A Modeling and Verification Approach based on Graph Transformation

Wafa Chama, Raida Elmansouri, Allaoua Chaoui
MISC Laboratory
Department of Computer Science
Faculty of Engineering
University Mentouri Constantine
Algeria
wafachama@gmail.com, raidaelmansouri@yahoo.fr, a_chaoui2001@yahoo.com

ABSTRACT: *UML is a standard modeling language with an open notation and several concepts to be widely used in software modeling. However, UML suffers from a lack of formal semantics. So their models still need to be formally checked against incoherencies or inconsistencies. To reach this goal we propose in this paper, a framework and a tool based on graph transformation allowing an automatic translation of some UML diagrams to equivalent Maude formal specifications. To realize this automatic mapping we use UML Class diagram formalism to define three meta-models. The first one for Class Diagram, the second for State Machine Diagram and the third for the Communication Diagram. Then, we propose a graph grammar to generate Maude specifications of the UML diagrams based on theses meta-models. The meta-modeling tool AToM³ is used to produce our visual modeling tool according to the proposed UML meta-models. An example is presented to illustrate our approach.*

Keywords: UML, Rewriting System, Maude Specification, Meta-model, Graph Grammar, AToM^3 Tool, Automatic Code Generation

Received: 28 October 2013, Revised 2 December 2013, Accepted 9 December 2013

1. Introduction

UML (Unified Modeling Language) is a graphical modeling language used to specify, visualize, and construct applications and software systems. UML contains a big number of diagrams; some are used to model the structure of a system while others are used to model the behavior of this one.

However, the UML models developed can contain incoherencies or inconsistencies which are difficult to detect manually because UML suffers from a lack of formal semantics. Formal methods represent an interesting solution to face this problem.

In this paper we develop a formal framework allowing the automatic translation of three diagrams which are Class Diagram (models the static structure), State Machine Diagram (specifies the dynamic behavior of each object) and Communication Diagram (represents a collection of interacting objects) into its equivalent Maude code using AToM^3 as a graph transformation tool.

The rest of the paper is organized as follows. In section 2, we give an overview of related work while section 3 presents briefly rewriting system and Maude language. In section 4 we give a brief introduction of the AToM^3 tool. Section 5 details the proposed translation by defining the three meta-models of UML diagrams used (Class Diagram, State Chart Diagram and Communication Diagram) and giving the rules of the graph grammar proposed; while Section 6 describes a case study in order to illustrate our translation approach. Finally, we give a conclusion and some perspectives in section 7.

2. Related Work

In [10], the authors presented some rules for mapping UML diagrams to their equivalent Maude specifications. The translation is made *manually*. In [9], the author presented another approach for transforming UML diagrams to their equivalent Maude specifications. The translation is also made *manually*. In this paper we propose *an automatic approach* and a tool environment that formally transforms UML diagrams into their equivalent Maude specifications using the meta-modeling tool AToM3 and graph grammars. Our approach is inspired from the work presented in [9] and graph grammars.

3. Rewritin Logic and Maude

Rewriting logic [7] has been introduced by José Meseguer allowing concurrent software specification and verification. It is implemented by several languages such as Maude [8].

Maude is a specification and programming language. It is simple, expressive and has a high-performance implementation. Maude defines three types of modules: Functional modules, System modules and Object-Oriented modules.

Functional modules allow us to define data types and their properties by the definition of signatures and equations; but the dynamic behavior of a system is defined by the use of rewrite laws which we introduce in *System modules*, these laws take the form (1).

$$R\text{: } [t]\ [t'] \text{ if } C \tag{1}$$

Which indicates that, according to rule $\boldsymbol{R}$, term $\boldsymbol{t}$ rewrites to $\boldsymbol{t'}$ if a certain condition $\boldsymbol{C}$ is verified. The condition C is optional, so rules can be unconditional. Finally, *Object-Oriented modules* add more appropriate syntax to describe the object paradigm such as objects, messages and configurations. Maude offers *"full Maude"* to support that; furthermore, it has its own model-checker that is used in checking system's properties.

4. ATOM^3 Tool

AToM^3 [2] is a visual tool used for multi-formalism modeling and Meta-Modeling. The two main tasks of AToM^3 are meta-modeling and Model transformation.

The first task refers to modeling formalisms concepts using Entity Relationship formalism or UML Class Diagram formalism. The second one uses Graph Grammar. It is composed of production rules [3]; each having graphs in their left hand side (LHS) and right hand side (RHS). For more details the reader is referred to [5].

5. The Proposed Approach

The steps of our proposed approach are as follows:

5.1 Meta-Modeling of Used UML Diagrams

In order to translate UML diagrams to equivalent Maude specifications, we propose three meta-models; the first one for the Class Diagram, the second one for State Machine Diagram and the third one for the Communication Diagram. Theses meta-models are represented by UML Class Diagram formalism and the constraints are expressed using Python code.

5.1.1 UML Class Diagram Meta-Model

A Class Diagram [1] is a type of static structure diagram. It represents the main building block in object-oriented modeling; it contains classes, their attributes, and their relationships: association, aggregation, composition, generalization and several types of dependencies. Our meta-model for UML Class Diagram (see Figure 1) is composed of the following classes:

a) Class Diagram
This class has a "*name*" and represents a Class Diagram.

b) Class_simple
This class describes the classes and has three attributes, namely "*class_attribut*", "*class_name*", and "*class_op*".

c) Association_simple
This class represents a simple relationship between two classes, and has three attributes: "*ass_name*", "*caleft*", and "*caright*" to indicate the multiplicity of instances (the number of objects that participate in the association).

d) Association_attribut
An association can possess its own properties, which can be introduced by this class. It inherits from Association_simple all its attributes, multiplicities, associations plus an attribute "*ass_attribut*".

e) Association_multiple
Higher order associations can be drawn with more than two ends, This class inherits from Association_simple all its properties with its own attribute "*carbas*".

f) Composition
This class describes a composition, has two attributes "*com_name*" and "*card* ".

