MODEL CHECKING AND CODE GENERATION FOR UML DIAGRAMS USING GRAPH TRANSFORMATION

Wafa Chama[1], Raida Elmansouri[2] and Allaoua Chaoui[3]

MISC Laboratory, University Mentouri2 Constantine, Algeria
[1]wafachama@gmail.com, [2]raidaelmansouri@yahoo.fr,
[3]a_chaoui@yahoo.com

ABSTRACT

UML is considered as the standard for object-oriented modelling language adopted by the Object Management Group. However, UML has been criticized due to the lack of formal semantics and the ambiguity of its models. In other hands, UML models can be mathematically verified and checked by using its equivalent formal representation. So, in this paper, we propose an approach and a tool based on graph transformation to perform an automatic mapping for verification purposes. This transformation aims to bridge the gap between informal and formal notations and allows a formal verification of concurrent UML models using Maude language. We consider both static (Class Diagram) and dynamic (StateChart and Communication Diagrams) features of concurrent object-oriented system. Then, we use Maude LTL Model Checker to verify the formal model obtained (Automatic Code Generation Maude). The meta-modelling AToM^3 tool is used. A case study is presented to illustrate our approach.

KEYWORDS

UML, Meta-modelling, Graph Grammar, AToM^3 tool, Rewriting System, Maude Specification.

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 the UML Diagrams we consider. Section 4 presents rewriting system, Maude language and its own LTL Model Checker. In section 5 we give a brief introduction of the AToM^3 tool. Section 6 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, and how Maude's model checker

DOI : 10.5121/ijsea.2012.3604

can be used to verify objects interactions; while Section 7 describes a case study in order to illustrate our translation approach. Finally, we give a conclusion and some perspectives in section 8.

2. RELATED WORK

In [4], the authors presented some rules for mapping UML diagrams to their equivalent Maude specifications. The translation is made *manually*. In [10], the author presented another approach for transforming UML diagrams to their equivalent Maude specifications. The translation is also made *manually*. In another hand, [5] presented a formal framework (a tool) based on the combined use of Meta-Modeling and Graph Grammars for the specification and the analysis of complex software systems using G-Nets formalism. Their framework allows a developer to draw a G-Nets model and transform it into its equivalent PrT-nets model automatically. In order to perform the analysis using PROD analyzer, their framework allows a developer to translate automatically each resulted PrT-Nets model into PROD's net description language. In [6] the authors proposed an approach for transforming UML Statechart and collaboration diagrams to equivalent Colored Petri nets models.

Some works studied the application of model checking techniques for verifying Statechart and Communication Diagrams. We found in [13], a prototype tool, HUGO, supporting verification of the objects interactions with the use of PROMELA Language, Büchi automata, and SPIN model checker. We can also cite the works in [14] and [15] in which, authors implemented MCC+, a UML model consistency checker, built as a plug-in for Poseidon for UML. 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 AToM^3 and graph grammars. Our formal description was validated by using Maude's LTL model checker. Our approach is inspired from the work presented in [10] and graph grammars.

3. UML DIAGRAMS

UML 2.2 contains fourteen diagrams, divided into two categories. Seven diagrams represent the *structure* of a system while the other seven diagrams represent the *behavior of a system.* In this paper we are concerned with three diagrams: Class diagram, Statechart diagram and communication diagram.

A class Diagram [7] is a type of static structure diagram. It represents [2] 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.

A StateChart diagram [7] [2] 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.

A UML Communication diagram, known as Collaboration diagram in the previous versions [2], is a type of interaction diagrams which describes the dynamic behavior of a system; it models the interactions between objects by sending messages. The message sent between two objects can be sequential (messages having a sequence number incremented 1, 2, ….), concurrent (two messages with the same sequence number, differentiated by an added name 1a, 1b, ……..), or they can be at the same time.

4. REWRITING LOGIC, MAUDE AND LTL MODEL CHECKER

Rewriting logic [8] introduced by José Meseguer allows concurrent software specification and verification. It is implemented by several languages such as Maude [3].

Maude is a specification and programming language. It is simple, expressive and has a high-performance implementation. Maude [3] contains three types of modules: Functional modules, System modules and Object-Oriented modules.

Functional modules allow 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 rules which are introduced in *System modules*, these rules 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.

Like SPIN, the Maude LTL model checker is a support of on-the-fly explicit-state model checking of concurrent systems. So, the range of applications amenable to model checking analysis is greatly expanded.

5. ATOM^3 TOOL

AToM^3 [1] 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 [5]; each having graphs in their left hand side (LHS) and right hand side (RHS) (see Figure 1). For more details the reader is referred to [9].

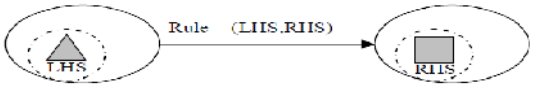


Figure 1. Rule-based Modification of Graphs

6. THE PROPOSED MAPPING AND VERIFICATION APPROACH

Our proposed Approach is performed in two steps. The first one deals with the automatic transformation of concurrent UML diagrams to their equivalent Maude formal specifications, and the second with the verification of the Communication Diagrams. These steps are described in the following (see Figure 2).

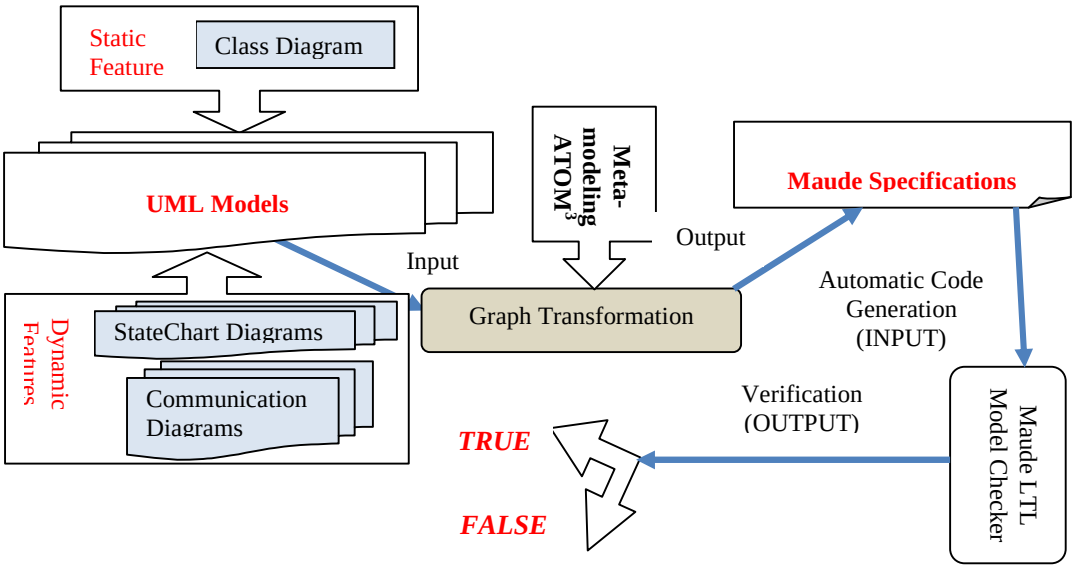


Figure 2. Overview of our Approach

6.1. Translation of UML Diagrams into Maude

In order to achieve this translation, we use the meta-modelling AToM^3 tool. We have defined three Meta-models associated respectively to the Class Diagram, StateChart Diagrams, and Communication Diagrams models. Theses Meta-models are represented by UML Class Diagram formalism and the constraints are expressed using Python code (see Figure 3).

