name.getValue()
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
transname = node1.name.getValue()
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
transname = node2.name.getValue()
node.rule2_executed = True
obfichier = open("picalculus.txt", "a")
obfichier.write("agent "+Aname+"("+transname+", "+transname+") "+
"start "+transname+", "+transname+", "+Aname+"("+transname+", "+transname+
") "+Aname+"")
```

Figure 6: Action mapping in the graph grammar

Examples of similar rules are *Action2decision*, *Action2Fork*, *Action2join*, *Action2merge*, *Action2final*, ...etc.

**Rule 3 : Fork mapping**

**Name :** fork2picalcul

**Priority :** 5

**Role:**This rule enables transforming a fork node that is linked by a single incoming arc from an action and outgoing arcs that end to actions (see Figure 7).

An example of a similar rule is *join2picalcul*. It is applied to locate a join node, and transforms it to pi-calculus code according to the semantic rules already described in Table 2.

**Rule 5 : Merge mapping**

**Name :** merge2picalcul

**Priority :** 3

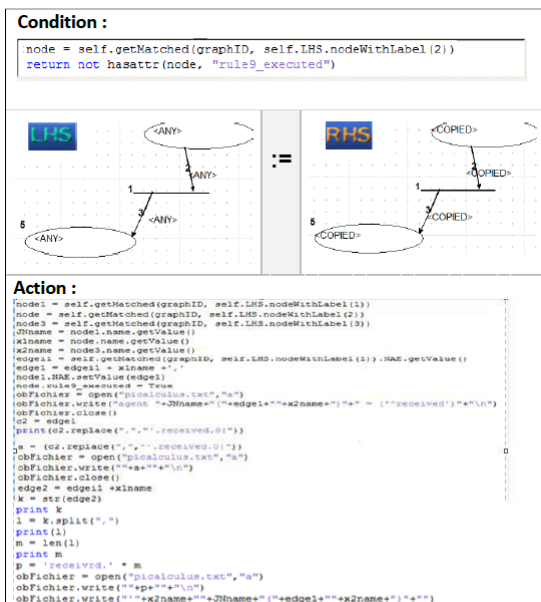


Figure 7: Fork mapping in the graph grammar

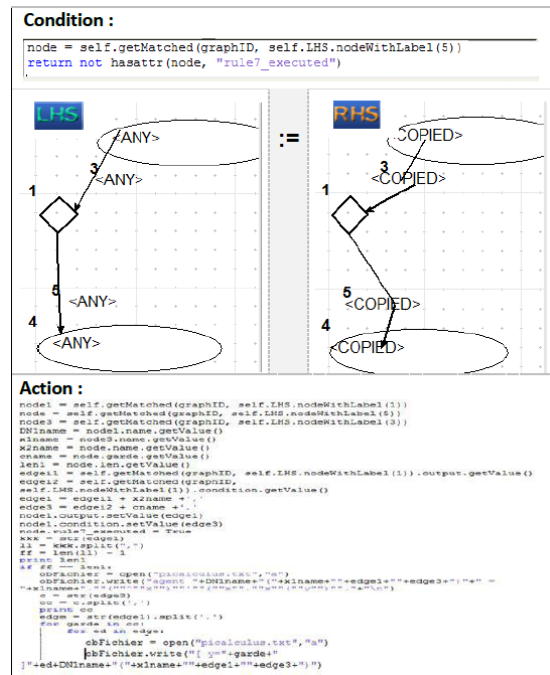


Figure 8: Merge mapping in the graph grammar

**Role:**This rule transforms a merge node with two or more incoming arcs from an action and an outgoing arc that ends to an action (see Figure 8).

An example of a similar rule is *decision2picalcul*. It is applied to locate a decision node, and transforms it to pi-calculus code according to the semantic rules already described in Table 2.

### 5 Example

In order to concretize the usefulness of the defined graph transformation, we tried to apply it to the simple example of the activity diagram presented in Figure 9. It should be noted that this example does not claim to be exhaustive, but it includes some important elements of an activity diagram such as: action, fork, join...etc.

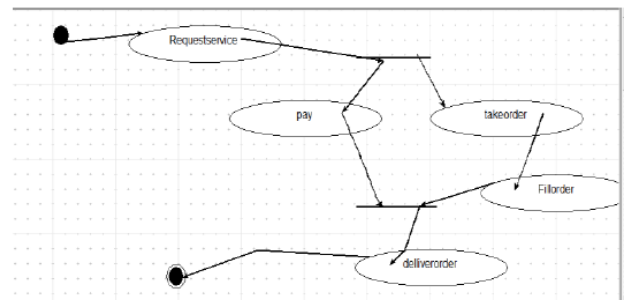


Figure 9: Example of an activity diagram

The generated pi-calculus code is described in Figure 10:

If we remove the transition between the action "deliverorder" and the final node, we create a deadlock in the diagram. Thus, the reloaded pi-calculus code will be similar except for the process (agent):