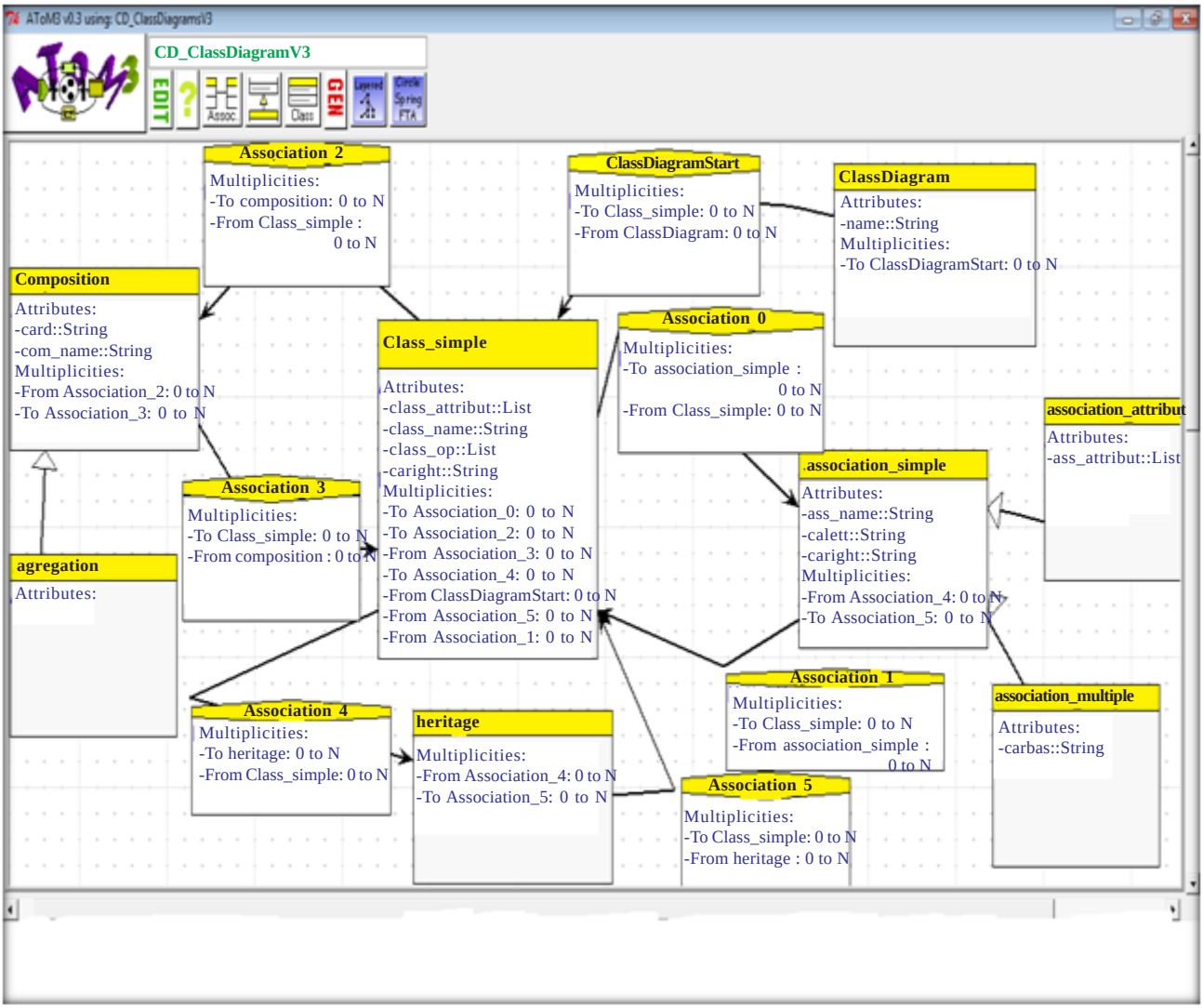

Figure 1. Class Diagram meta-model

g) Agregation
This class represents an aggregation. It inherits from Composition all its attributes, multiplicities and associations.

h) Heritage
This class represents a generalization relationship (is also known as the inheritance or "*is a*" relationship).
We have also associations included in the meta-model to express hierarchy (see Figure 1); they are drawn as invisible links.