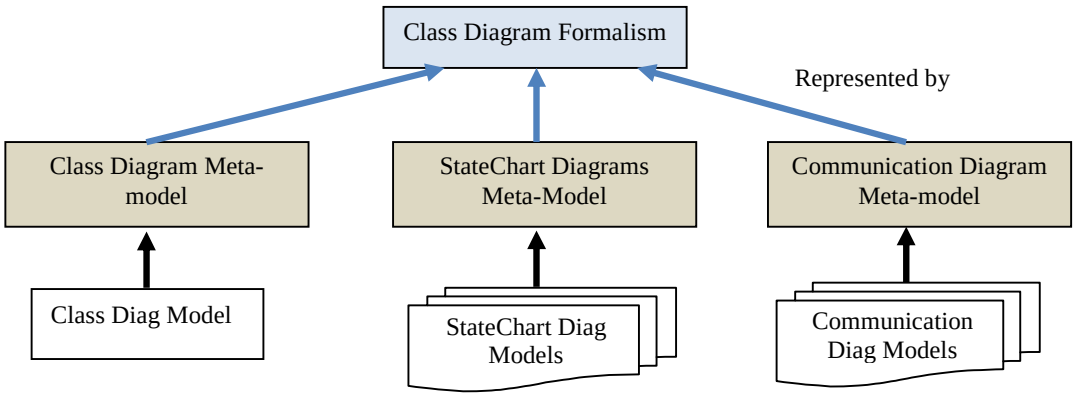


Figure 3. The Input of our Framework

6.1.1. UML Class Diagram Meta-Model

Following the description of UML Class Diagram given in section 3, we have proposed a Meta-Model Class Diagram [11] with eight Classes, seven Associations, and three inheritance associations as shown in Figure 4.

ClassDiagram: This class has a *name* and represents a Class Diagram;

Class_simple: This class describes the classes and has three attributes, namely *class_attribut*, *class_name* and *class_op*;

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);

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;*

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;*

Composition: This class describes a composition, has two attributes *com_name* and *card*;

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

Heritage "inheritance": This class represents a generalization relationship (is also known as the inheritance or "is a" relationship).

The associations are drawn as invisible links (see Figure 5).

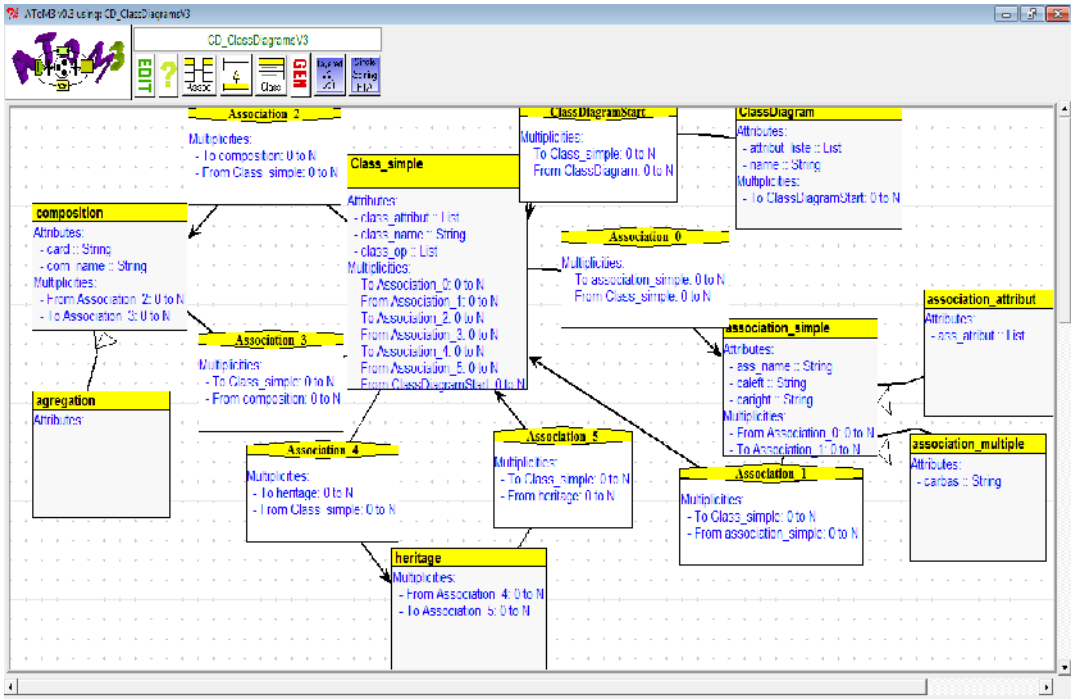


Figure 4. Class Diagram Meta-Model

Based on the Meta-model of Figure 4, we have generated using AToM^3 a tool for Class Diagram as shown in the tool bar of Figure 5.

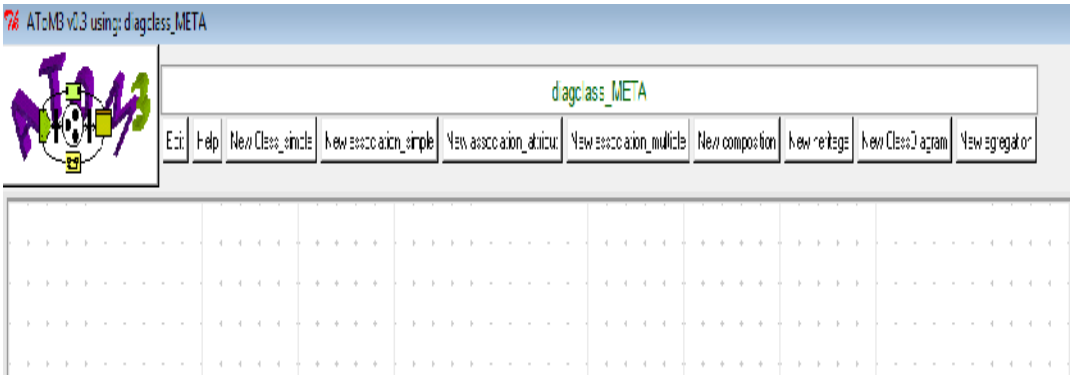


Figure 5. A tool for Class Diagram generated using AToM^3

6.1.2. UML StateChart Diagram Meta-Model

We have ameliorated our StateChart Diagram Meta-Model proposed in [11]. We took into account the advanced concepts of statechart diagram such as concurrent, sequential states, and transitions complex using join, fork and choice. So, we added three classes SC_Point_junction, SC_Point_decision, and Trans_Complex associated respectively to Junction Points, Decision Points, and complex transitions (see Figure 6).

Our Meta-Model is composed of the following classes:

StateChart: This class has an attribute *Name* and represents a State Machine in the diagram;

SC_Initail: This class marks the initial state of a statechart diagram or the initial state of a composite state;

SC_Final: This class marks the final state of a statechart diagram;

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);

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

SC_Point_jonction: This class describes Junction points (junction points are an artifact graph "a pseudo-state" which allows sharing of transition segments).

