



EDITING AND ANALYZING AUML MODEL: A MAUDE BASED TOOL

¹NOURA BOUDIAF, ²KAMEL BENSABER, ³KARIMA SID, ⁴RADIA SAHBI

¹Assoc. Prof., Department of Computer Science, University of Oum El Bouaghi, Algeria

²Phd., Eurofunk Kappacher GmbH Company, Salzburg, Austria

³Engineer., Department of Computer Science, University of Oum El Bouaghi, Algeria

⁴Engineer., Department of Computer Science, University of Oum El Bouaghi, Algeria

E-mail: boudiafn@gmail.com, kbensaber@yahoo.de, sidkarima87@gmail.com, sahbi.r@hotmail.com

ABSTRACT

The lack of formal semantics in the existing formalisms describing multi-agents models combined with multi-agents systems complexity are sources of several problems during their development process. Formal methods are known to bring rigorous and precise descriptions. In previous papers, we have proposed a formal and generic framework called AUML-Maude allowing formal description and validation of AUML model with Maude. This language, based on rewriting logic, offers a rich notation supporting formal specification, implementation and verification of concurrent systems. In this paper, we enrich the translation AUML-Maude by new ideas and we propose a rewriting logic based tool for the edition and the analysis of AUML model. The tool allows to the user to draw AUML system graphically and translates the graphical representation to Maude specification for analysis. This tool allows preserving the graphic notations offered by AUML model for clarity and getting a formal specification in Maude for formal semantics and analysis.

Keywords: *Multi-Agents Systems, AUML Model, Formal Specification, Formal Semantics, Rewriting Logic, Maude, Graphical Interface, Edition, Analysis*

1. INTRODUCTION

The formalization of multi-agents systems (MAS) is not a very recent idea. Many approaches aiming MAS' formal specification have been proposed in the literature: graphic methods such Petri nets [2], approaches representing an adaptation of object-oriented specification methods like Lotos [9], and more recently approaches based on some kinds of logic like temporal logic [13]. In the literature, the proposed approaches aiming MAS' formal specification are often limited to some specific aspects. Several notations are often used to describe the same MAS. Such combinations constitute a serious obstacle to rigorous and founded checking of the properties of the described systems [10].

We showed in previous papers [12] the feasibility and the interest to formalize some aspects of AUML multi-agents model using Maude language. The constructions offered by this language are rich enough to capture the multiple aspects of the AUML model. Maude [11] is

considered as one of the most powerful languages in description, programming and verification of concurrent systems. Maude is a formal language based on a sound and complete logic called rewriting logic [10]. This logic allows us to reason correctly on non-deterministic concurrent systems in terms of the "true concurrency" semantics. The majority of formal methods used in the framework of formalization of MAS do not bring anything more in terms of expressivity or verification power compared to rewriting logic because they are integrated in the rewriting logic [10], [11].

In addition to its power of expression, Maude offers many possibilities of validation and verification. For validation, it supports simulation in flexible way. For verification, Maude supports Model Checking of invariants and of course the known LTL Model Checking. The Maude Model Checking of invariants that could work sometimes with even infinite systems is based on accessibility analysis by creating a part or all the reachability graph of the system. Model Checking techniques are an important issue in the field of concurrent



systems checking. Model Checking of invariants aims detecting some incoherent states by describing ‘safety properties’ (well known in the literature as something bad should never happen). Moreover, this kind of Model Checking helps us to be sure that some necessary states are really accessible from the initial one. The LTL Model Checker is more powerful and more flexible. The LTL Model Checker of Maude is designed to combine Maude and linear temporal logic (LTL) in order to benefit from the two formalisms advantages [7], [5].

The generated AUML-Maude descriptions have been validated by means of simulation and Model Checking thanks to the Maude platform. We offer to the user to re-use the obtained model AUML-Maude core. The user can model his application directly in Maude by making an import of modules implementing AUML-Maude core.

However, Maude system offers textual way to the user to create and deal with AUML model. Execution under Maude system is done by using command prompt style. In this case, we loose the graphical aspect of AUML which is important for the clarity, simplicity and readability of a preliminary system description. Moreover, AUML is a model very adapted to the requirements of MAS descriptions, which make easier the development of preliminary MAS description. However, Maude is a general language; it has not specific tools to catch easily MAS’ requirements. Also, Maude allows a detailed version from the beginning, which is not very recommended for the preliminary descriptions of systems.

The purpose of this paper is double:

- Proposing new ideas about the translation AUML-Maude; that update and complete those proposed in [12].
- An interactive tool to create and analyze MAS by using AUML notations. The tool proposed in this paper, allows the user to graphically edit an AUML system (class diagram, Template (level 1), sequence diagram (level 2) and statechart diagram (level 3)) and then converts the graphical representation to its equivalent description in Maude. Thereafter, the tool calls the Maude system for the execution (Simulation, Model Checking of Invariants or LTL Model Checking) of the obtained code and reconverts the obtained result described in Maude to a graphical representation. With the help of AUML system example, we will show the aim functionalities of the tool.

Let’s note that there is a big number of tools for the creation and the manipulation of MAS, most of these tools are implemented in the imperative

languages. Some known ones are AgentTool [6], AgentBuilder [1] and MadKit [8]. These tools are Java-based graphical development environments to help users analyze, design, and implement MAS. With regard to these tools only the preliminary specification in our application, is diagrammatically developed by using the AUML notations, the rest of the development process phases is made in Maude: formal specification, implementation and validation in terms of simulation and verification in terms of Model Checking. Our application has not yet reached the level of these tools in terms of wide offered services. For the moment, it is a preliminary version including some services to be completed in the future. But, in our knowledge, the tool presented in this paper is one of the few analysis tool rewriting logic-based for MAS. This tool allows us to benefit from the power of rewriting logic in specification and analysis of concurrent systems in the context of MAS.

The remainder of this paper is organized as follows: In section 2, we present briefly the AUML model. In the section 3, we present an example of system described by using AUML. In section 4, we give a short outline on the rewriting logic. Section 5 is a general presentation of Maude language; to its two levels: specification level and verification. In the section 6, we explain our proposed process for the formalization of AUML model by using Maude. Most important functionalities of our application are illustrated in section 7 with the help of the example. Finally, we discuss our current work and give some conclusions and future work in section 8.

2. AUML MODEL

AUML (Agent Unified Modelling Language) is a very famous Multi-agent model. AUML is an extension of the UML (AUML = Agent + UML), adapted to the analysis and the agent-oriented design. It supports all UML diagrams (analysis, design) by modifying and adapting them to describe the agents such that:

- Objects in UML are replaced by agents and roles in AUML;
- New notations in AUML to represent complex interactions between agents;
- Embedded protocols are supported by AUML;

AUML allows representing and describing interactions protocols between agents (AIP), it allows describing inter and intra-agents behaviour. AUML focuses on:

- Agents classes’ representation;



- Description of the interactions between agents.

2.1. Agents Class Diagram

This diagram is composed of some agent classes, each class is a collection of agents that play a role and they have the same behaviour.

2.1.1. Elements of Agent Class

Agent Role. A role is a characteristic of a collection of agents having same properties, interfaces, services and specific behaviour. The general form is: agent class name / role1, ..., rolen

Internal Agent State Description. Definition of instances variables that express the agent state. It is the attribute concept of the object approach which is often used. For each attribute, some characteristics (ex: visibility (public, private), ..) are defined.

Actions. It could be pro-active or reactive. Pro-active means that the agent itself which provokes the action. On the other hand, reactive means that the agent waits for another agent to provoke this action for it.

An action is defined through a signature and semantic:

Signature: visibility + name + list of parameters.

Semantic: pre-conditions + post-conditions + invariants.

Note. Pre-condition is the condition that must be checked before the execution of the method or the creation of an element and Post-Condition is the condition that must be checked after the execution of the method or the destruction of an element.

Methods. Every method is described by using pre-conditions and post-conditions.

Services. Provided services are described informally.

Exchanged Messages. Description of the sent and received messages through the specification of the protocols. An exchanged message between agent is composed by a communication acts. In fact, an agent does a local work by executing its internal behaviour (which expressed by a statechart diagram: composed by actions and states). When an agent arrived at a given state, it sends a message to the other agent (which is in communication with it), at the reception, the message invokes some actions

of the agent receiver which will continue doing its local work.

2.2. AUML Levels

A hierarchical view is adopted in AUML. The first level gives a global view, and the last level is a detailed view on intra-agent behaviour. Here, we present briefly all the levels and we leave details to the next section.