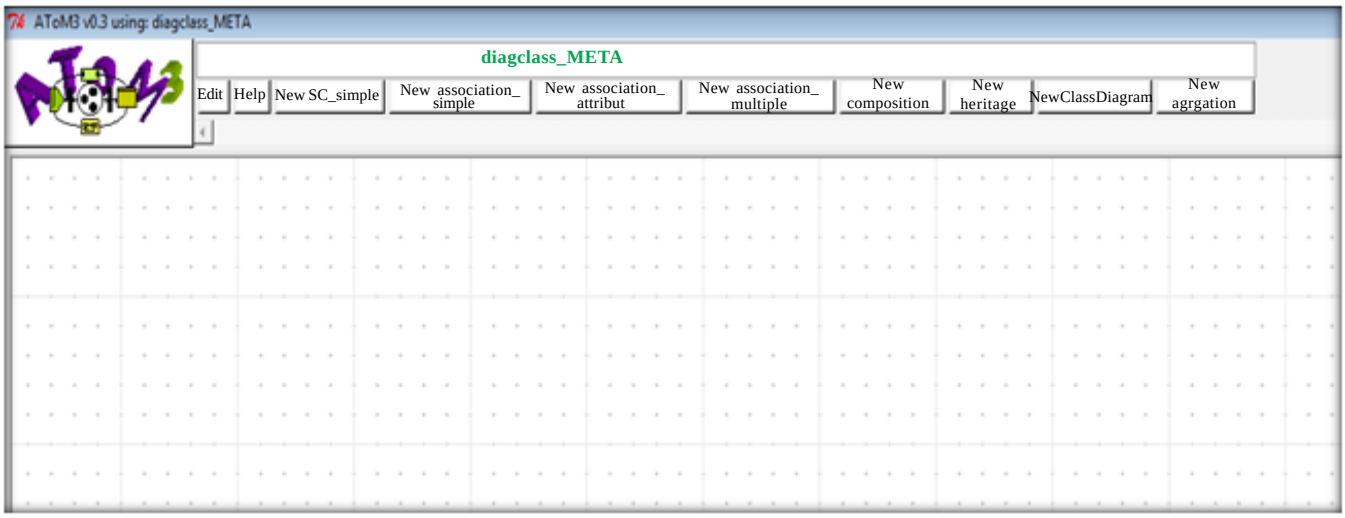

Figure 2. A tool for Class Diagram generated using AToM^3

And from this meta-model we generate a tool to manipulate the Class Diagram as shown in the tool bar of Figure 2.

5.1.2 UML Statechart Diagram Meta-Model
A State Chart diagram [1] is used to model the behavior of a system; contains states and other types of transitions (events and actions); states may also contain sub diagrams called Composite states which can be sequential or concurrent. State Chart transitions are denoted by standard finite state machine arcs that define a change from one state to a successor one.

Our meta-model for UML State Chart diagrams (see Figure 3) is composed [4] of the following classes:

a) StateChart
This class has an attribute "*Name*" and represents a State Machine in the diagram.

b) SC_Initail
This class marks the initial state of a statechart diagram or the initial state of a composite state.

c) SC_Final
This class marks the final state of a statechart diagram.

d) SC_State
This class describes simple states and it has three attributes, namely "*Name*" (textual string for identification, can be anonymous), "*EntryAction*" and "*ExitAction*" (actions executed on entering and exiting the state respectively).

e) SC_CompositeState
It represents the composite states and inherits from SC_State all its attributes, multiplicities and associations.

Associations are also included in the meta-model to allow the connections between the differences classes (see Figure 3).

And from this meta-model we generate a tool to manipulate the StateChart diagram as shown in the tool bar of Figure 4.

5.1.3 UML Communication Diagram Meta-Model
A UML Communication diagram is a type of interaction diagrams which describes the dynamic behavior of a system, it models the interactions between objects by sending messages. Our meta-model contains two classes:

a) CommunicationDiagram
This class has a "*com_name*" and represents Communication Diagram.