```

agent delliverorder(x6,x7) =
x6.t.delliverorder(x6,x7)

```

Our tool will analyze the code using MWB, it immediatly detects the deadlock (a state that has no outgoing transitions) and the process concerned. The final action of the graph grammar points out the problem by coloring the the corresponding element of the deadlocked process in the graph of the diagram (see Figure 11).

### 6 Conclusion

The result of our work is an automatic approach that enables visual modeling of systems behavior using UML activity diagrams and their verification using pi-calculus. The proposed approach is based on the graph transformation, and it is carried out using the AToM3 tool. The meta-modeling is used to define an environment for activity diagrams while the graph grammars are used to automate the translation and the analysis feedback. We saw in the last example the deadlock property, some other properties need more advanced feedback mechanisms to be understandable such as counter-examples. In future work, we plan to enable such mechanism by interpreting feedback analysis using Sequence Diagrams depicted in AToM3. In addition, we intend formalizing and integrating in our

```

picalculus.bt - Bloc-notes
Fichier Edition Format Affichage ?
agent IN(start IN,x) =startIN.'x.IN(startIN,x)
agent Requestservice(x,x1) =x.t.'x1.Requestservice(x,x1)
agent FN(x4,x4,x6) = ('/received')
x4.'received.0|x4.'received.0]
receivrd.receivrd.
'x6nom(x4,x4,x6)
agent deliverorder(x6,x7) =x6.t.'x7.deliverorder(x6,x7)
agent takeorder(x2,x8) =x2.t.'x8.takeorder(x2,x8)
agent FN(x1,x2,x3) = x1.'/traversed')
agent pay(x3,x4) =x3.t.'x4.pay(x3,x4)
(x2'.traversed.0|x3'.traversed.0|traversed.traversed.
FN(x1,x2,x3)
agent X7(AFN) = .X7(AFN)
agent F11order(x8,x4) =x8.t.'x4.F11order(x8,x4)
    