First Level. It's a representation of the global protocol. It is generally described by using sequence diagram, package or template. This level gives some details comparing to the element 'Exchanged messages' of the class diagram. But, the representation of this level will be detailed more in the next level.

Second Level. This level focuses on the different inter-agents interactions. It can be described by using sequence, collaboration, state, or activity diagrams.

Third Level. This level gives some tools to allow a detailed description of each intra-agent behaviour. An activity or statechart diagrams are often used in this level.

2.3. Detailed Description of Levels

In this section, we will present in detail each level and the tools used in it. But, we focus only on tools that we adapted to be translated to Maude language in this project.

2.3.1. First Level: Template

The template is just a global specification of the protocol by indicating the communicating agents and the messages that will be exchanged between them. The basic idea of the template is to give a reusable component that will be specialized and detailed in the second level by using for instance sequence diagram.

2.3.2. Second Level: Sequence Diagram

Actors Naming. agent class name/role1, ..., rolen

Exchanged Messages. Here, messages are composed of communication acts. In the figure 1, a part example of sequence diagram, where the Agent1 sends a message (composed of the communication act CA-1) to Agent2 which answers by sending the message CA-2.

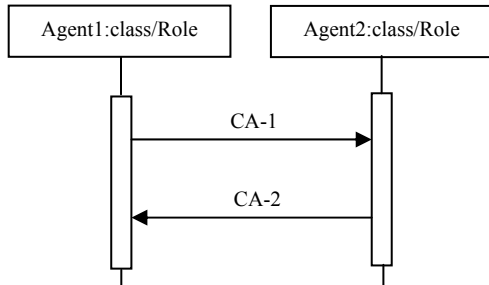


Figure 1. Basic form of agent's communication

On another hand, a message is not always simple like that; it can be composed of many communications acts that are connected to each other by some 'connectors'. In AUML, three connectors are defined to allow concurrent threads of interaction between agents (figure 2):

- 'AND' Connector: Concurrent sending of CA-1, ..., CA-n, the figure 2.a shows the notation of the 'AND' connector.
- 'OR' Connector: Concurrent sending of 0 or more CA-1, ..., CA-n. So, there is a need of mechanism to decide which CA-i sequence will be sent. The figure 2.b depicts how the 'OR' connector is represented in AUML.
- 'XOR' Connector: sending only one CA-i ($i=1, \dots, n$) at the same time. A notation of this connector is depicted the figure 2.c.

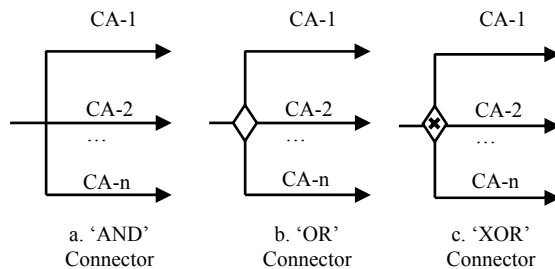


Figure 2. Concurrent threads of interaction

2.3.3. Third Level: Statechart Diagram

This diagram allows changing states of an agent during messages (communication acts) between agents. It allows expressing some constraints on protocols. Moreover, it allows a layered view on states to cope with state explosion problem. In figure 3, we find an example of statechart diagram and the coming of a communication act (CA).

When the agent is the initial state (black circle) and action Action-1 occurs, the agent changes its state to S0. But, when the agent in state S0, it invokes the execution of the method Method-1. At the end of the execution of Method-1, the agent continues changing its state to S1 if the action Action-2 appears. The appearance of an action depends on its kind if it is pro-active or reactive.

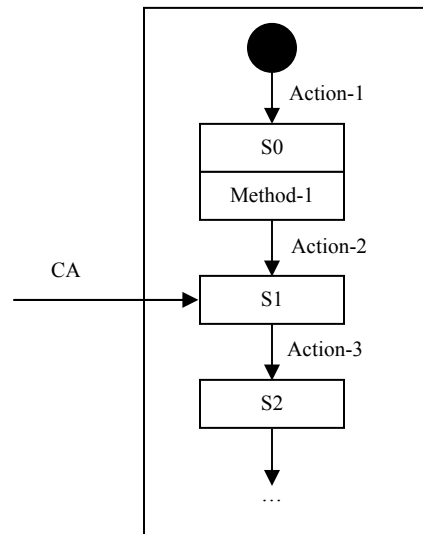


Figure 3. Typical form of statechart diagram with the arrival of communication act

3. EXAMPLE

In this section, we present a known example in the literature which is 'System of Service Information Integrated on the Mobile', (SSII), but here, we took this system and we brought some modifications to it. This example will help us explaining our tool.

3.1. System Presentation

A user sends a request to the system SSII to provide him some information on his mobile. The system responds the request of the user by looking for in its data base which is integrated with many sites web sources and selected automatically information that needs the user.

3.2. System Agents Modelling

Here, we determine the agents of the system, relationships between them and the roles of each agent. Three agents are distinguished:

Server: it communicates with the user to receive from him the demand and sends him the result. It



controls all the system and contacts the distributor agent.

Distributor: this agent creates requests that are appropriate to the capacities of the wrappers on each web site.

Wrapper: this extracts automatically necessary information; it answers directly the demand of the distributor.

3.3. Classes Modelling/Class Diagram Creation

Three classes are defined: server, distributor and wrapper that are connected via communication protocol. The figure 6 shows the class diagram system with the three classes with their protocol and cardinalities. For explanation reasons, we take server class,

- First box: the class name is server and the role is receiver;
- Second box: attributes, only one attribute (adressIP);
- Third box: Methods, two methods are defined: create-sub-demand and integrate-response-distributor();
- Fourth box: actions (reactive and proactive);
- Fifth box: communication acts. The server receives demand (on the left) and sends sub-demand (on the right).

3.4. Intra-Agents Behaviour Modelling/Statechart Diagrams Creation

The statechart diagrams of the three agent's classes are described in the figure 9. We explain here only the Server Statechart Diagram.

Server Statechart Diagram.

- First, the server agent is in InitialState;
- Then it changes its state to receivedemand after receiving initiation-state from the user;
- When a receive-demand comes (a demand form the user), it consumes this action and it changes the state to 'create-sub-demands'; here it executes its method create-sub-demand;
- Now, the agent has sub-demands;
- The agent sends the communication act 'sub-demand' to the distributor and the action send-sub-demand to it-self, so it will enter in the state wait-server until receiving the answer from the distributor;
- The server receives the communication act 'response-distributor', which invokes the action receive-response-distributor allowing it to change the state to

'integraterespondedistributor'; here the server agent execute its method 'integrate-response-distributor()' to integrate the distributor answer and create the final result.

- It sends this answer to the user by creating 'send-response-distributor' and it switches to FinalState.

3.4. Protocols Modelling/Sequence Diagram Creation

In the figure 8, we show how the three agents communicate via sending and receiving communications acts.

3.5. Global Protocol Modelling/Template Creation

Template is used to create global protocol between agents. In this system, two templates of are created; the template describing the general protocol between sender, distributor and the other one between distributor and wrapper. We do not have a DeadLine because of the simplicity of the example.

4. REWRITING LOGIC REVIEW

In rewriting logic, each concurrent system is represented by a rewrite theory $\mathfrak{R} = (\Sigma, E, L, R)$. Its static structure is described by the signature (Σ, E) , whereas its dynamic structure is described by the set of labelled rewrite rules R , which are applied modulo the equation E . An important consequence of the rewriting logic definition is that a rewrite theory $\mathfrak{R} = (\Sigma, E, L, R)$ can be viewed as an executable specification of the concurrent system that it formalizes. In this section we recall the basic definitions of the rewriting logic.

A labelled rewrite theory \mathfrak{R} is a 4-tuple $\mathfrak{R} = (\Sigma, E, L, R)$ where (Σ, E) is a signature; Σ is the sorts set and operators and E is a set of Σ -equations. The signature (Σ, E) is an equational theory which describes the particular algebraic structure of the states of a system (multiset, binary tree, etc.) which are distributed according to this same structure. $R \subseteq L \times (T_{\Sigma, E}(X))^2$ is the set of pairs whose first component is a label and the second is a pair of E -equivalence classes of terms, with $X = \{x_1, \dots, x_n, \dots\}$ a countable infinite set of variables. The elements of R are called conditional rewrite rules. They describe the elementary and local transitions in a concurrent system. Each rewrite rule corresponds to an action being able to occur,

simultaneously, with other actions. The rewriting will operate on equivalence classes of terms, modulo the set of equations E . For a rewrite rule $(r, ([t], [t']), ([u_1], [v_1]), \dots, ([u_k], [v_k]))$ we use the notation, $r: [t] \rightarrow [t']$ if $[u_1] \rightarrow [v_1] \wedge \dots \wedge [u_k] \rightarrow [v_k]$, where $[t]$ represents the equivalence class of the term t . A rule r expresses that the equivalence class containing the term t is changed to the equivalence class containing the term t' if the conditional part of the rule, $[u_1] \rightarrow [v_1] \wedge \dots \wedge [u_k] \rightarrow [v_k]$, is verified.