SC_Point_decision: This class represents Decision points (decision point allows choice, it has an input and at least two outputs).

Trans_Complex: This class marks complex transitions.

Associations are also included in the meta-model to allow the connections between the differences classes (see Figure 6).

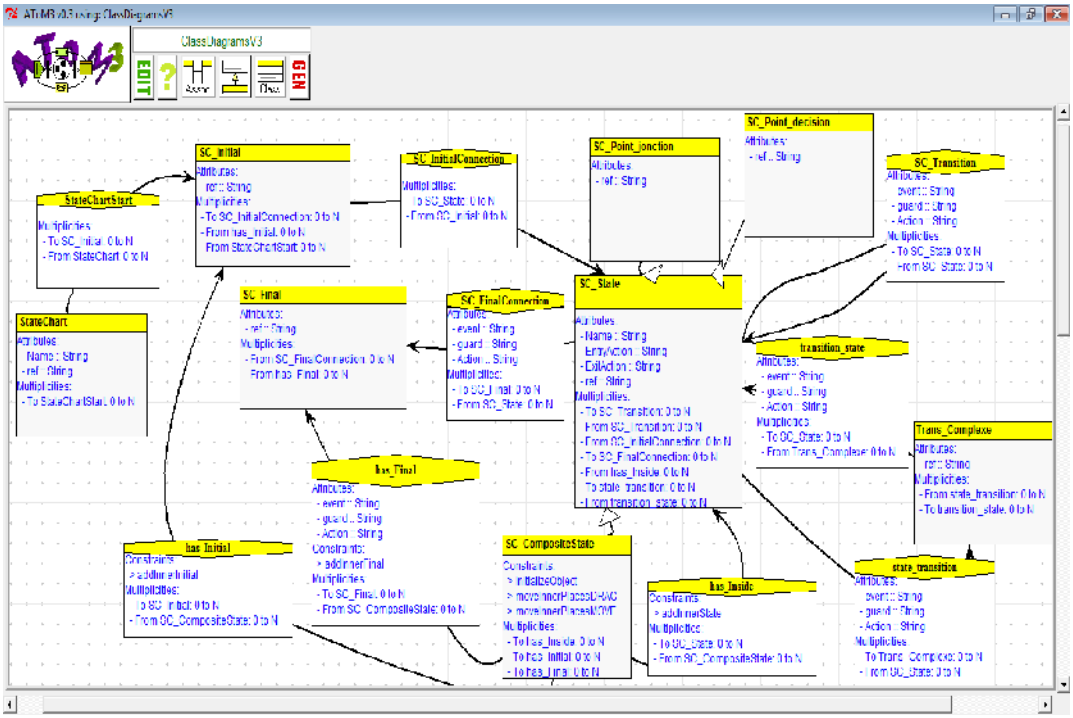


Figure 6. StateChart Diagram Meta-Model

And from this meta-model we generate using ATOM^3 a tool to manipulate the StateChart diagram as shown in the tool bar of Figure 7.

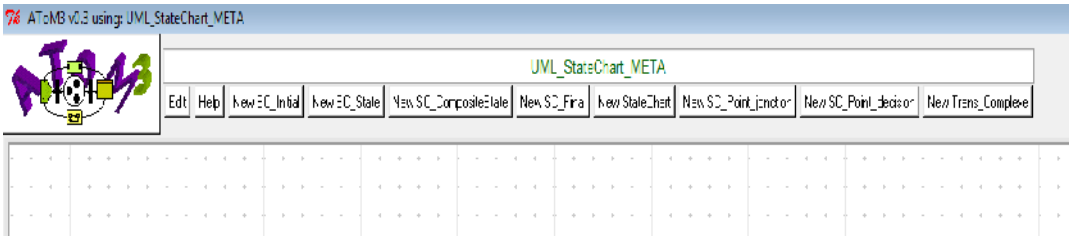


Figure 7. A tool for StateChart Diagram generated using AToM^3

6.1.3. UML Communication Diagram Meta-Model

Communication Diagram models the interactions between objects by sending messages. So, our Meta-model is composed mainly of two classes as chown in Figure 8.

CommunicationDiagram: This class has a *com_name* and represents Communication Diagram;
Collaboration: This class represents an object interacted, and has one attribute *name_coll;* and one association named *relationcoll* for representing messages (see Figure 8).

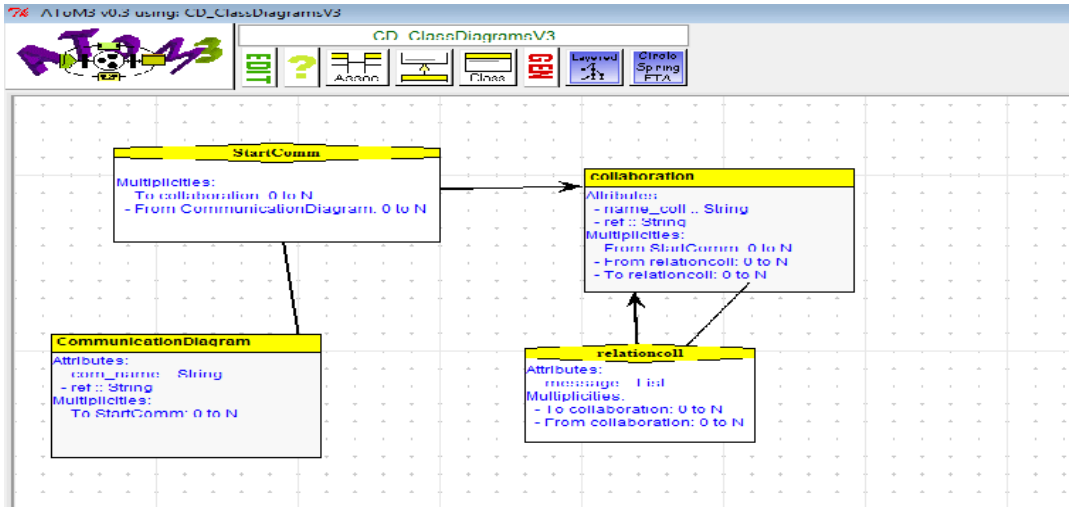


Figure 8. Communication Diagram Meta-Model

And from this meta-model we generate a tool to manipulate the Communication diagram as shown in the tool bar of Figure 9.