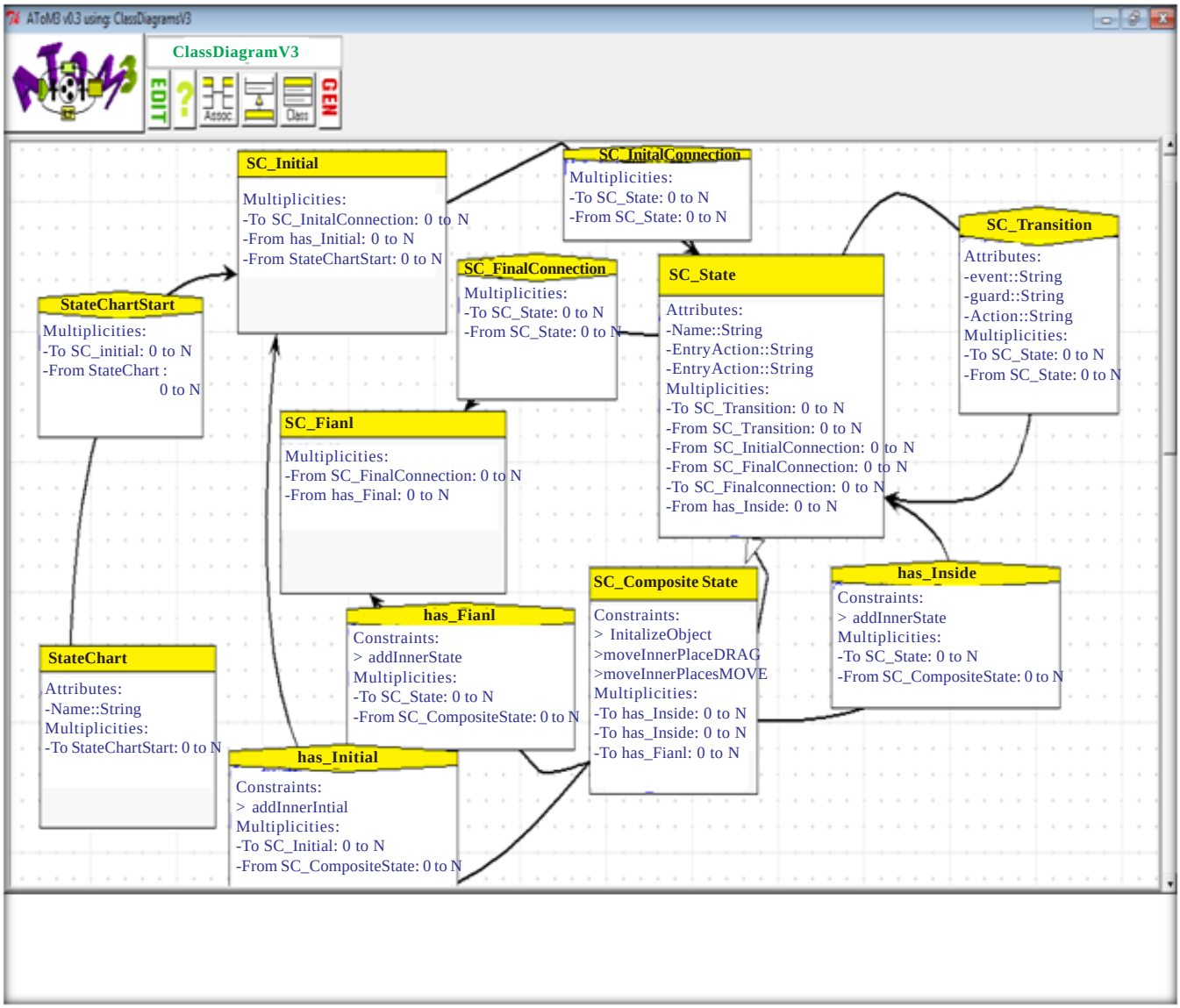

Figure 3. StateChart diagram meta-model

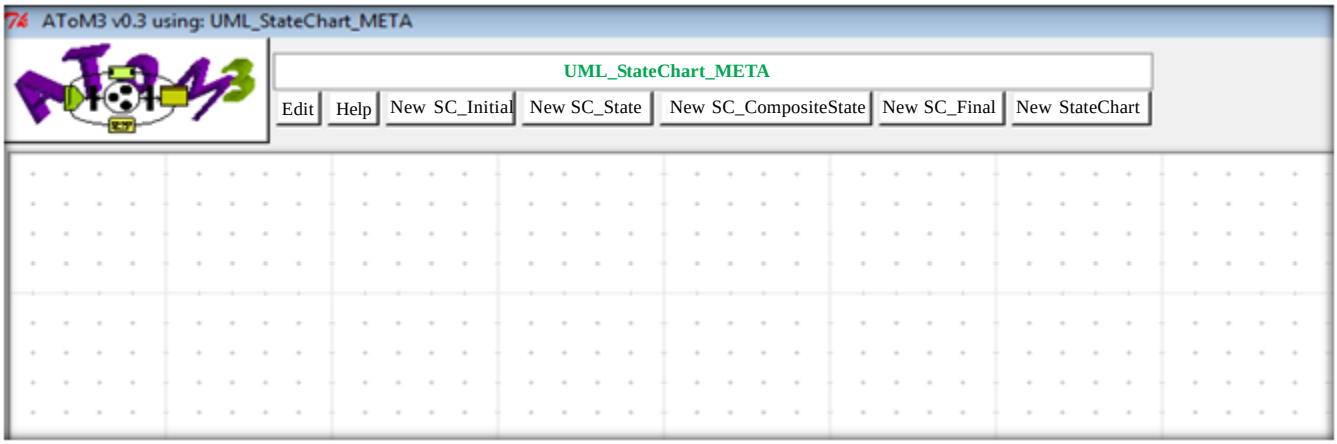

Figure 4. A tool for StateChart Diagram generated using AToM^3

b) Collaboration

This class represents an object interacted, and has one attribute "*name_coll*"; and one association named "*relationcoll*" for representing messages as shown in Figure 5.

And from this meta-model we generate a tool to manipulate the Communication diagram as shown in the tool bar of Figure 6.

5.2 Generation of Maude Specifications

We have proposed a graph grammar containing eighteen rules which will be applied in ascending order (each rule has a priority), none of these rules will change the UML models because we are concerned by code generation (Maude Specifications).