Given a labeled rewrite theory \mathfrak{R} , we say that \mathfrak{R} entails a sequent $r: [t] \rightarrow [t']$, or that $r: [t] \rightarrow [t']$ is a (concurrent) \mathfrak{R} -rewrite and write $\mathfrak{R} \vdash r: [t] \rightarrow [t']$ iff $[t] \rightarrow [t']$ is derivable from the rules in \mathfrak{R} by a finite application of the deduction rules (reflexivity, transitivity, congruence, and replacement) of rewriting logic.

A rewrite theory is a static description of a concurrent system. Its semantics is defined by a mathematical model which describes its behavior. The model for a given labelled rewriting theory $\mathfrak{R} = (\Sigma, E, L, R)$ is a category $\tau_{\mathfrak{R}}(X)$ whose objects (states) are equivalence classes of terms $[t] \in T_{\Sigma, E}(X)$ and whose morphisms (transitions) are equivalence classes of proof-terms representing proofs in rewriting deduction.

- **Reflexivity.** For every $[t] \in T_{\Sigma, E}(X)$:

$$\frac{}{[t] \rightarrow [t]}$$

- **Congruence.** For every $f \in \Sigma_n$, $n \in \mathbb{N}$:

$$\frac{[t_1] \rightarrow [t'_1] \quad \dots \quad [t_n] \rightarrow [t'_n]}{[f(t_1, \dots, t_n)] \rightarrow [f(t'_1, \dots, t'_n)]}$$

- **Replacement.** For every rewriting rule

$$r: [t(x_1, \dots, x_n)] \rightarrow [t'(x_1, \dots, x_n)] \quad \text{in } R,$$

$$\frac{[w_1] \rightarrow [w'_1] \quad \dots \quad [w_n] \rightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \rightarrow [t'(\bar{w}'/\bar{x})]}, \text{ such that}$$

$\bar{t}(\bar{w}/\bar{x})$ indicates the simultaneous substitution

of w_i for x_i in \bar{t} .

- **Transitivity.** $\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

5. MAUDE LANGUAGE

Maude is a specification and programming language based on rewriting logic [11], [5]. Maude

is simple, expressive and efficient. It is rather simple to program with Maude, considering that it belongs to the declarative programming languages. It is possible to describe using Maude different types of applications, from prototyping ones to high concurrent applications. Maude is a competitive language in terms of execution and simulation with imperative programming languages. In Maude language, two levels of specification are defined. The first level is related to the specification of the system while the second is related to the specification of the properties.

5.1 System Specification Level

This level is based on rewriting theory. It is mainly described by system modules. Three types of modules are, in fact, defined in Maude. The functional modules allow defining data types. The system modules define the dynamic behavior of a system. Lastly, the object-oriented modules which can, in fact, be reduced to system modules. However, they offer explicitly the advantages of the object paradigm.

Functional Modules. The functional modules define data types and related operations, which are based on equations theory. By using equations like simplification rules, each expression called term could be evaluated to its reduced form called canonical representation. All the equal terms by means of equations form an equivalence class. The canonical form represents all the terms of the same equivalence class. The set of all the equivalence classes of the ground (i.e., variable-free) terms constitutes a denotational model for a functional module (initial algebra). Equations in a functional module are oriented. They are used from left to right and the final result of the simplification of an initial term is unique independently of the order in which these equations are applied. In addition to equations, this type of modules supports membership's axioms. These axioms impose constraints so that a term is of a particular type if a certain condition is satisfied. This condition is a conjunction of equations and unconditional tests of memberships.

System Modules. The system modules define the dynamic behavior of a system. This type of module augments the functional modules by the introduction of rewriting rules. This type of module offers a maximum degree of concurrency. A system module describes a "rewriting theory" which includes kinds, operations and three types of statements: equations, memberships and rewriting



rules. These three types of statements can be conditional. A rewriting rule specifies a “local concurrent transition” which can proceed in a system. The execution of such transition, specified by the rule, can take place when the left part of a rule matches to a portion of the global state of the system and the condition of the rule is valid.

Object-Oriented Modules. Although an object-oriented module can be specified by a system module, the object-oriented module offers syntax more suitable compared to the system module to describe the basic entities of the object paradigm like, for an instance, objects, messages and configuration. To make easier the description to a user of Maude, the predefined object-oriented module CONFIGURATION is introduced, encapsulating the basic concepts of object-oriented programming. A part of this module is described in section 4.1. The remainder of this module will be presented thereafter. A typical form of a configuration is: Ob-1... Ob-m M-1... M-n

Such that Ob-1... Ob-m are objects, M-1... M-n are messages, it does not matter the order (commutativity). In general, a rewriting rule has the following form:

rl Ob-1... Ob-k M-1... M-n => Ob-1' ... Ob-j' Ob-K+1... Ob-m M-1'... M-n'

Such that Ob-1' ... Ob-j' are updated versions to the objects Ob-1 ... Ob-j if $j \leq k$, and Ob-k+1... Ob-m are new created objects. If a left part of a rule contains only one object and only one message, this rule is known as asynchronous. On the other hand, the rule is known as synchronous if its left hand side contains several objects. The remainder of the module CONFIGURATION defines the syntax of the objects as follows:

sorts Oid Cid .

sorts Attribute AttributeSet . subsort Attribute < AttributeSet .

op none : -> AttributeSet . op __ : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: none] .

op <_ : | _> : Oid Cid AttributeSet -> Object [Ctor object] .

In this syntax, the objects have the following general form: < O: C | att-1,..., att-k > such that O is an identifier of object, C is an identifier of a class, and att-1,..., att-k are attributes of objects. Only one rewriting rule makes it possible to express at the same time: consumption of certain floating messages, sending of new messages, destruction of

the objects, creation of new objects, changes of certain objects states, etc.

5.2 Properties Specification Level

This level of properties specification in Maude defines the properties of the system to be checked. By evaluating the set of states that are reachable from an initial state, Model Checking allows the verification a given property in a state or a set of states. According to the kind of the property, two kinds of Model Checking are defined: Model Checking Invariants and the well known LTL Model Checking.

5.2.1. Model Checking of Invariants

In this Kind of Model Checking, the property is specified as an invariant. Through the utilization of the command search, Maude system goes over all states accessible from initial state to prove that such invariant is violated or not by detecting some incoherent states describing ‘safety properties’ (well known in the literature as something bad should never happen). The syntax of the command search is as follows:

search in [Module-Name] : Initial-State =>* Any-State [such that Condition] .

If we instantiate Any-State by the general principal state of the specification, for instance CF:Configuration (CF is a variable of type Configuration) without indicating any condition, search command returns the accessibility graph of the specification, i.e. all reachable states from Initial-State. But, if we give a more detailed form of configuration or a condition, search command returns only states that match such form of configuration or verify such condition. That’s the way we will use this command to check necessary reachable configurations. If we give a condition we do not want it to be true in any accessible state, and if search command returns some states, that means the presence of some incoherent states. That’s the way we will use this command to check if there is or not incoherent states.