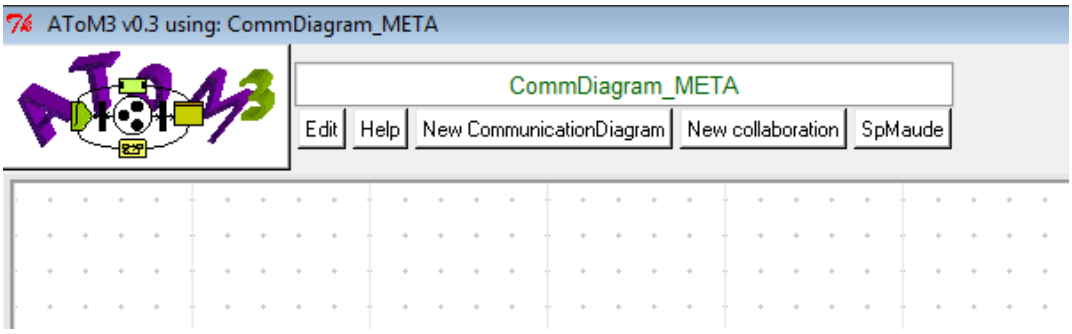


Figure 9. A tool for Communication Diagram generated using AToM^3

6.1.3. Generation of Maude Specifications

The framework which we have obtained in the previous section by means of Meta-Modelling only allows the user to create, load and save UML models. In this section, we have proposed a graph grammar containing thirty-two rules, which will be applied in ascending order by the rewriting system until no more rules are applicable. Note that each rule has a priority. None of these rules will change the input UML models because we are concerned by the code generation (Maude Specifications). We have represented only some rules in Figure 10, and described as follows:

- *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 [10].

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

Note that [10] to represent states, we declared an algebraic structure:
sorts SIMSTATE COMSTATE STATE .
subsort SIMSTATE < COMSTATE .
subsort COMSTATE < STATE .
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 associated to classes, and marks the association as visited (Asso.Visit = 1).

Note that to define a class [10], we can use the following syntax:
Class class_name | Status : STATE, $attr_1$: sort_$attr_1$… $attr_n$: sort_$attr_n$, asso_name : Oid.
(**$attr_1$:** Attribute of a class, **sort_$attr_1$:** a type of the defined attribute.)

- *Rules 7 PreSeqEvents (Priority 7):* Is executed to generate the appropriate Maude syntax (presents a sequence of events in a Collaboration in files related to classes).

Note [10] that to define a sequence of events in a collaboration, we can use the following syntax:

```
sort seq col .
subsort Msg < seq .
op null : -> seq [ctor] .
op _-_ : seq seq -> seq [ctor assoc id: null ] .
op collaboration : seq -> col .
subsort col < Configuration.
```

seq : is a list that represents the sequence of events. This structure is not commutative in order to respect the order of events, and its identity element is null. So, **collaboration(seq):** is the operation which represent the Collaboration.

- *Rules 8, 9, 10 EtatInit, EtatFin, EtatSimple (Priority resp 8, 9, 10):* These rules are applied to select respectively an initial state, a final state and a simple state (not previously processed) of StateChart Diagram to generate the corresponding Maude code.
- *Rule 11 EtatComposite (Priority 11):* Is applied to locate a Composite state (sequential or concurrent) not previously processed, and generate the appropriate Maude code.
- *Rules 12 to 17 EvStaSimp, EvCompSimp, EvSimpComp, EvDecState, EvStTraCom, EvTraComSt (Priority resp 12 to 17):* These rules are applied to locate all events declared in StateChart Diagram not previously processed, and generate the appropriate Maude code.

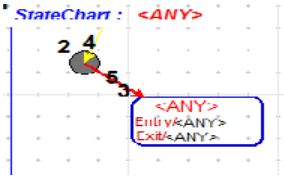
Note that [10], Events (signals, deferrals, call methods, ….) are translated to messages in Maude specification. A message in Full Maude is declared as follows:
nom_event msg: Oid Oid p, ..., pn 1 -> Msg.

p_1,....p_n : are kinds of parameters in a message, if this message represent a call of Method.

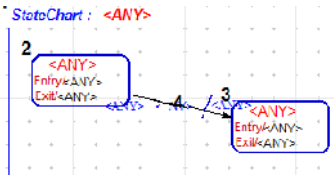
- *Rule 18 DecVar (Priority 18):* It allows to declare all the variables used in rewrite rules.

- *Rules 19* to *30 (Priority resp 19, 30):* These rules are applied to mark all transitions as visited, and generate the corresponding Maude specification. Figure 10 shows the studied transitions.
-

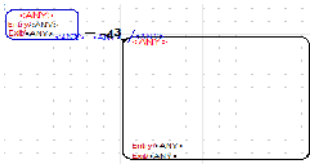
Note that [10] each transition in the StateChart specified by an appropriate rewrite rule (rewriting rules are perfectly adequate to describe the changes between states).



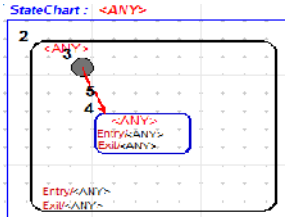
19. ruleInit: generate code Maude associated to this



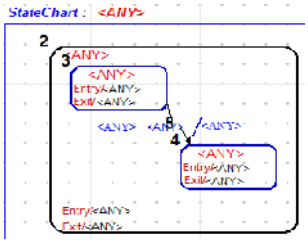
20. ruleTranSimp: generate code Maude associated to the transition between two simples states



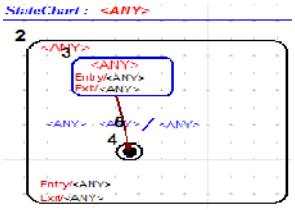
21. ruleSimpComp: generate code Maude associated to this transition



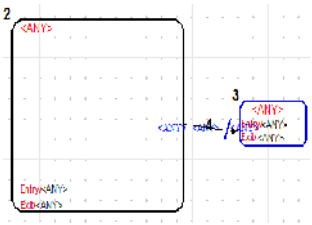
22. ruleComposite: generate code Maude associated to this transition



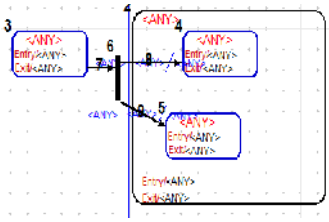
23. ruleDansComp: generate code Maude associated to transition between two simple states but in a Composite one



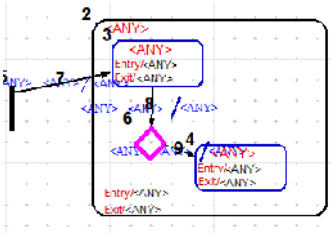
24. ruleCompfinal: generate code Maude associated to the transition in the Figure



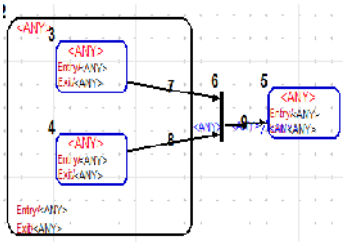
25. ruleFinalState: generate code Maude between composite and simple state



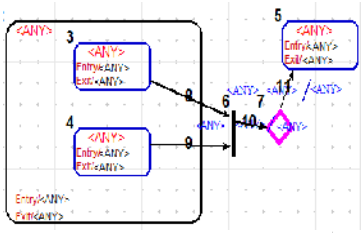
26. ruleStaConcurrent: generate code Maude for concurrent states



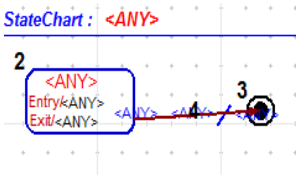
27. ruleChoice: generate code Maude associated to decision point in a composite state



28. ruleJoinState: generate code Maude associated to Join (complex transition)



29. ruleChoiceExt: generate code Maude associated to decision



30. ruleFinal: generate code Maude associated to the last transition

Figure 10. Some rules to generate the Automatic *Maude* Code from UML models

Note that, none of our rules proposed will change the input UML models because we are concerned by the code generation (Maude Specifications). So, the Left Hand Side of a rule (LHS rule1) = the Right Hand Side of the same rule (RHS rule1).