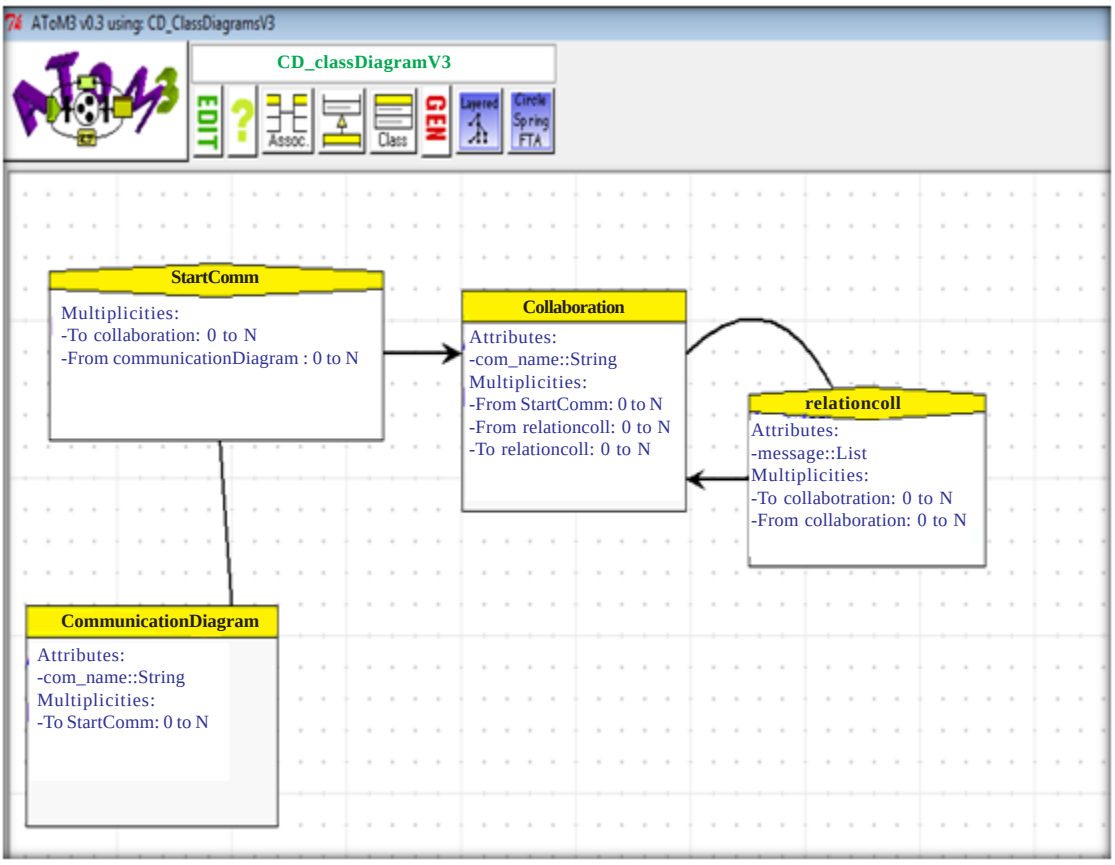

Figure 5. Communication diagram meta-model

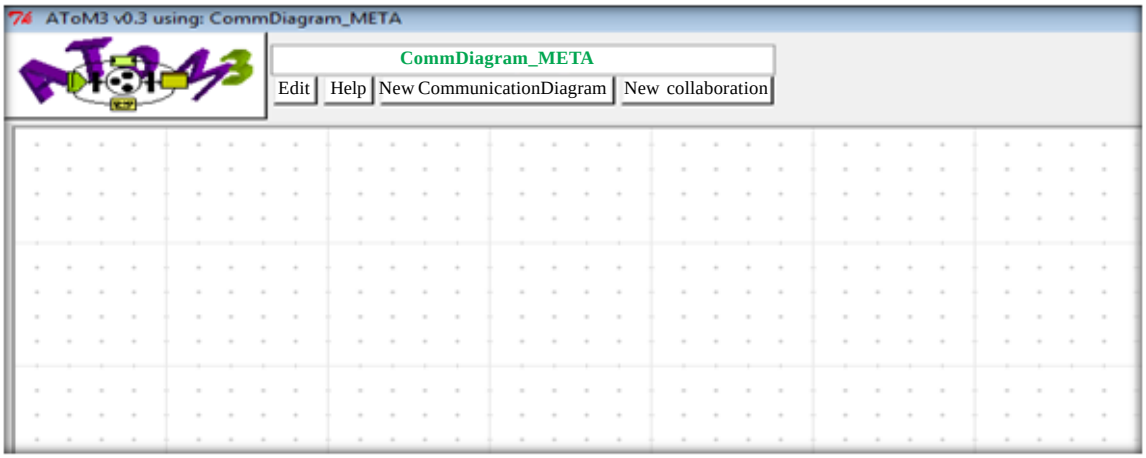

Figure 6. A tool for Communication Diagram generated using AToM^3

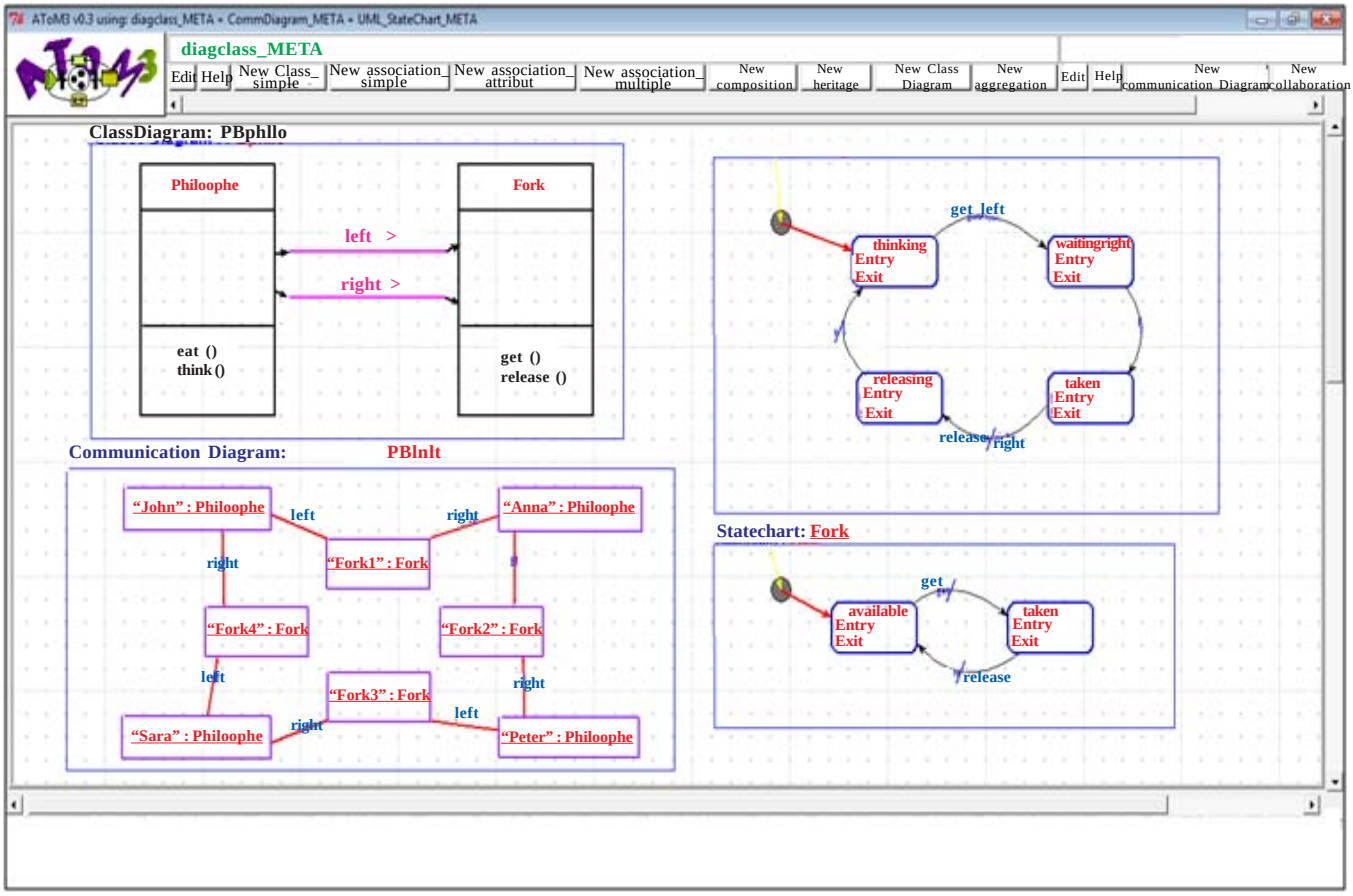

Figure 7. Example of dining philosopher's problem created in our framework

• Rules 1, 2 Nfile, ExtrInf (Priority resp 1, 2)
These rules are applied to locate a class not previously processed (Visited = = 0), and create a file for each one.

Note that each UML Class with its own StateChart is represented with an object-oriented module in Maude [9].

• Rule 3 VStateChart (Priority 3)
It is applied to generate the appropriate Maude syntax (add the concepts for states "*simple or composite*" in files which is related to classes) depending on the condition (nom_class = = nom_stchart).

Note that [9] to represent states, we declared an algebraic structure as in (2).

```
sorts SIMSTATE COMSTATE STATE .
subsort SIMSTATE < COMSTATE .
subsort COMSTATE < STATE .                                    (2)
op none : -> COMSTATE [ctor] .
op _||_ : COMSTATE COMSTATE ->
COMSTATE [ctor assoc comm id: none].
```

• Rules 4, 5, 6 DefClass, AssoSimple, EndClass (Priority resp 4, 5, 6)
These rules are applied to generate Maude code, and marks the association as visited (Asso.Visit = 1).

Note [9] that to define a class, we can use the following syntax (3):

$$\textit{Class}\ \text{class\_name} \mid \textit{Status} : \text{STATE}, \text{attr}_1 : \textit{sort\_attr}_1 \ldots\ \textit{attr}_n : \textit{sort\_attr}_n,\ \textit{asso\_name} : \textit{Oid} \qquad (3)$$

• Rules 7, 8, 9 EtatInit, EtatFin, EtatSimple (Priority resp 7, 8, 9)