5.2.2. LTL Model Checking

A property is expressed in a temporal logic LTL (Linear Temporal Logic). Model Checking supported by the Maude’s platform uses LTL logic for its simplicity and the well defined procedures of decision which it offers (for more details, see [7], [5]). In a predefined module LTL, one finds the definition of operators for the construction of a formula (property) in linear temporal logic. By hiding certain details of implementation, one finds part of LTL operators in the syntax of Maude. LTL operators are represented in Maude by using a



syntactic form similar to their original form. For example, the operation `[]` is defined in Maude to implement the operator (always). An operator is applied to a formula to give a new formula.

fmod LTL is

```
...
----- Defined LTL operators
op <>_ : Formula -> Formula .      ----- eventually
op []_ : Formula -> Formula .      ----- always
op _=>_ : Formula Formula -> Formula .
----- strong implication
op _<=>_ : Formula Formula -> Formula .
----- strong equivalence
...
endfm
```

In addition to that, there is a need to an operator indicating when a given formula is true or false in a certain state. Such operator (`|=`) is found in the predefined module SATISFACTION.

```
fmod SATISFACTION is
protecting LTL . sort State .
op _|= _ : State Formula ~> Bool .
endfm
```

The state `State` is generic. After specifying the behavior of a system in a Maude system module, the user can specify several predicates expressing certain properties related to the system. These predicates are described in a new module which imports two others: the module SATISFACTION and the one describing the dynamic aspect of the system. Assume for example, `M-PREDS` the name of the module describing the predicates on the states of the system. `M` is the name of the module describing the behavior of the system. The user must specify that the selected state (configuration in this example and the rest of the paper) for its own system is sub-type of the sort `State`. At the end, we find the module `MODEL-CHECKER` which offers the Model-Check function. The user can call this function by specifying a given initial state and a formula. Maude Model Checker checks if this formula is valid (according to the nature of the formula and the procedure of Model Checker adopted by the Maude system) in this state or the set of all accessible states from the initial state. If the formula is not valid, a counterexample is displayed. The counterexample concerns a state in which the formula is not valid.

```
mod M-PREDS is
protecting M . including SATISFACTION .
subsort Configuration < State .
...
endm
fmod MODEL-CHECKER is
including SATISFACTION .
```

```
...
op counterexample : TransitionList TransitionList
-> ModelCheckResult [ctor] .
op modelCheck : State Formula
~> ModelCheckResult .
...
endfm
```

6. AUML-MAUDE TRANSLATION

In this section, we explain some ideas about our proposed translation from AUML to Maude that we use to create the tool AUML-Maude. The translation can not be completely automatic because of the some informal aspects of Maude. So we open the code to the user to add some rewriting rules concerning in particular intra-agent behaviour. The proposed translation touches some aspects of class diagram, sequence diagram and statechart diagram.

6.1. Class Diagram Translation

We used the concept of object to implement the AUML agent. Then, a class agent in AUML will be represented by an object class of Maude. Object concepts such attribute and method in AUML will be represented naturally by attribute and message concepts of Maude. In the sequel, we will explain in detail how we describe every concept of AUML by using Maude concepts.

Class Agent. We used class object in Maude to describe it.

Attribute. We used also attribute concept.

Method. It will be implemented by using message concept of Maude.

Action. Basically, an action will be represented by a message also Maude, but we created two kinds of messages: Pro-Active and Rea-Active, as follows:

```
sort Pro-Active . sort Rea-Active . subsort Pro-Active < Msg . subsort Rea-Active < Msg .
```

Communication-Acts. It is a message too, and here also we created a special kind of message that we called Communication-Acts:

```
sort Communication-Acts . subsort Communication-Acts < Msg .
```

Identification Mechanism. The `CONFIGURATION` module in Maude, offers the sort `Oid` as a generic mechanism of identification. We create for each class `C` translated, a space for the identification of its objects, that we called `COid`

(class name+Oid). To be a valid identification space, COid should be a sub-sort of Oid:
subsort COid < Oid .

For simplicity, we have chosen the string as mechanism identification, so we add:
subsort String < COid .

Role. First, we create a generic sort called Role and a generic attributed called CurrentRole of sort Role. This attribute will contain the current role of the agent.

op CurrentRole : _ : Role -> Attribute .

Of course, for more than one role, we count the maximal roles defined for an agent class, for instance n, and during the translation, we create new attributes CurrentRole1, ..., CurrentRole(n-1) (in addition to CurrentRole which already exists).

op role-1 : -> Role . op role-2 : -> Role op role-(n-1) : -> Role .

For the use of these attributes inside an object:

< A : C | CurrentRole : role-1, CurrentRole1 : role-2, ..., CurrentRole(n-1) : role-n , serveurState : InitialState , Atts >....

Cardinalities. To implement cardinalities, first we propose the module ACQ-LIST to create a special data type list that we called Acq-List. This data type will help us specifying acquaintances of an agent.

fmod ACQ-LIST is sorts Acq-Elt Acq-List . subsort Acq-Elt < Acq-List .

op empty : -> Acq-List [ctor] .

op _ : Acq-Elt Acq-List -> Acq-List [ctor] .

op error-Acq-Elt : -> Acq-Elt [ctor] . op error-Acq-List : -> Acq-List [ctor] .

var E : Acq-Elt . var L : Acq-List .

eq E ; empty = E .

op head-Acq : Acq-List -> Acq-Elt .

eq head-Acq(empty) = error-Acq-Elt .

eq head-Acq(E ; L) = E . eq head-Acq(E) = E .

op tail-Acq : Acq-List -> Acq-List .

eq tail-Acq(empty) = error-Acq-List .

eq tail-Acq(E ; L) = L . eq tail-Acq(E) = empty .

...

endfm

Depending on the number of links that an agent of a class, gets with others agents of other classes, we create a number of attributes equal to the number of links. We discuss cardinalities of the form n..m:

So, the agents of class C_l, have n..m cardinality with agents of class C,

The agents of class C_l, have n₁..m₁ cardinality with agents of class C₁,

...

The agents of class C_l, have n_k..m_k cardinality with agents of class C_k,

After that, we create attributes Acquaintance, Acquaintance-1, ..., Acquaintance-n-1:

op Acquaintance : _ : Acq-List -> Attribute .

op Acquaintance-1 : _ : Acq-List -> Attribute .

...

op Acquaintance-n-1 : _ : Acq-List -> Attribute .

Such that Acquaintance is a list containing between n and m elements, every element is an agent identifier of class C, every attribute Acquaintance-i is a list containing between n_i and m_i agent identifier of class C_i, ...,

and every attribute Acquaintance-i is a list containing between n_i and m_i agent identifier of class C_i.

In fact, we preferred offering Acquaintance as generic attribute and the others are created in the moment of the translation according of maximal the number of links that can have a class agent with others.

But in the sequel and for simplicity reasons, we consider only cardinalities of the form 0..1.

6.2. Third Level (Statechart Diagram) Translation

This diagram has already a representation in Maude language. To describe agent state in Maude, we use an attribute. First, we create a general sort we called GeneralState and two specific values of this sort: InitialState and FinalState to describe initial and final state of any agent. In the moment of the translation, for any class agent C, we create:

- A sort CStateKind (class name+"StateKind"): the type of all agent states, we have to put: subsort CStateKind < GeneralState.
- A state in Maude: op S : -> CStateKind . For every state S occurring in the agent state-transition diagram.
- An attribute CState (class name+"State"): op CState : _ : GeneralState -> Attribute . So, the value of this attribute are InitialState, FinalState and all states S of the agent state-transition diagram.

Let's take the statechart diagram of the figure 3, we focus only on the consumption the actions, and not how the action is created. That depends on the nature of the actions if it is reactive or proactive;

which will be discussed later. Then, when the agent is in its initial state and the occurrence of Action-1 allows it to change this state to S0:

```
rl [rule-name] : Action-1 < A : Class1 |
Class1State : InitialState, atts1 >=>
< A : Class1 | Class1State : S0, atts1 > .
```

Also, if the agent is the state S0 and the arrival of Action-2 allows it to change its state to S1, etc.

```
rl [rule-name] : Action-2 < A : Class1 |
Class1State : S0, atts1 >=> < A : Class1 |
Class1State : S1, atts1 > .
```

...

6.3. Second Level (Sequence Diagram) Translation

To allow two or many agents communicating via a protocol, we created some operations:

```
op _ : Configuration Configuration ->
Configuration [assoc comm id: none] .
op [] : Configuration -> Configuration .
op AgentProtocol : Configuration -> Msg .
```

The operation [] allows us to encapsulate an agent with its floating messages (actions, ...), for instance :

```
[Action1 ... Actionnn < A : Class1 | Class1State :
S0, atts1 >]
```

The operation AgentProtocol encapsulates all agents that are communicating via a protocol. If we get only two agents A and B communicating, we have the form:

```
AgentProtocol([ ... < A : Class1 | atts1 >], [ ... <
B : Class2 | atts2 >])
```

But if we get three agents A, B and communicating, so can get the form:

```
AgentProtocol([ ... < A : Class1 | atts1 >], [ ... <
B : Class2 | atts2 >], [ ... < C : Class3 | atts3 >])
```

That's why, we proposed the operation _ which allows to the operation AgentProtocol to have an extensible number of configuration as parameters.

Note. Let's note that when an agent is encapsulated inside a protocol, it does not mean that it can not work individually by consuming its own actions and changing its state.

Before translation, we need some information from the user. He should precise in what state of the agent sender a communication act will be sent and what action of the receiver agent this communication act will invoke. In general way, to send a communication act CA to agent B, the agent A prepares it locally by executing the following rewriting rule:

```
rl [rule-name] : < A : Class1 | Class1State : S0,
Acquaintance-i : B, atts1 >=>
```

```
CA (Operation-A < A : Class1 | Class1State : S0,
Acquaintance-i : B, atts1 >). [1]
```

According to AUMML model, when the agent creates a communication act, it activates also an operation Operation-A which could be a method or an action.