- *Rule 31 DiagComm (Priority 31):* Is applied to locate Communication Diagram and create a new file include all the object-oriented modules.
- *Rule 32 CommConfig (Priority 32):* This rule is applied to select a collaboration (not previously processed (Com = = 0) and generate its equivalent Maude code (in general, it allows detecting Configurations).

Our graph grammar contains also a final action which deletes all the global variables.

6.2. Verification of Communication Diagrams

We have named our graph grammar **UML2MAUDE**. When the UML2MAUDE's execution finished, the resulting model is Oriented-Object module Maude. So, we have used LTL Maude Model Checker to verify the resulting modules. We will check the execution of collaborations towards the internal behaviour of objects specified by the StateChart Diagrams. In other words, we can execute the system in a way which ensures that the sequence of events have been executed will go perfectly with those provided in the Communication diagram.

The following structure, allows only the execution of transitions which can perfectly produce the behavior described by the Communication Diagram. We said that, the collaboration is verified, if at the end of calculation, we have: **collaboration(null)**. If this structure is not empty, it is clear that the collaboration cannot terminate the execution (collaboration incorrect, ie the order of interactions between objects can never occur). An example is illustrated to give a clearer view on this type of verification.

Var
X : seq .
S1 S2 : state .
Cf : configuration .
Rl [*rule's_name*] : **collaboration (M_1 - X) M_1** < O : C | Status : S1 …> Cf =>
collaboration (X) < O : C | Status : S2 …> … Cf .

7. CASE STUDY

We will illustrate our approach at the hand of a simple ATM Machine example presented in [12]. This ATM Machine dispensing cash to a use, when he inserts his card and his code PIN correctly. After interred these information, the ATM transfers data to the Bank for verifying purposes. When the PIN code is wrong, the ATM demands to re-enter the code PIN. Card will be invalid after the seizure of three wrong codes PIN, and the ATM rejects the card. Figure 11 presents the ATM example in our framework.