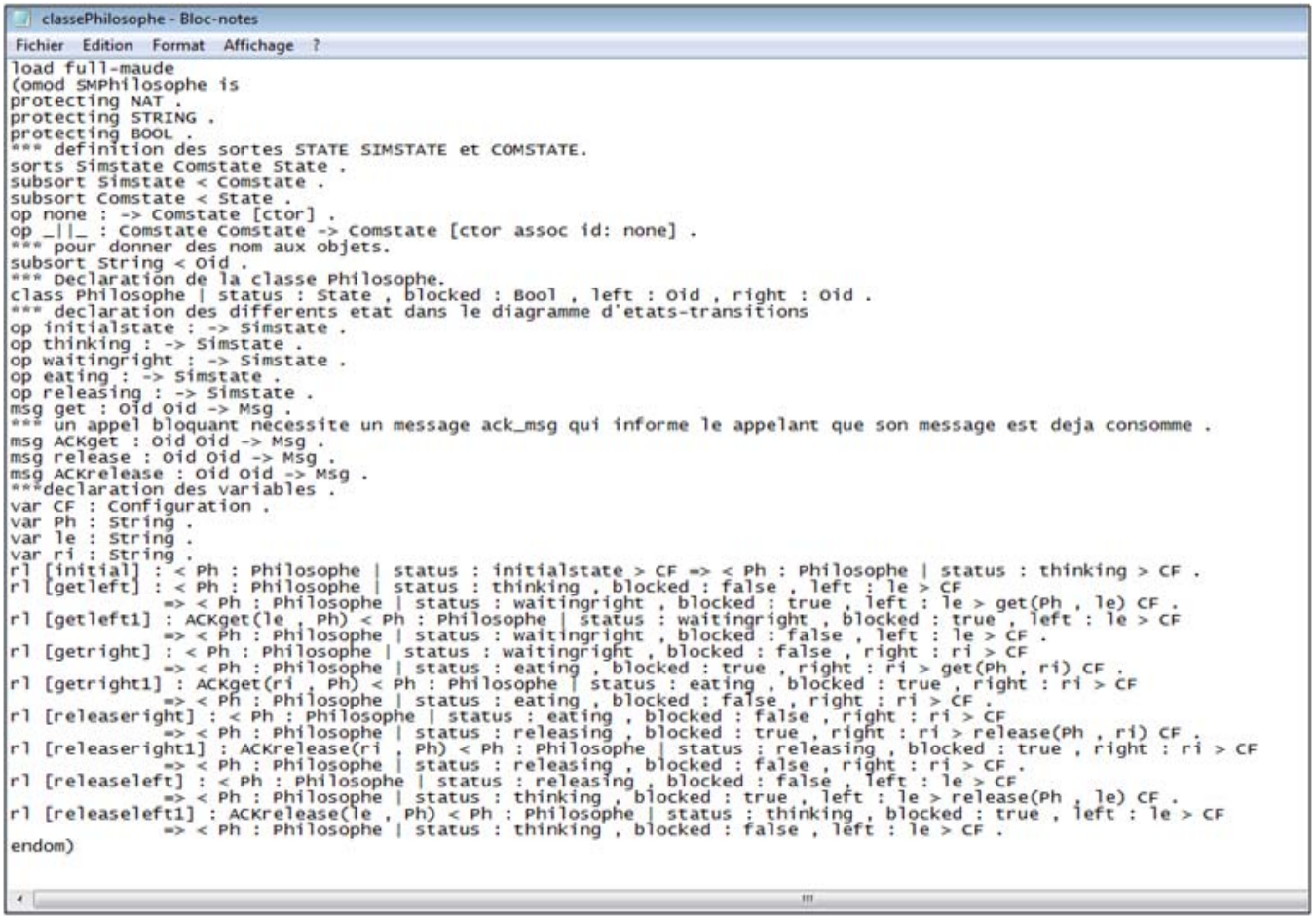

Figure 8. Generated Maude specification of Class Philosopher with its own StateChart

These rules are applied to select respectively an initial state, a final state and a simple state (not previously processed) to generate the corresponding Maude code.

• Rule 10 DecEven (Priority 10)
Is applied to locate an event not previously processed, and generate the appropriate Maude code.

• Rule 11 DecVar (Priority 11)
Is applied to declare all the variables used in rewrite rules.

• Rules 12, 13 Init_rule, Tran_rule (Priority resp 12, 13)
These rules are applied to mark the transition as visited, and generate the corresponding Maude specification.

Note that each transition in the StateChart specified by an appropriate rewrite rule (rewriting rules are perfectly adequate to describe the changes between states) [9].

• Rule 14 FinModule (Priority 14)
Is applied to mark the end of the object-oriented module in the files.

• Rule 15 DiagComm (Priority 15)
Is applied to locate a Communication diagram not previously processed (Vdiag = = 0) and create a new file include all the object-oriented modules.

• Rules 16, 17 initialsitP, initialsitF (Priority resp 16, 17)
These rules are applied to select a collaboration (not previously processed Com = = 0) and generate its equivalent Maude code.

• Rule 18 FModInit (Priority 18)