If the occurred operation is a method (Operation-A=Method-A) the consumption of Method-A needs some other rules that could change internal state of the agent.

But, we do not have such method behaviour in formal way that we could translate it automatically to Maude. All what we can do is creating automatically a rule which allows the consumption of the method obviously and creating the next action Action-A according to the statechart of the agent A:

```
rl [rule-name] : Method -A < A : Class1 |
Class1State : S0, atts1 >=>
```

```
Action-A < A : Class1 | Class1State : S0, atts1 > .
```

Note. Here Action-A is pro-active action, so the agent creates it by itself. But if the action is reactive, the agent waits the action coming from another agent which could be the user. In this case, the reactive action is put from the beginning (initial configuration) in the space accorded to the agent between [].

In this rule, Method -A does not infect the state of the agent; it is just a bridge to invoke the corresponding action in the statechart. But we do not stop here; we propose opening a window to the user to complete the behaviour of the method. Even he does not add any other rule, he has at least an executable Maude code that validate some aspects expressed by class, sequence and statechart diagrams.

In both cases, the operation is a method or action; at the end we get an action. This action is consumed by using the rewriting rules implementing statechart diagram. Such consumption makes the agent changing its state from S0 to S1 and makes it ready to send the CA to the corresponding agent B. The following rule allows passing CA from the space of A to the space of B:

```
rl [rule-name] : AgentProtocol([ CA < A : Class1 |
Class1State : S1, Acquaintance-i : B, atts1 >],
```

```
[ < B : Class2 | Class2State : S2, Acquaintance-j :
B, atts2 >], CF)
```

```
=> AgentProtocol([ < A : Class | Class-State : S1,
Acquaintance-i : A, atts1 >],
```

```
[CA < B : Class2 | Class2State : S2, Acquaintance-
j : B, atts2 >], CF) . [2]
```



CF is a variable of sort Configuration indicating the remaining of agents involved in the protocol, or CF is none (empty).

When B receives the message composed of CA, it continues its work by creating the appropriate action which is précised by the user Operation-B:

```
rl [rule-name]: CA < B : Class2 | Class2State : S2,
Acquaintance-j : B, atts2 >
=> Operation-B < B : Class2 | Class2State : S2,
Acquaintance-j : B, atts2 > [3]
```

...

When the three connectors 'AND', 'OR', and 'XOR' appear in the messages, we keep the form of these rules ([1], [2] and [3]) with only some little changes. Let's explain how we adapt these rules [1], [2] and [3] to translate messages containing one of the three connectors 'AND', 'OR', and 'XOR'.

'AND' Connector Translation. To send CA-1 CA-2 ... CA-n concurrently, we translate this in the same way as the rule [1], but instead of creating only one communication act CA, the agent A creates all the sequence as:

```
rl [rule-name] : < A : Class1 | At : Vi, Class1State :
S0, Acquaintance-i : B, atts1 > =>
CA-1 CA-2 ... CA-n (ActionA < A : Class1 | At :
Vi, Class1State : S0, Acquaintance-i : B, atts1 >).
```

After that, the agent A passes this sequence to the agent B, we change the rule [2] to the following rule:

```
rl [rule-name] : AgentProtocol([ CA-1 CA-2 ...
CA-n < A : Class1 | Class1State : S1,
Acquaintance-i : B, atts1 >], [ < B : Class2 |
Class2State : S2, Acquaintance-j : B, atts2 >], CF)
=> AgentProtocol([< A : Class | Class-State : S1,
Acquaintance-i : A, atts1 >],
[CA-1 CA-2 ... CA-n < B : Class2 | Class2State :
S2, Acquaintance-j : B, atts2 >], CF).
```

...

'XOR' Connector Translation. Here, we give the hand to the user, to choose an attribute and according to a value of this attribute, one of the CA-i will be sent. Let the At is the attribute chosen by the user, and after that, he has to indicate for every value which CA-i must be sent. Suppose that for the value Vi, it's CA-i that will be sent, we get the following rules:

```
rl [rule-name] : AgentProtocol([CF1 CA-1 < A :
Class1 | At : V1, Class1State : S1, Acquaintance-i :
B, atts1 >], [ < B : Class2 | Class2State : S2,
Acquaintance-j : B, atts2 >], CF)
=> AgentProtocol([< A : Class1 | At : V1,
Class1State : S1, Acquaintance-i : B, atts1 >],
[CA-1 < B : Class2 | Class2State : S2,
Acquaintance-j : B, atts2 >], CF)
```

...

```
rl [rule-name] : AgentProtocol([CA-n < A : Class1 |
At : Vn, Class1State : S1, Acquaintance-i : B, atts1
>], [ < B : Class2 | Class2State : S2, Acquaintance-
j : B, atts2 >], CF)
```

```
=> AgentProtocol([ < A : Class1 | At : Vn,
Class1State : S1, Acquaintance-i : B, atts1 >],
[CA-1 < B : Class2 | Class2State : S2,
Acquaintance-j : B, atts2 >], CF)
```

'OR' Connector Translation. The user must indicate the value Vi of an attribute to send some CA-i:

```
rl [rule-name] : AgentProtocol([CA-i1 CA-i2 ...
CA-ik < A : Class1 | At : Vi, | Class1State : S1,
Acquaintance-i : B, atts1 >],
[ < B : Class2 | Class2State : S2, Acquaintance-j :
B, atts2 >], CF)
=> AgentProtocol([< A : Class1 | At : Vi, |
Class1State : S1, Acquaintance-i : B, atts1 >],
[CA-i1 CA-i2 ... CA-ik < B : Class2 | Class2State :
S2, Acquaintance-j : B, atts2 >], CF)
Such that ij = 1, ..., n
```

...

6.4. First Level (Template) Translation

We do not need the translation of the template, because all details appear in the sequence diagram of level 2 that we will translated in the next section.

6.5. Generic Module

We regrouped all generic elements (sorts, attributes, actions and operations) that we proposed in one module (called GENERIC-PROTOCOL) to be included in any generated Maude code:

```
mod GENERIC-PROTOCOL is
pr ACQ-LIST . including CONFIGURATION .
sort GeneralState . sort Communication-Acts .
sort Role . sort Pro-Active . sort Rea-Active .
subsort Pro-Active < Msg .
subsort Rea-Active < Msg .
subsort Communication-Acts < Msg .
subsort Oid < Acq-Elt .
op CurrentRole : _ : Role -> Attribute .
op Acquaintance : _ : Acq-List -> Attribute .
op InitialState : -> GeneralState . op FinalState : ->
GeneralState .
op _->_ : Configuration Configuration ->
Configuration [assoc comm id: none] .
op [] : Configuration -> Configuration .
op AgentProtocol : Configuration -> Msg .
endm
```

7. DESCRIPTION OF THE AUML-MAUDE TOOL

In this section and through the previous example, we will explain some aim functionalities of the AUML-Maude application.

7.1. General Description and Architecture of the Tool

As described in figure 4, the most principal working steps of the AUML-Maude Edition and Generation tool. The figure 5 summarizes the principal steps of the AUML-Maude LTL Properties Checker. The architecture of this last is similar to 'AUML-Maude Simulator' and 'AUML-Maude Invariants Properties Checker'.

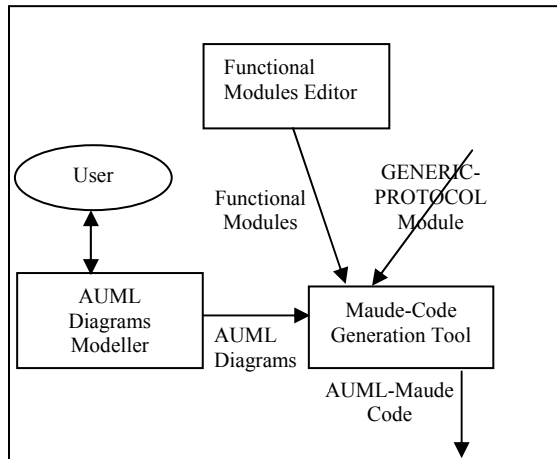


Figure 4. Methodical view on AUML-Maude Edition and Generation Tools

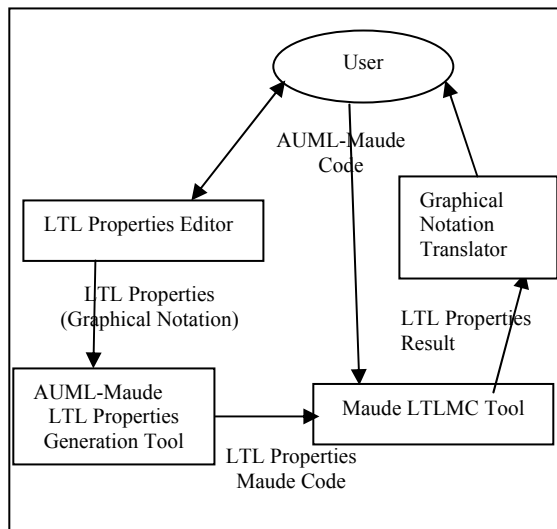


Figure 5. Methodical view on AUML-Maude LTL Properties Checker

Now, through the previous example, we explain in following sections the main options of the application.