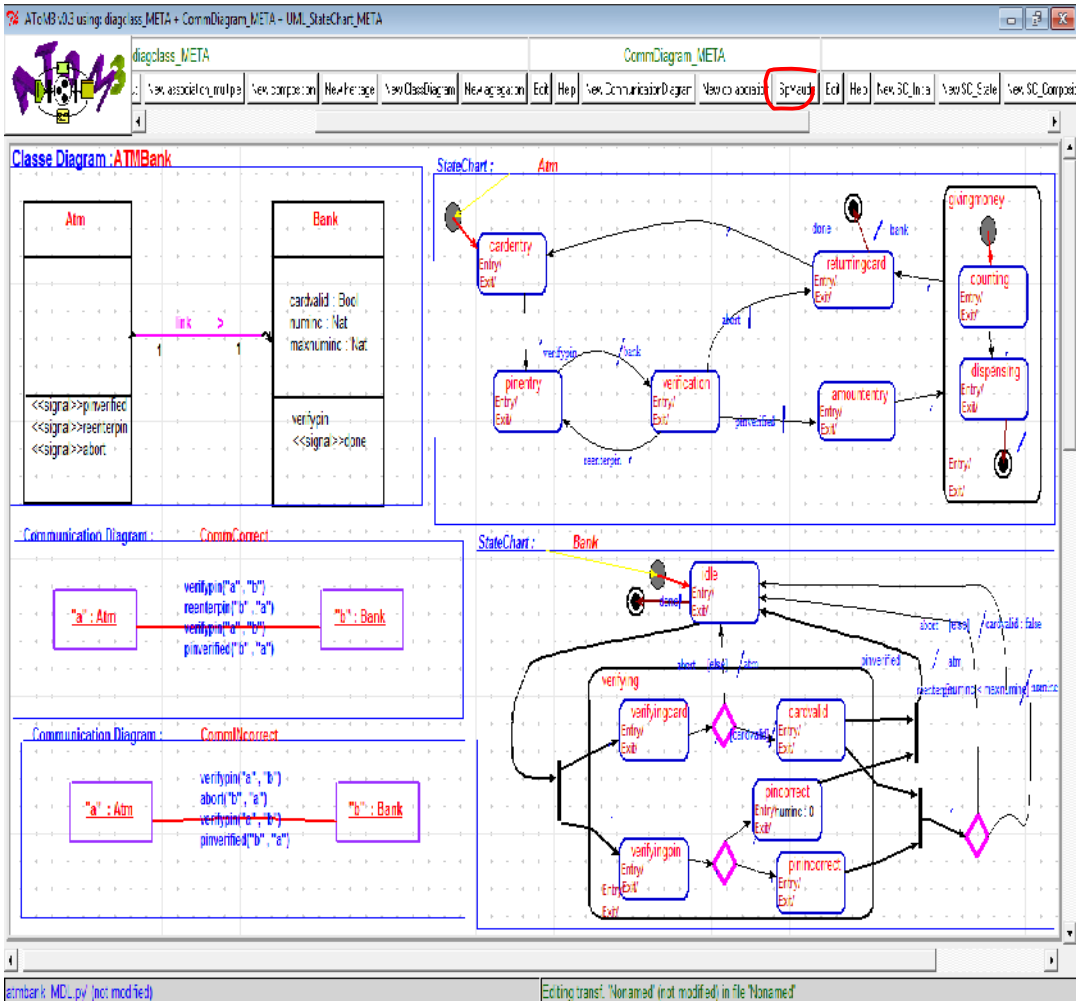


Figure 11. Example of ATM example created in our framework

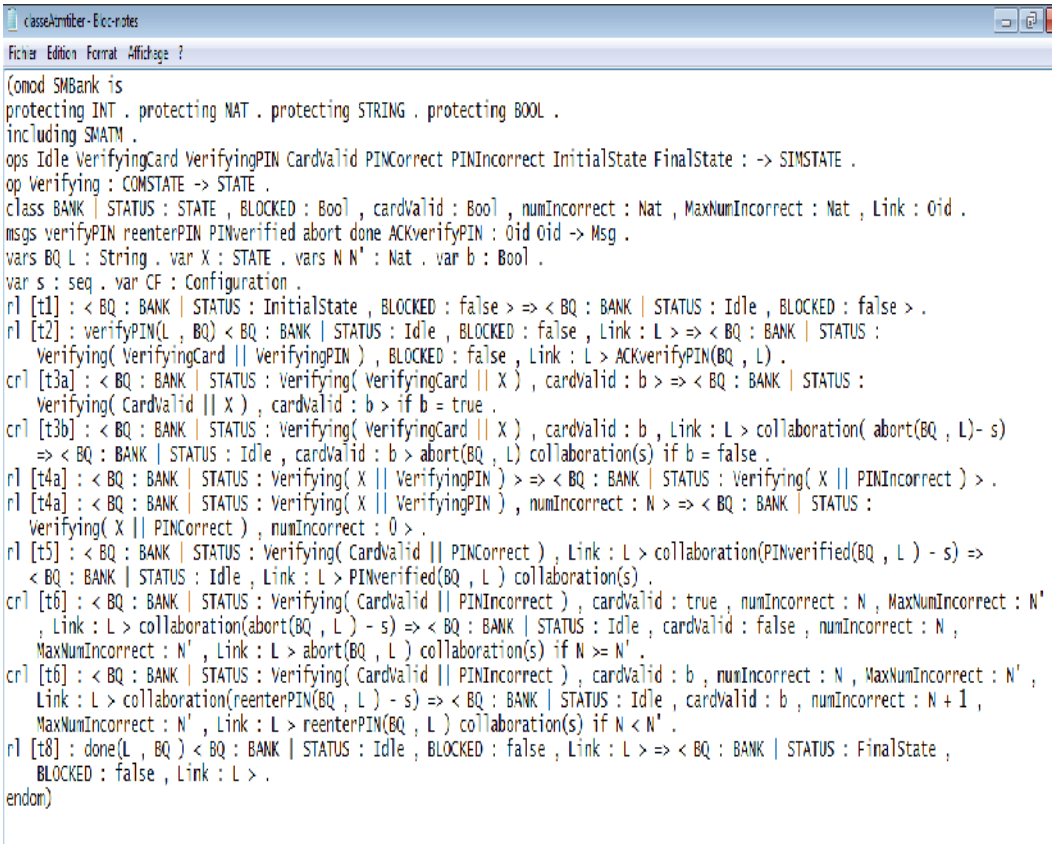
The example's developers proposed [12] two Communication diagrams, the first one (CommCorrect) specifies an expected interaction between an ATM "a" and a Bank "b". In contrast, CommINcorrect describes an undesired behaviour, when the Bank aborts a transaction, the card became invalid. So to check Communication Diagrams using Maude LTL Model Checker, we have to translate all the Diagrams associated to the ATM example into its equivalent Maude specifications. To generate Maude description in our framework, we have just to click on the ***"SpMaude"*** button in the user interface (see Figure 11). The result of the automatic generated files is shown in Figure 12, Figure 13, and Figure 14.

classeAtmt.ber - Bloc-notes

Fichier Edition Format Affichage ?

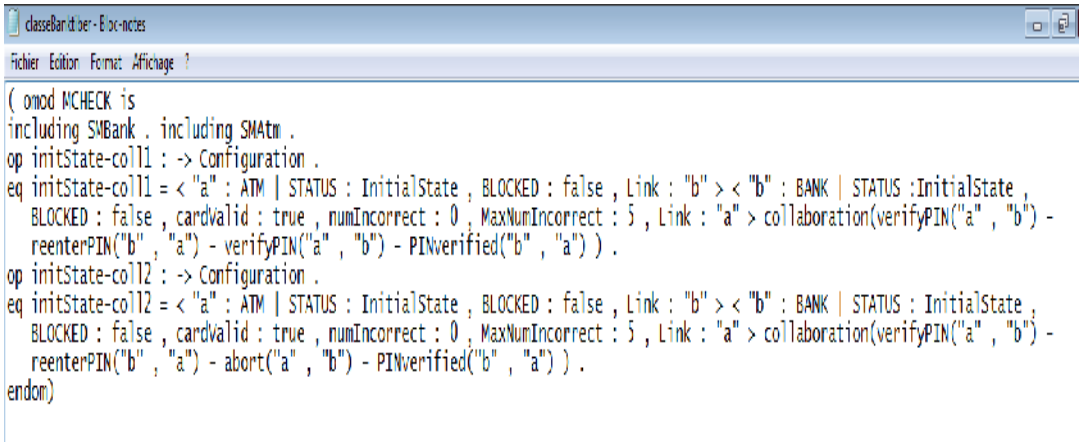
```
(omod SMATM is
protecting INT . protecting NAT . protecting STRING . protecting BOOL .
subsort String < Oid .
sorts SIMSTATE COMSTATE STATE .
subsort SIMSTATE < COMSTATE .
subsort COMSTATE < STATE .
op none : -> COMSTATE [ctor] .
op _||_ : COMSTATE COMSTATE -> COMSTATE [ctor assoc comm id: none] .
sort seq col .
subsort Msg < seq .
op null : -> seq [ctor] .
op _-_ : seq seq -> seq [ctor assoc id: null ] .
op collaboration : seq -> col .
subsort col < Configuration .
ops CardEntry PINEntry Verification AmountEntry ReturningCard Counting Dispensing InitialState FinalState : -> SIMSTATE .
op GivingMoney : COMSTATE -> STATE .
class ATM | STATUS : STATE , BLOCKED : Bool , Link : Oid .
msgs verifyPIN reenterPIN PINverified abort done ACKverifyPIN : Oid Oid -> Msg .
vars AT L : String . var X : STATE . vars N N' : Nat . var b : Bool . var s : seq .
var CF : Configuration .
rl [t1] : < AT : ATM | STATUS : InitialState > => < AT : ATM | STATUS : CardEntry > .
rl [t2] : < AT : ATM | STATUS : CardEntry > => < AT : ATM | STATUS : PINEntry > .
rl [t3] : < AT : ATM | STATUS : PINEntry , BLOCKED : false , Link : L > collaboration(verifyPIN(AT , L) - s) => < AT : ATM |
   STATUS : Verification , BLOCKED : true , Link : L > verifyPIN(AT , L) collaboration(s) .
rl [t3ack] : ACKverifyPIN(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : true , Link : L > => < AT : ATM | STATUS :
   Verification , BLOCKED : false , Link : L > .
rl [t4] : reenterPIN(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L > => < AT : ATM | STATUS : PINEntry
   , BLOCKED : false , Link : L > .
rl [t5] : PINverified(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L > => < AT : ATM | STATUS :
   AmountEntry , BLOCKED : false , Link : L > .
rl [t6] : abort(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L > => < AT : ATM | STATUS : ReturningCard
   , BLOCKED : false , Link : L > .
rl [t7] : < AT : ATM | STATUS : AmountEntry , BLOCKED : false , Link : L > => < AT : ATM | STATUS : GivingMoney(InitialState) ,
   BLOCKED : false , Link : L > .
rl [t8] : < AT : ATM | STATUS : GivingMoney(InitialState) , BLOCKED : false , Link : L > => < AT : ATM | STATUS :
   GivingMoney(Counting) , BLOCKED : false , Link : L > .
rl [t9] : < AT : ATM | STATUS : GivingMoney(Counting) , BLOCKED : false , Link : L > => < AT : ATM | STATUS :
   GivingMoney(Dispensing) , BLOCKED : false , Link : L > .
rl [t10] : < AT : ATM | STATUS : GivingMoney(Dispensing) , BLOCKED : false , Link : L > => < AT : ATM | STATUS :
   GivingMoney(FinalState) , BLOCKED : false , Link : L > .
rl [t11] : < AT : ATM | STATUS : GivingMoney(FinalState) , BLOCKED : false , Link : L > => < AT : ATM | STATUS :
   ReturningCard , BLOCKED : false , Link : L > .
rl [t12] : < AT : ATM | STATUS : ReturningCard , BLOCKED : false , Link : L > collaboration(done(AT , L ) - s) => < AT : ATM |
   STATUS : FinalState , BLOCKED : false , Link : L > done(AT , L ) collaboration(s) .
rl [t13] : < AT : ATM | STATUS : FinalState , BLOCKED : false , Link : L > CF => < AT : ATM | STATUS : FinalState , BLOCKED :
   false , Link : L > CF .
endom )
```