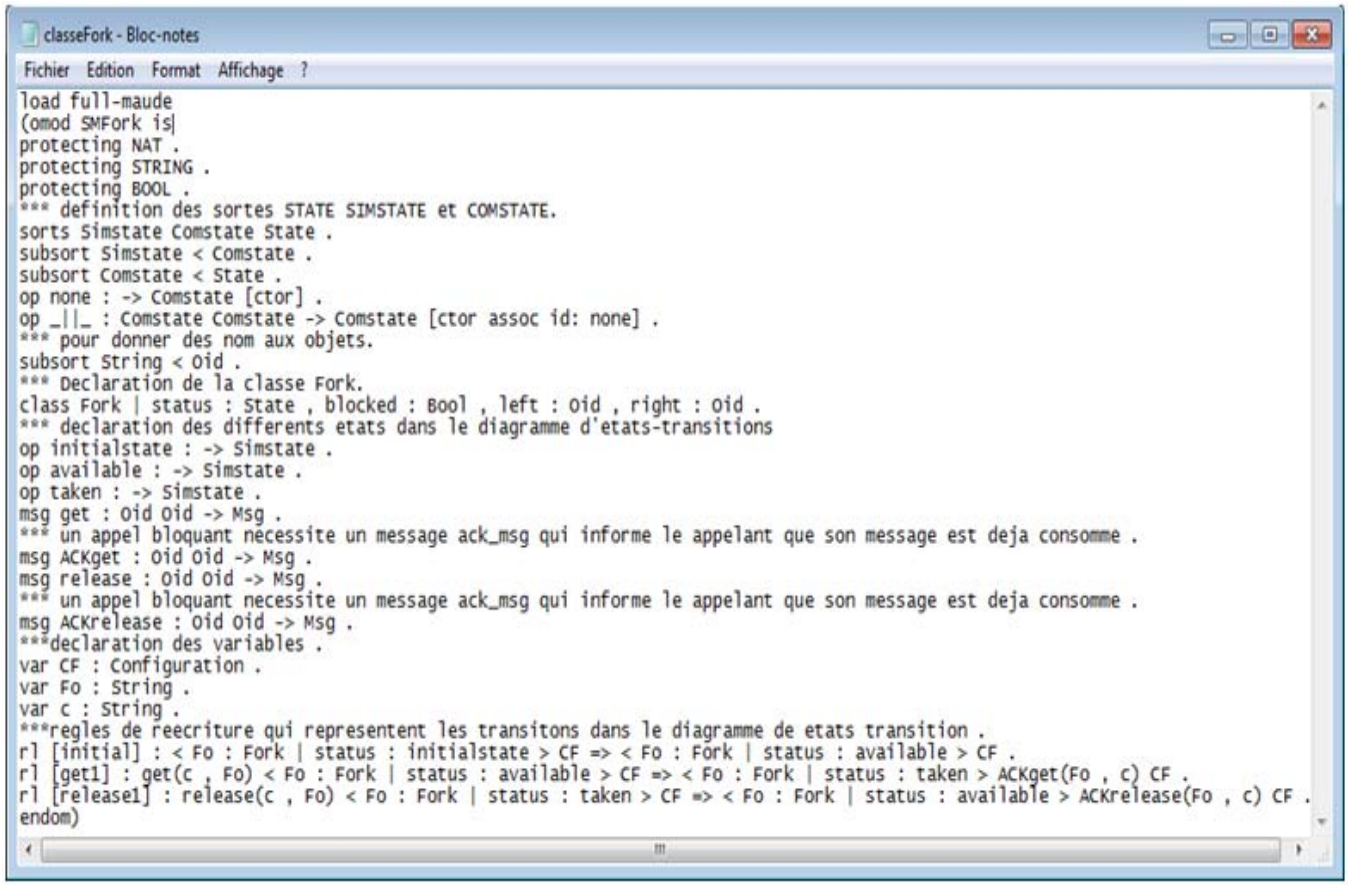

Figure 9. Generated Maude specification of Class Fork with its own StateChart

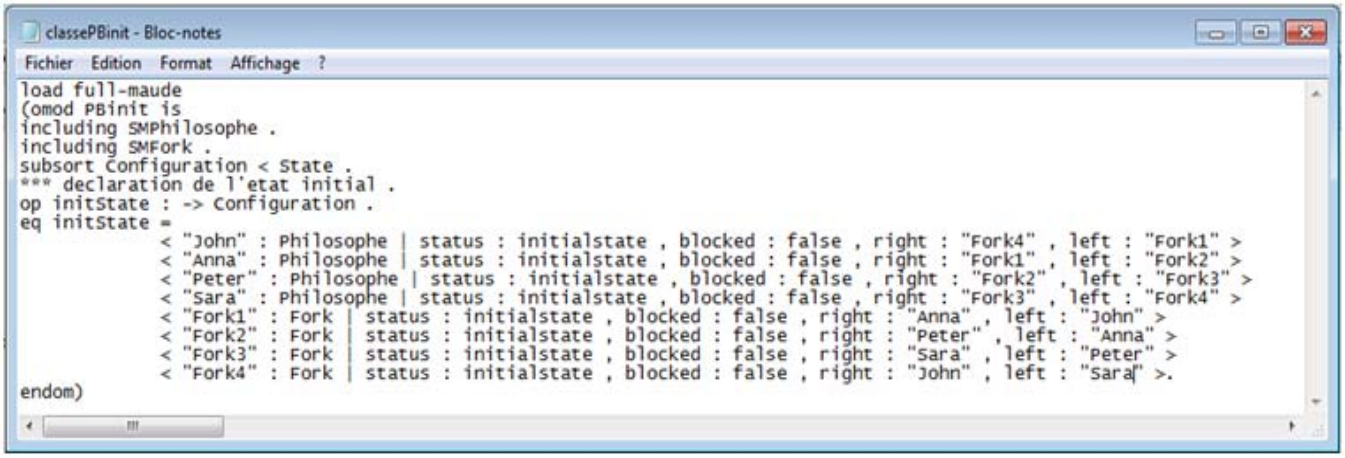

Figure 10. Generated Maude specification of dining philosophers problem

Is applied to mark the end of the object-oriented module that is related to the Communication diagram.

Our graph grammar has also a final action which erases all the global variables.

6. Case Study

To illustrate our approach, we applied it on the example of dining philosophers problem used in [6]. We propose four philosophers doing one of the two things: eating or thinking, Figure 7 presents UML models that represent this problem.

To translate this graphical representation into its equivalent Maude code in our framework, we have just click on the "*Generate SpMaude*" button that allows executing our graph grammar defined in the previous section. The result of the automatic generated files is shown in Figure 8, Figure 9, and Figure 10.

7. Conclusion and Future Work

In this paper, we have proposed an approach and a visual modeling tool. This approach takes the applications modeled in UML language and translates them into a rewriting system expressed in Maude language. To achieve this transformation, we have used UML Class diagram formalism as meta-formalism and proposed three meta-models for the UML input models; we have also proposed a graph grammar to generate Maude code in a graphical way. The meta-modeling tool AToM^3 is used. In a future work, we plan to include the verification phase using the Maude LTL Model Checker and to give a feed back of the results.

References

[1] Laurent, A. (2009). UML 2 de l'apprentissage à la pratique (cours et exercices).

[2] AToM^3 Home page, version 3.00, http://atom3.cs.mcgill.ca.

[3] Kerkouche, E., Chaoui, A. (2009). A formal framework and a tool for the specification and analysis of G-Nets models based on graph transformation, International Conference on Distributed Computing and Networking ICDCN'09, LNCS, 5408, p. 206–211, Springer-Verlag Berlin Heidelberg India.

[4] Kerkouche, E., Chaoui, A., Bourennane, E., Labbani, O. (2010). A UML and colored petri nets integrated modeling and analysis approach using graph transformation, *Journal of Object Technology*, published by ETH Zurich, Chair of Software Engineering, © JOT.

[5] Rozengerg, G. (1999). Handbook of Graph Grammar and computing Graph Transformation, *World Scientific*.

[6] Lilius, J., Porres Paltor, I. (1999). UML: A tool for verifying UML models, *In*: Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), p. 255–258, IEEE Computer Society.

[7] Meseguer, J. (1992). A Logical Theory of Concurrent Objects and its Realization in the Maude Language, G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, p. 314–390.

[8] Clavel, M., Duran, F., Eker, S., P., Lincoln, N. MartiOliet, Meseguer, J., Talcott, C. Maude Manual (version 2.4). SRI International, http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf

[9] Tibermacine, O. (2009). UML et Model Checking, Master thesis supervised by Professor A. Chaoui, University El Hadj Lakhdar Batna, Algeria. (in Frensh).

[10] Gagnon, P., Mokhati, F., Badri, M. (2008). Applying model checking to concurrent UML models, *Journal of Object Technology*, 7 (1) 59–84, January.