7.2. Main Functionalities of AUML-Maude Tool

7.2.1. Functional Module Editor

Types should be created in Maude. So, we provide a simple text editor to the user create his specific data types. But, a little library containing the classical basic data types is available; this library containing some data types supported by Maude or we developed; it includes: Float, Int, String, List, Queue, Stack ... etc. After creating a new data type, the user can add it to the library.

7.2.2. AUML Graphical Edition (Modelling)

We give to the user a battery of tools to create and manipulate different AUML diagrams. Of course, there are some classical accessories buttons for zoom in, zoom out, background colour change, ...

Class Diagram Tools. As showed in the figure 6, necessary tools to create and update class diagram and its elements are available including tools to create and update agent class: agent role, state (attribute), actions, communication actions; general protocol, cardinalities, ..

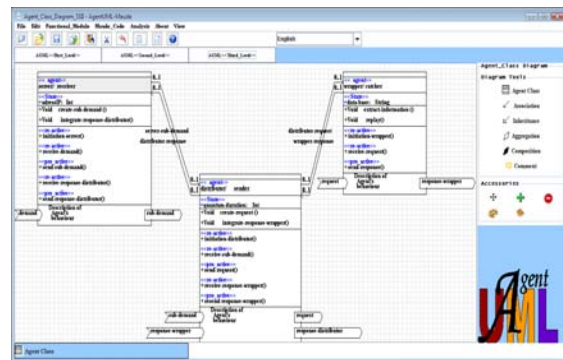


Figure 6. Class Diagram menu and SSII-Example Creation

First Level Tools. The figure 7 depicts the interface of the set of tools we provided to the user to create agents templates. In this figure, we created the two templates of the example, the template describing the general protocol between sender, distributor and the other one between distributor, wrapper.

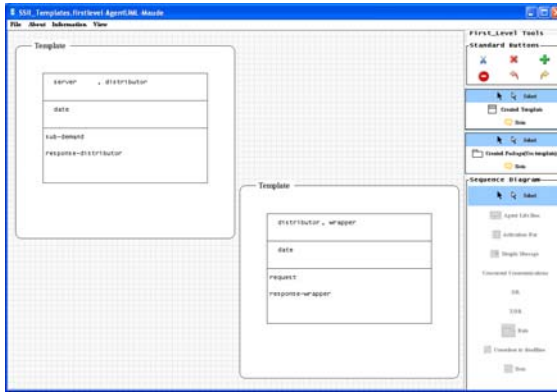


Figure 7. First Level menu and SSII-Example Template Creation

Second Level Tools. In the figure 8, one finds second level interface with necessary tools to create sequence diagram. The figure contains also the sequence diagram of the SSII example.

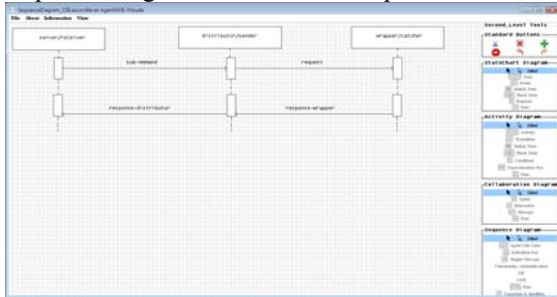


Figure 8. Second Level menu and SSII-Example Sequence Diagram Creation

Third Level Tools. In the figure 9, one finds third level interface with necessary tools to create statecharts of agents' classes. The figure contains also the statecharts of the three agents' classes of the SSII example.

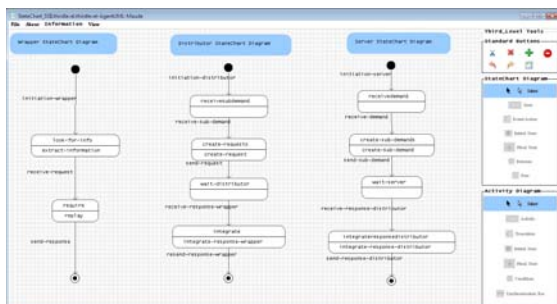


Figure 9. Third Level menu and SSII-Example Statechart Diagram Creation

7.2.3. Maude Code Generation

According to the translation strategy that we proposed in the section 6, when the user asks for 'Code generation' in 'Maude Code' menu, he gets first a space to add some rewriting rules completing

the agent's behaviour and after that he gets his the equivalent Maude code. The figure 10 shows a part of Maude code equivalent to the SSII example.

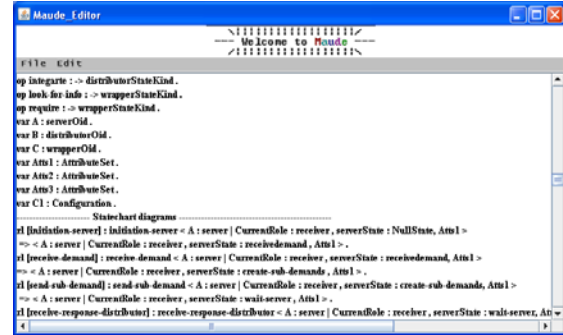


Figure 10. Maude Code Generation of the SSII system

7.2.4. Analysis Tools

In this section, we explain how every analysis tool works and the Maude code generated for every part.

Initial Configuration Creation. Indeed, an initial state is constituted of instances of different classes and some floating messages. Then, we give to the user a graphical way to create its initial configuration. In the window of figure 11, we show how the user can create his instances of class, we give display to him:

- List of actions that must appear form the beginning (those invoked by the user);
- List of existent classes, the user chooses a class to create instances of this class, to instantiate a class, the user should fill: 'instance name' (of type String), 'attributes values' and 'initial state'.

After instantiating a class, the information available like: Instance name: CName; each Attribute At_i is V_i; ClassState : S; allow us to create the initial configuration. If the user gives all the values of the attribute, so we create in Maude, the instance: < CName : Class | At₁ : V₁, ..., At_n : V_n, ClassState : S >

But if the user does not fill all the attributes values, we add in the instance:

< CName : Class | ..., At_j : V_j, ClassState : S, atts >.

Finally, the initial configuration of all system is composed of all these instances and some floating actions encapsulated as parameter of the function AgentProtocol and every instance is encapsulated between [] with its appropriate actions: AgentProtocol([Action₁ ... Action_m < CName : Class | ..., At_j : V_j, ClassState : S, atts >], ...).

Consequently, the initial configuration of the SSII example is as follows:

op Initial-Configuration : -> Configuration .

eq Initial-Configuration =

```
AgentProtocol([initiation-server receive-demand <
"ser" : server | adressIP : 10 , CurrentRole : receiver
, serverState : InitialState , Acquaintance : "dis" > ]
, [ < "dis" : distributor | quantum-duration : 1 ,
CurrentRole : sender , distributorState : InitialState
, Acquaintance : "wrap" , Acquaintance-1 : "ser" > ]
, [ < "wrap" : wrapper | data-base : "data" ,
CurrentRole : catcher , wrapperState : InitialState ,
Acquaintance : "dis" > ] ) .
```

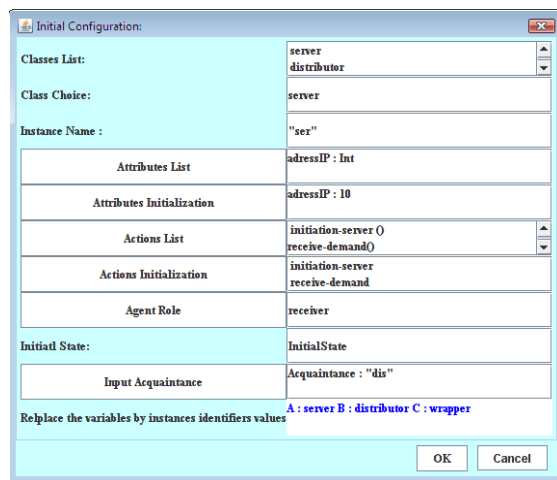


Figure 11. An Example of class instance creation

Simulation. The simulation consists of transforming the initial state to another by doing one or many rewriting actions. To do simulation, the simulator needs from the user the initial state of the AUMI model. The user may give to the simulator the number of rewriting steps if he wants to check intermediary states. If this number is not given, the simulator continues the rewriting operation until reaching final state. In figure 12, one asked the application to perform the simulation on the previous initial configuration without indicating the number of rewriting steps (0). After validation, the code Maude generator creates: rew Initial-Configuration .