Figure 12. Generated Maude specification of Class ATM with its own StateChart



```
(omod SMBank is
protecting INT . protecting NAT . protecting STRING . protecting BOOL .
including SMATM .
ops Idle VerifyingCard VerifyingPIN CardValid PINCorrect PINIncorrect InitialState FinalState : -> SIMSTATE .
op Verifying : COMSTATE -> STATE .
class BANK | STATUS : STATE , BLOCKED : Bool , cardValid : Bool , numIncorrect : Nat , MaxNumIncorrect : Nat , Link : Oid .
msgs verifyPIN reenterPIN PINverified abort done ACKverifyPIN : Oid Oid -> Msg .
vars BQ L : String . var X : STATE . vars N N' : Nat . var b : Bool .
var s : seq . var CF : Configuration .
rl [t1] : < BQ : BANK | STATUS : InitialState , BLOCKED : false > => < BQ : BANK | STATUS : Idle , BLOCKED : false > .
rl [t2] : verifyPIN(L , BQ) < BQ : BANK | STATUS : Idle , BLOCKED : false , Link : L > => < BQ : BANK | STATUS :
    Verifying( VerifyingCard || VerifyingPIN ) , BLOCKED : false , Link : L > ACKverifyPIN(BQ , L) .
crl [t3a] : < BQ : BANK | STATUS : Verifying( VerifyingCard || X ) , cardValid : b > => < BQ : BANK | STATUS :
    Verifying( CardValid || X ) , cardValid : b > if b = true .
crl [t3b] : < BQ : BANK | STATUS : Verifying( VerifyingCard || X ) , cardValid : b , Link : L > collaboration( abort(BQ , L)- s)
    => < BQ : BANK | STATUS : Idle , cardValid : b > abort(BQ , L) collaboration(s) if b = false .
rl [t4a] : < BQ : BANK | STATUS : Verifying( X || VerifyingPIN ) > => < BQ : BANK | STATUS : Verifying( X || PINIncorrect ) > .
rl [t4a] : < BQ : BANK | STATUS : Verifying( X || VerifyingPIN ) , numIncorrect : N > => < BQ : BANK | STATUS :
   Verifying( X || PINCorrect ) , numIncorrect : 0 > .
rl [t5] : < BQ : BANK | STATUS : Verifying( CardValid || PINCorrect ) , Link : L > collaboration(PINverified(BQ , L ) - s) =>
   < BQ : BANK | STATUS : Idle , Link : L > PINverified(BQ , L ) collaboration(s) .
crl [t6] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) , cardValid : true , numIncorrect : N , MaxNumIncorrect : N'
    , Link : L > collaboration(abort(BQ , L ) - s) => < BQ : BANK | STATUS : Idle , cardValid : false , numIncorrect : N ,
    MaxNumIncorrect : N' , Link : L > abort(BQ , L ) collaboration(s) if N >= N' .
crl [t6] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) , cardValid : b , numIncorrect : N , MaxNumIncorrect : N' ,
    Link : L > collaboration(reenterPIN(BQ , L ) - s) => < BQ : BANK | STATUS : Idle , cardValid : b , numIncorrect : N + 1 ,
    MaxNumIncorrect : N' , Link : L > reenterPIN(BQ , L ) collaboration(s) if N < N' .
rl [t8] : done(L , BQ ) < BQ : BANK | STATUS : Idle , BLOCKED : false , Link : L > => < BQ : BANK | STATUS : FinalState ,
    BLOCKED : false , Link : L > .
endom)
```

Figure 13. Generated Maude specification of Class Bank with its own StateChart



```
( omod MCHECK is
including SMBank . including SMAtm .
op initState-coll1 : -> Configuration .
eq initState-coll1 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS :InitialState ,
   BLOCKED : false , cardValid : true , numIncorrect : 0 , MaxNumIncorrect : 3 , Link : "a" > collaboration(verifyPIN("a" , "b") -
   reenterPIN("b" , "a") - verifyPIN("a" , "b") - PINverified("b" , "a") ) .
op initState-coll2 : -> Configuration .
eq initState-coll2 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS : InitialState ,
   BLOCKED : false , cardValid : true , numIncorrect : 0 , MaxNumIncorrect : 3 , Link : "a" > collaboration(verifyPIN("a" , "b") -
   reenterPIN("b" , "a") - abort("a" , "b") - PINverified("b" , "a") ) .
endom)
```

Figure 14. Generated Maude specification of ATM example

Concerning the verification, we have just introduced the resulting modules in Maude WorkStation (see Figure 15). Then we can check the interactions between objects using Maude's LTL Model checker. So; we will see, if the two collaborations finish their execution relative to specifications of ATM and Bank objects. To have the result, we use the command **rew** of Maude (see Figure 16).

