A Modeling and Verification Approach Based on Graph Transformation

Wafa Chama, Raida Elmansouri, and Allaoua Chaoui

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 these 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.

Index Terms—AToM3 tool, graph grammar, maude specification, meta-model.

I. INTRODUCTION

UML (Unified Modeling Language) is a graphical modeling language used to specify, visualize, 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 AToM3 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 AToM3 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.

II. RELATED WORK

Reference [1], the authors presented some rules for mapping UML diagrams to their equivalent Maude specifications. The translation is made manually. In [2], 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 [2] and graph grammars.

III. REWRITING LOGIC AND MAUDE

Rewriting logic [3] has been introduced by José Meseguer allowing concurrent software specification and verification. It is implemented by several languages such as Maude [4].

Maude is a specification and programming language. It is simple, expressive and has a high-performance implementation. Maude [4] 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 “R : [t] → [t’] if C,” which indicates that, according to rule R, term t rewrites to t’ if a certain condition 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.

IV. AToM3 TOOL

AToM3 [5] is a visual tool used for multi-formalism modeling and meta-modeling. The two main tasks of AToM3 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 [6]; each having graphs in their left hand side (LHS) and right hand side (RHS). For more details the reader is referred to [7].

V. THE PROPOSED APPROACH

The steps of our proposed approach are as follows:

A. 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.

UML Class Diagram Meta-Model: A Class Diagram [8] is a type of static structure diagram. It represents [9] 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 Fig. 1.) is composed of the following classes:

1) **ClassDiagram**: This class has a name and represents a Class Diagram.
2) **Class_simple**: This class describes the classes and has three attributes, namely class_attribut, class_name and class_op.
3) **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).
4) **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.
5) **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.
6) **Composition**: This class describes a composition, has two attributes com_name and card.
7) **Aggregation**: This class represents an aggregation. It inherits from Composition all its attributes, multiplicities and associations.
8) **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 Fig. 1), they are drawn as invisible links. And from this meta-model we generate a tool to manipulate the Class Diagram as shown in the tool bar of Fig. 2.

UML Statechart Diagram Meta-Model: A StateChart diagram [8]–[9] is used to model the behavior of a system, contains states and other types of transitions (events and actions); states may also contain subdiagrams called Composite states which can be sequential or concurrent. StateChart 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 StateChart diagrams (see Fig. 3.) is composed [10] of the following classes:

1) **StateChart**: This class has an attribute Name and represents a State Machine in the diagram.
2) **SC_Initail**: This class marks the initial state of a statechart diagram or the initial state of a composite state.
3) **SC_Final:** This class marks the final state of a statechart diagram.

4) **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).

5) **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 Fig. 3).

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

![Fig. 4. A tool for statechart diagram generated using AToM³.](image)

**UML Communication Diagram Meta-Model:** A UML Communication diagram [9] 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:

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

2) **Collaboration:** This class represents an object interacted, and has one attribute *name_coll*; and one association named *relationcoll* for representing messages as shown in Fig. 5.

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

![Fig. 5. Communication diagram meta-model.](image)

**B. 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).

**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 [2].

**Rule 3 VStateChart (Priority 3):** 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 [2] to represent states, we declared an algebraic structure:

- `sorts SIMSTATE COMSTATE STATE .`
- `subsort SIMSTATE < COMSTATE .`
- `subsort COMSTATE < STATE .`

**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 [2] that to define a class, we can use the following syntax:

- `Class class_name | Status : STATE, attr1 : sort_attr1, … attrn : sort_attrn, asso_name : Oid .`

**Rules 7, 8, 9 EtatInit, EtatFin, EtatSimple (Priority resp 7, 8, 9):** 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) [2].

**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): 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.

VI. CASE STUDY

To illustrate our approach, we applied it on the example of dining philosophers problem used in [11]. We propose four philosophers doing one of the two things: eating or thinking. Fig. 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 (see Fig. 7) that allows executing our graph grammar defined in the previous section. The result of the automatic generated files is shown in Fig. 8, Fig. 9, and Fig. 10.

VII. 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 AToM³ is used. In a future work, we plan to include the verification phase using the Maude LTL Model Checker and to give a feedback of the results.

REFERENCES


Wafa Chama was born in Constantine in 1986. She is a Doctorate's degree student at the laboratory MISC (Manipulation et Implémentation des Systèmes Critique) University Montouri2 Constantine, department of computer science, Faculty of Engineering, Algeria. She received her Master degree in 2009 from the University of Constantine. Her field of study is Software Engineering, and more specifically Graph Transformation. She can be reached at wafachama@gmail.com

Raida Elmansouri is an assistant professor with the department of computer science, Faculty of Engineering, University Montouri2 Constantine, Algeria. She received her Master degree in Computer science in 1997 and her PhD degree in 2009 from the University of Constantine. Her field of interest includes information systems and formal methods. She can be reached at raidaelmansouri@yahoo.fr

Allaoua Chaoui is a professor at the laboratory MISC (Manipulation et Implémentation des Systèmes Critique) University Montouri2 Constantine, department of computer science, Faculty of Engineering, Algeria. He received his Master degree in Computer science in 1992 (in cooperation with the University of Glasgow, Scotland) and his PhD degree in 1998 from the University of Constantine (in cooperation with the CEDRIC Laboratory of CNAM in Paris, France). He has served as associate professor in Philadelphia University in Jordan for five years and University Montouri2 Constantine for many years. During his career he has designed and taught courses in Software Engineering and Formal Methods. He has published many articles in International Journals and Conferences. He supervises many Master and PhD students. His research interests include Mobile Computing, formal specification and verification of distributed systems, and graph transformation systems. He can be reached at a_chaoui2001@yahoo.com.