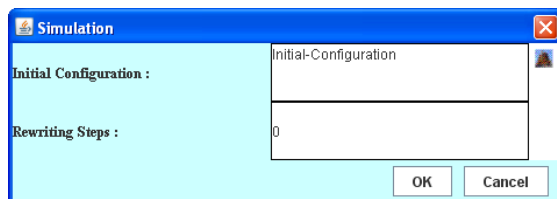


Figure 12. Simulation window

We notice that infinite case is possible. Let's get back to the example, after displaying the Maude code equivalent to AUMI model; we can now execute the code by clicking 'Simulation' option of 'Analysis' menu. In our case, the unlimited rewriting of the example under Maude system gives the result in the figure 13. Let's note that we exposed the result configuration in textual way in Maude language and in graphical manner when we shows every class instance is showed in a window.

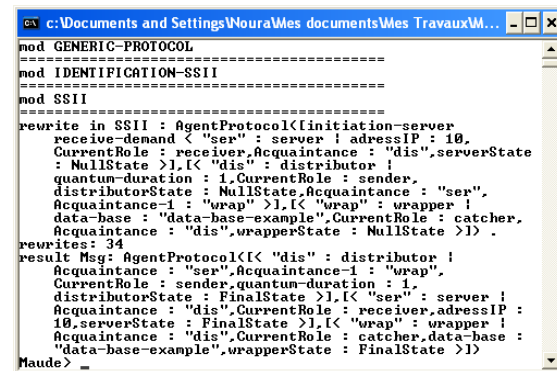


Figure 13. Simulation of the SSII example under Maude system

Model Checking of Invariants. We give the interface to the user, to seize the necessary parameters as described in the figure 14 we make the work of the user as easy as possible.

Source State: The user can put any configuration but we give him the obvious choice which is Initial-Configuration.

States Number: natural number N which could be optional.

Rewriting Proof Steps: It could be *, +, ! or natural number.

Target State: We give to the user the obvious choice which is C:Configuration.

Condition: It's optional.

After validating, the tool creates the code and calls Maude system for execution. For the translation, for instance, if State Number and Condition are empty, we get the code:

search SourceState =>SP TargetState .

Such that SP is Rewriting proof steps, and N is State Number. Otherwise, we get:

search [N] SourceState =>SP TargetState such that Condition .

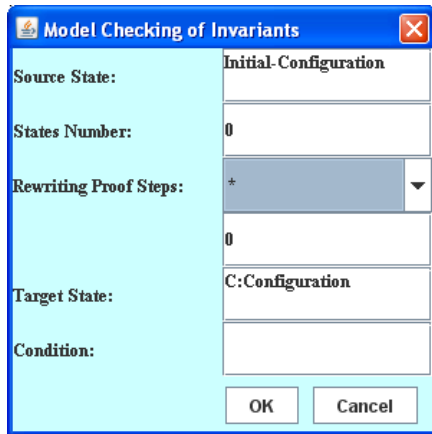


Figure 14. Model Checking of invariants Window

LTL Model Checking. In the option 'Analysis' of the main menu, we choose 'LTL Model Checking' option which contains three options that are: 'Predicate', 'Property' and 'Model Check' to create respectively predicates, properties and to model check some created properties.

Predicates. When the user chooses to create a predicate, a window opens and the user should seize 'Predicate Name' and to add 'Variables', 'Signature' and 'Equations' by clicking respectively on 'Add Variables', 'Add Signature' and 'Add Equations'.

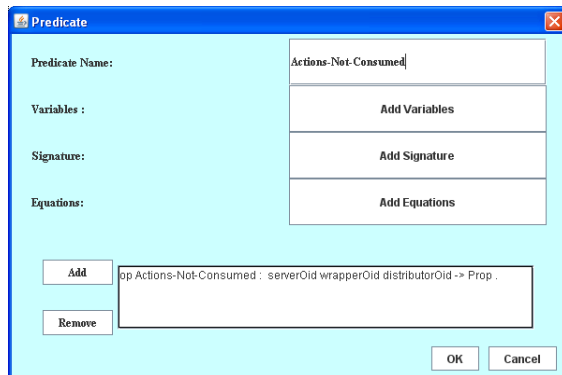


Figure 15. Predicate main window

When the user clicks on 'Add Variables', another window will be opened entitled 'Variables' as described in the figure 16, to fill all needed variables (if there are variables) concerning this predicate. The user can add as many variables as he needs. For sorts, the user chooses only a sort from a list of valid already defined sorts.

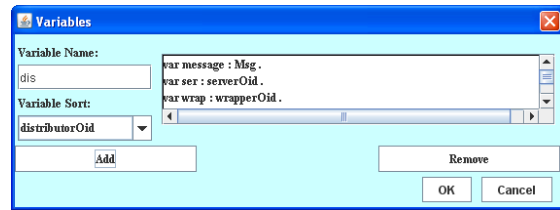


Figure 16. Variables creation window

In this case, the code that will be generated is: `var ser : serverOid . var wrap : wrapperOid ...`. If the user clicks on 'Add Signature', another window will appear to complete the 'Signature' (figure 17) of the predicate. For signature, the user needs only entering the list of the 'Sorts' that are the domains of the predicate, because the co-domain of any predicate is predefined and it is 'Prop'. Of course, the user has a limited list of valid already defined sorts to choose among them. For instance, the signature example of the figure ... will be translated to: `op Actions-Not-Consumed : serverOid wrapperOid distributorOid -> Prop .`

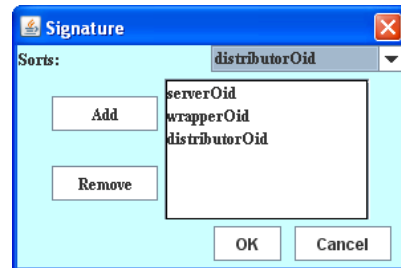


Figure 17. Signature predicate creation window

Finally, the window of the 'Equations' will be opened when clicking on 'Add Equations'. For an equation, we need the parameters in the figure 18.

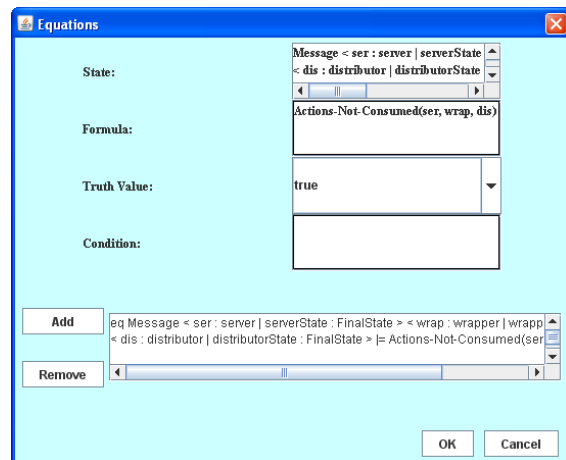


Figure 18. Predicate equation creation window

In general way, only Condition is optional, if Condition is not empty, so we get the generated code equivalent to the equation: $ceq \text{State} \models \text{Formula} = \text{TruthValue} \text{ if Condition}$.

Otherwise, we get: $eq \text{State} \models \text{Formula} = \text{TruthValue}$.

So, the equation example (figure 18) will be translated in Maude:

```
eq Message < ser : server | serverState : FinalState
> < wrap : wrapper | wrapperState : FinalState >
< dis : distributor | distributorState : FinalState > |=
Actions-Not-Consumed(ser, wrap, dis) = true .
```

Properties. To create a property, the user needs 'Property Name' and 'Property Contents'. We give buttons containing all LTL operators to make filling 'Property Contents' easy. The figure 19 shows a property creation example.