```

Figure 10: The generated pi-calculus code

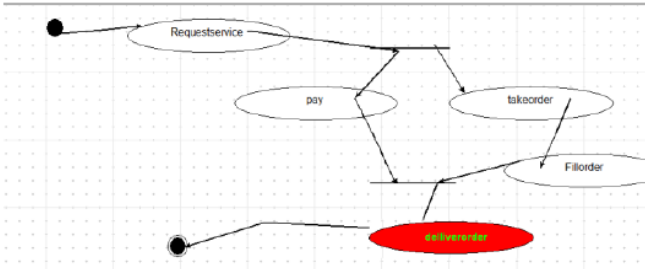


Figure 11: The analysis feedback on the diagram

framework other notational elements such as expansion region and interruptible activity region. We plan also to apply our approach to a wider range of real-world critical systems in order to experiment its performance.

References

[OMG17] Object Management Group (OMG). Unified Modeling Language (UML), Superstructure, v2.5, <http://www.omg.org/>, 2017.

[VM94] B. Victor, F. Moller. The Mobility Workbench -A Tool for the pi-calculus. In: Dill, D. (ed.) CAV 1994. LNCS, vol. 818, pp. 428-440. Springer, Heidelberg, 1994.

[Roz97] G.Rozenberg. Handbook of Graph Grammars and Comp (Vol. 1). World scientific. doi:10.1142/3303, 1997.

[Mil99] R. Milner. Communicating and Mobile Systems: The pi-calculus. Cambridge University Press, 1999.

[OMG04] Object Management Group (OMG), Model Driven Architecture (MDA), <http://www.omg.org/mda>, 2004.

[ATo02] AToM3 home page, (online) available at: <http://atom3.cs.mcgill.ca/>, 2002.

[BCR00] E. Borger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams.

8th International Conference, AMAST 2000, volume 1816 of LNCS, pages 293-308. Springer-Verlag, 2000.

[Rod00] R. W. S. Rodrigues. Formalising UML Activity Diagrams using Finite State Processes, UML2000 workshop.2000.

[BD00a] C. Bolton and J. Davies. On giving a behavioral semantics to activity graphs, in UML 2000 Workshop Dynamic Behavior in UML Models: Semantic Questions, 2000.

[BD00b] C. Bolton and J. Davies, Activity graphs and processes, in 2nd Int. Conf. Integrated Formal Methods, LNCS 1945, 2000, pp. 77-96.

[EW02] R. Eshuis and R. Wieringa. Verification support for workow design with UML activity graphs, in 22nd Int. Conf. on Software Engineering, ACM Press, 2002, pp. 166-176.

[Esh06] R. Eshuis. Symbolic model checking of UML activity diagrams, ACM Trans. Software Engineering and Methodology 15(1), 2006.

[SH05] H. Storrle and J. H. Hausmann. Towards a formal semantics of UML 2.0 activities, in German Software Engineering Conf. 2005, 2005.

[Sto04a] H. Storrle. Semantics of Control-Flow in UML 2.0 activities, in 2004 IEEE Symp. Visual Languages and Human Centric Computing, IEEE Computer Society, pp. 235-242, 2004.

[Sto05] H. Storrle. Semantics and verification of data flow in UML 2.0 activities, Electronic Notes in Theoretical Computer Science 127,35-52, 2005.

[Sto04b] H. Storrle. Structured nodes in UML 2.0 activities, Nordic Journal of Computing 11(3) (2004) 279-302.

[BG03] J. P. Barros and L. Gomes. Actions as activities and activities as Petri nets, in Workshop on Critical Systems Development with UML, 2003.

[Lam08] V.S.W. Lam. On pi-calculus semantics as a formal basis for UML activity diagrams. International Journal of Software Engineering and Knowledge Engineering 18(4), 541-567, 2008.

- [YZ04] D. Yang, S.S. Zhang. Using pi-calculus to formalize UML activity diagrams. In: 10th Int. Conf. and Workshop on the Engineering of Computer-based Systems, pp. 47-54. IEEE Computer Society, 2004.
- [GD03] E. Guerra and J. DeLara. A Framework for the Verification of UML Models. Examples using Petri Nets. In JISBD'03. Alicante,2003.
- [Ker+10] E. Kerkouche, A. Chaoui, E. Bourenane, O. Labbani. A UML and Colored Petri Nets Integrated Modeling and Analysis Approach using Graph Transformation. In Journal of Object Technology, vol. 9, no. 4, 2010, pages 25-43.
- [CEC12] W. Chama, R. Elmansouri, A. Chaoui. Model Checking and Code Generation for UML Diagrams using Graph Transformation. International Journal of Software Engineering and Applications (IJSEA), Vol.3, No.6, November, 2012.
- [Bel+14] A. Belghiat, A. Chaoui, M. Maouche, M. Beldjehem. Formalization of Mobile UML Statechart Diagrams using the pi-calculus: An Approach for Modeling and Analysis. In G. Dregvaite and R. Damasevicius (Eds.): ICIST, CCIS 465, pp. 236247. Springer, 2014.
- [BC16] A. Belghiat, A. Chaoui. Mapping Mobile Statechart Diagrams to the Pi-Calculus using Graph Transformation: An Approach for Modeling, Simulation and Verification of Mobile Agent-based Software Systems. International Journal of Intelligent Information Technologies (IJIIT) 12(4), 2016.
- [BCB16] A. Belghiat, A. Chaoui, and M. Beldjehem. Capturing and Verifying Dynamic Systems Behavior Using UML and Pi-Calculus. In Theoretical Information Reuse and Integration (pp. 59-84). Springer, 2016.