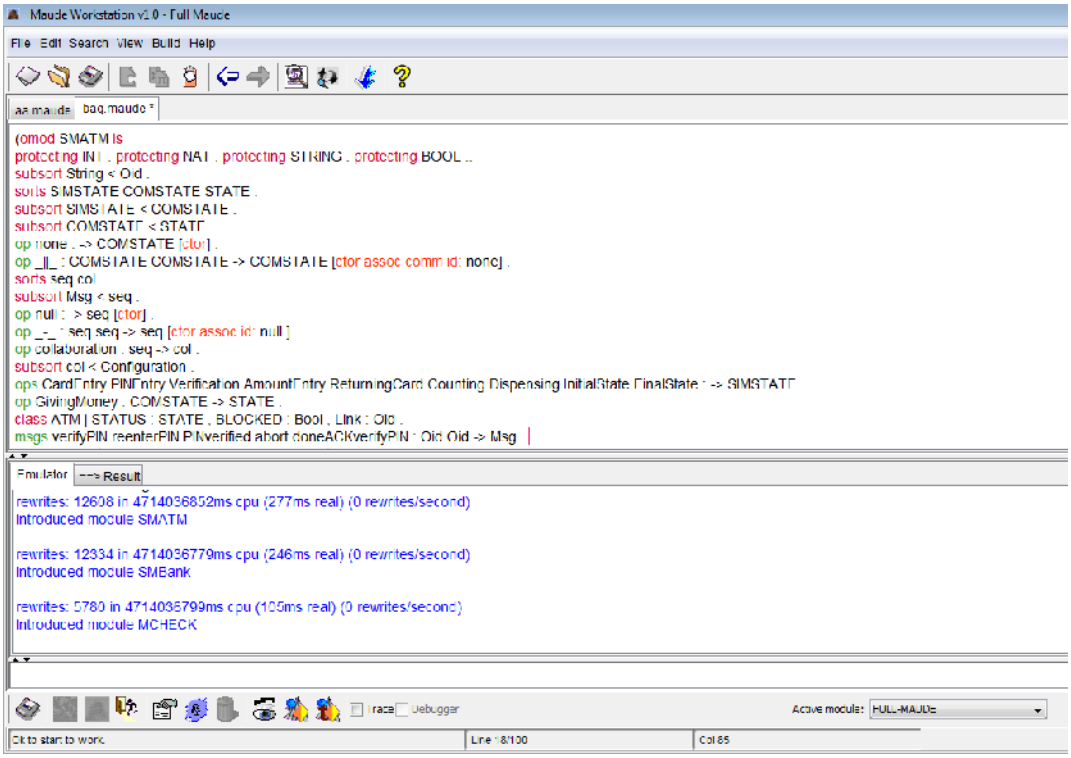


Figure 15. Resulting modules in Maude Workstation

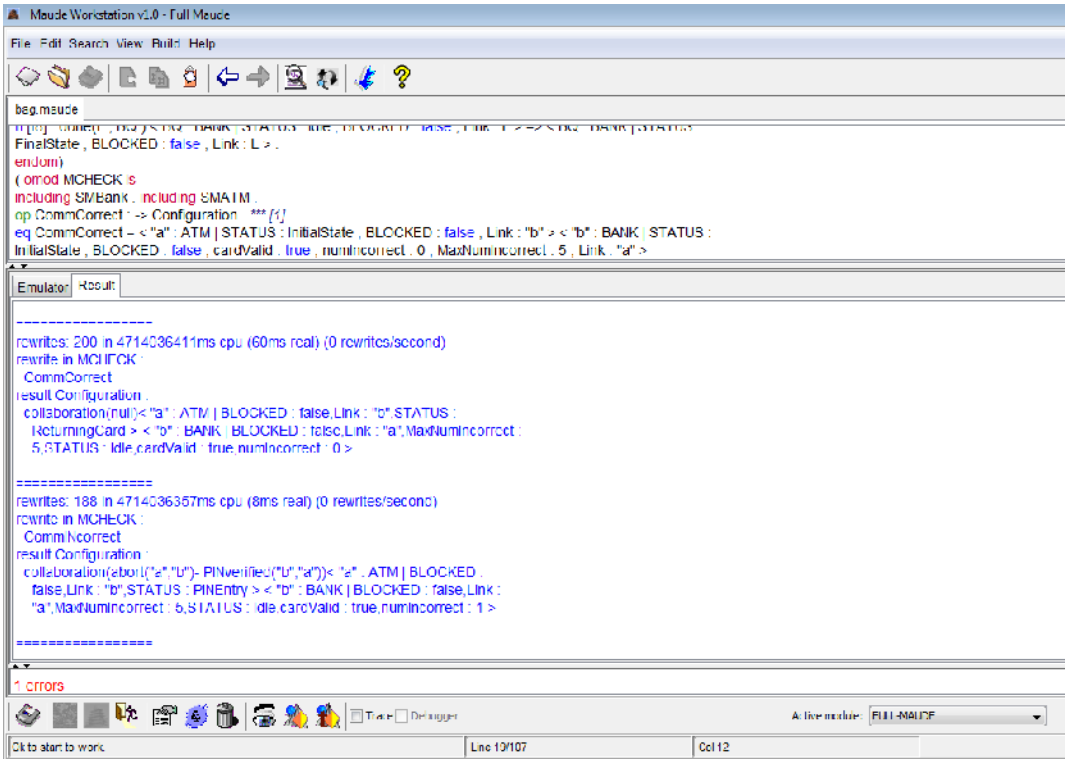


Figure 16. Verification's results

This verification (corresponding to CommCorrect diagram) gives a result of a final state with an empty list (list that represents the collaboration), which conform that the ATM and Bank objects arrive to execute the collaboration.

On the other hand, the second result confirms that the ATM and Bank objects cannot execute this one. So, the second Communication Diagram (CommINCorrect) is incorrect.

8. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach and a visual modeling tool based on the combined use of Meta-Modeling and Graph Grammars. This approach takes the applications modeled in UML language (both static and dynamic aspects) 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. Moreover, we used Maude's model checker to verify objects interactions. In a future work, we plan to include the verification of some LTL proprieties (deadlock, mutual exclusion,….) and to give a feed back of the results.

REFERENCES

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

[2] Blanc, Xavier & Isabelle, Mounier (28 Septembre 2006) UML 2 pour les développeurs cours avec exercices corrigés, Eyrolles ISBN : 2-212-12029-X

[3] Manuel, Clavel & Francisco, Duran & Steven, Eker & Patrick, Lincoln & Narciso, MartiOliet & José, Meseguer & Carolyn, Talcott, (2008) Maude Manual (version 2.4), SRI International, http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf

[4] Patrice, Gagnon & Farid, Mokhati & Mourad, Badri (January 2008) "Applying Model Checking to Concurrent UML Models," Journal of Object Technology, Vol 7, No 1, pp 59-84.

[5] Elhillali, Kerkouche & Allaoua, Chaoui (2009) "A Formal Framework and a Tool for the Specification and Analysis of G-Nets Models Based on Graph Transformation". Proc. International Conference on Distributed Computing and Networking, ICDCN 09, LNCS, Springer-Verlag Berlin Heidelberg India., Vol 5408, pp 206-211.

[6] Elhillali, Kerkouche & Allaoua, Chaoui & El-Bay, Bourennane & Ouassila, Labbani (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.

[7] Laurent, Audibert (2009) UML 2 de l'apprentissage à la pratique (cours et exercices). Editions Ellipses, ISBN 10: 2729852697.

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

[9] Rozenberg, Grzegorz (1999) Handbook of Graph Grammar and computing Graph Transformation, World Scientific.

[10] Okba, Tibermacine (2009) UML et Model Checking, Mémoire de Magister, encadré par Pr. Allaoua, Chaoui, Université El Hadj Lakhdar, Batna, Algeria.

[11] Wafa, Chama & Raida, Elmansouri & Allaoua, Chaoui (2012) "Using Graph Grammars to Generate Code Maude from UML models", Internal Report, N° : MISC/MFGL/02/2012.

[12] Alexander, Knapp & Stephan, Merz. (2002) "Model checking and code generation for UML state machines and collaborations". In Proceeding of the 5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002), Report 2002-11, University Augsburg.

[13] Timm, Schäfer & Alexander. Knapp & Stephan, Merz (23 July 2001) "Model checking UML State Machines and collaborations", In Workshop on Software Model Checking, Volume 55, Number 3, Paris, France, pages: 357-369.

[14] Jocelyn, Simmonds & Cecilia. Bastarrica & Nancy, Hitschfeld-Kahler & Sebastian, Rivas (September 2008) "A Tool Based on DL for UML Model Consistency Checking". In IJSEKE Journal (International Journal of Software Engineering and Knowledge Engineering), Volume 18, Number 6, pages: 713-735.

[15] Jocelyn, Simmonds & Cecilia, Bastarrica (2005) "Description Logics for Consistency Checking of Architectural Features in UML 2.0 Models". Technical report, Department of Computer Science, University of Chile.

Authors

Wafa Chama is a PhD student at MISC Laboratory (Manipulation et Implémentation des Systèmes Critique) University Mentouri2 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 assistant Professor with the department of computer science, Faculty of Engineering, University Mentouri 2 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 Professor at University Mentouri 2 Constantine, department of computer science, Faculty of Engineering, Algeria. He is the head of the research team MFGL in MISC Laboratory. 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 Mentouri 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 .