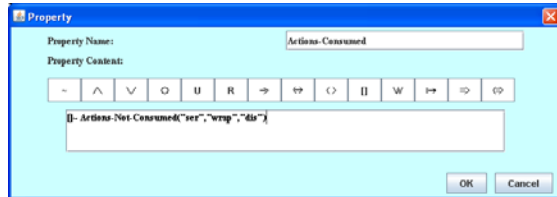


Figure 19. Property creation window

The property example of figure 19 will be translated in Maude as:

```
op Actions-Consumed : -> Prop .
eq Actions-Consumed = []~ Actions-Not-Consumed("ser", "wrap", "dis") .
```

Model Check. A list of all created properties is given to the user in a window (figure 20); he can choose one or many of them to model check. After he makes his choice, the tool creates the appropriate code of all the parts (predicates, properties and Model Check) and calls the Maude LTL Model Checker to verify the properties. The code will be in a file, and it is composed of two modules. The module MODULE-PREDICATS which contains the definition of predicates, such MODULE is the name of module containing the Maude code generated for the user AUML models. The second module is called MODULE-PREDICATS-CHECK which contains the definition of Initial-Configuration and the LTL properties. The previous created LTL properties example will invoke the creation of a file containing the following Maude code:

```
in model-checker.maude --- To add automatically
in ssii.maude ----- File containing the Maude code
of SSII example
```

```
mod SSII-PREDICATS is pr SSII . pr MODEL-
CHECKER . subsort Configuration < State .
----- Variables
var ser : serverOid . var wrap : wrapperOid . var dis
: distributorOid .
...
----- Predicates
op Actions-Not-Consumed : serverOid wrapperOid
distributorOid -> Prop .
eq Message < ser : server | serverState : FinalState
> < wrap : wrapper | wrapperState : FinalState >
< dis : distributor | distributorState : FinalState > |=
Actions-Not-Consumed(ser, wrap, dis) = true .
...
endm
mod SSII-PREDICATS-CHECK is
including SSII-PREDICATS . including MODEL-
CHECKER . including LTL-SIMPLIFIER .
op Initial-Configuration : -> Configuration .
...
----- Properties
op Actions-Consumed : -> Prop .
eq Actions-Consumed = []~ Actions-Not-Consumed("ser", "wrap", "dis") .
...
endm
----- LTL Model Checking of properties
red in SSII-PREDICATS-CHECK :
modelCheck(Initial-Configuration , Actions-Consumed) .
```

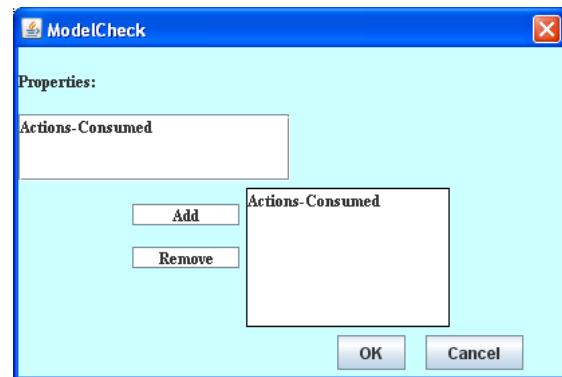
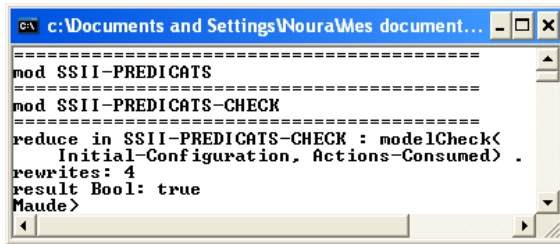


Figure 20. Model Check window

The execution of this generated code for LTL properties under Maude system gives the result described in the figure 21.



```

=====
mod SSII-PREDICATS
=====
mod SSII-PREDICATS-CHECK
=====
reduce in SSII-PREDICATS-CHECK : modelCheck(
  Initial-Configuration, Actions-Consumed) .
rewrites: 4
result Bool: true
Maude>

```

Figure 21. Properties verification under Maude system

7.3. Technical Aspect of the AUML-Maude Tool

This tool is implemented under MS-Windows XP with the following tools:

- The programming language Eclipse 3.4 is used for implementing the graphical editor, and the translation of graphical representation of a AUML model to its equivalent Maude description;
- The version 2.0.1 of Maude system is used for the simulation and Model Checking of invariants of the generated description of the AUML model;
- The Maude LTL Model Checking.

For graphic modelling reason, we held the Eclipse language with regard to the other languages for its graphics power, its capacity to manipulate objects and its compatibility with the other languages.

8. CONCLUSION & FUTURE WORK

In this paper, we have proposed some ideas about translation of AUML concepts to Maude language; also we have shown the outline of a graphical application for AUML model. This tool allows using AUML model in simple manner by introducing a graphical interface. This is not new, but in our knowledge; our proposed application is among few tools that allow the translation of this graphic representation to a Maude code. We take advantage of Maude system to run, simulate and verify the obtained AUML-Maude code.

We consider this work as an important investigation way, because the creation of a formal version of AUML multi-agents model and its verification by using the same language (Maude), allows us to give sound and complete semantics to AUML model in individual way. On other hand, it allows its integration with other multi-agents models (that we described in Maude before ([3], [4]) in one language and so their interoperability. Consequently, several models developed initially on different plate-forms can now communicate easily thanks to their formal versions Maude-based. Also these different multi-agents models can be

described and checked by using only a logic which allows a rigorous and founded formal verification.

The work presented in this paper constitutes the first step in the construction of a rich environment for edition and analysis AUML model. The analysis techniques including simulation, Model Checking of invariants and LTL Model Checking allow the verification of the 'correctness of a specification'. On another hand, Maude offers other techniques to check the 'correctness of the semantic of specification'. We plan integrating to the proposed environment these techniques supported by Maude. In our sense, Maude offers an environment presenting enough possibilities for specification, programming, validation and verification of MAS. Consequently, we do not need the hybrid integration of many methods which often causes complexity of description, confusion, semantic loss and not rigorous checking.

REFERENCES:

- [1] AgentBuilder R.M, An Integrated Toolkit for Constructing Intelligent Software Agents, AgentBuilder, *Reference Manual*, Avril 2000.
- [2] I. Bakam, F. Kordon, C. Le Page, F. Bousquet, Formalization of a Spatialized Multiagent Model Using Coloured Petri Nets for the Study of a Hunting Management System, *FAABS2000*, Greenbelt, 2000.
- [3] N. Boudiaf, F. Mokhati, M. Badri, and L. Badri. "Specifying DIMA Multi-Agent Models Using Maude". M. W. Barley and N. Kasabov (Eds.) : PRIMA 2004, Lecture Notes in Artificial Intelligence (LNAI) 3371, pp. 29–42, 2005, Springer-Verlag Berlin Heidelberg 2005.
- [4] N. Boudiaf, F. Mokhati and M. Badri. "Supporting Formal Verification of DIMA Multi-Agents Models: Towards a Framework based on Maude Model Checking". *International Journal of Software Engineering and Knowledge Engineering*, ISSN: 0218-1940, Vol. 18, No. 7, November 2008.
- [5] M. Clavel, et al., Maude Manual (Version 2.3) , *Internal report*, SRI International, 2007.
- [6] S. A. DeLoach, M. Wood, Developing Multiagent Systems with agentTool, (ATAL'2000), Berlin, 2001.
- [7] S. Eker, José Meseguer, and Ambarish Sridharanarayanan., The Maude LTL model checker, Volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
- [8] J. Ferber, O. Gutknecht, MadKit User's Guide, *Version 3.1 - last modification: 09/07/2002*.



- [9] J.-L. Koning, Algorithms for Translating Interaction Protocols into a Formal Description, *IEEE International Conference on Systems, Man, and Cybernetics Conference*, Tokyo, Japan, 1999.
- [10] J. Meseguer, Rewriting Logic as a unified model of concurrency: a Progress Report, *Springer-Verlag, LNCS 119*, pp. 331-372, 1996.
- [11] J. Meseguer, Rewriting Logic and Maude: a Wide-Spectrum Semantic Framework for Object-Based Distributed Systems, *FMOODS2000*, 2000.
- [12] F. Mokhati, N. Boudiaf, M. Badri, and L. Badri. "Translating AUML Diagrams into Maude Specifications: A Formal Verification of Agents Interaction Protocols". *Journal of Object Technology (JOT)*, ISSN 1660-1769, USA, 2007.
- [13] P. Torroni, Computational Logic in Multi-Agent Systems : Recent Advances and Future Directions, Computational Logic in Multi-Agent Systems, *Special Issue of Annals of Mathematics and Artificial Intelligence*, Vol. 42 Nos. 1-3, pp. 293-305, 2004.