JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 4, Mai - June 2007

Translating AUML Diagrams into Maude Specifications: A Formal Verification of Agents Interaction Protocols

Farid Mokhati, Département d'Informatique, Université Larbi Ben M'hidi, Algérie.
Noura Boudiaf, CNAM, Laboratoire Cederic, Paris, France.
Mourad Badri and **Linda Badri,** Département de Mathématiques et d'Informatique, Université du Québec à Trois-Rivières, Canada.

Abstract

Agents Interaction Protocols (AIPs) play a crucial role in multi-agents systems development. They allow specifying sequences of messages between agents. Major proposed protocols suffer from many weaknesses. We present, in this paper, a formal approach supporting the verification of agents' interaction protocols described by using AUML formalism. The considered AUML diagrams are formally translated into Maude specifications. Based on rewriting logic, the formal and object-oriented language Maude offers an interesting way for concurrent systems formal specification and programming. The Maude environment integrates a model-checker based on Linear Temporal Logic (LTL) supporting formal verification of distributed systems. The proposed approach essentially allows: (1) translating the description of agents' interactions, specified using AUML formalism, in a Maude specification and, (2) applying the model-checking techniques supported by Maude to verify some properties of the described system. A case study is presented to illustrate our approach.

1 INTRODUCTION

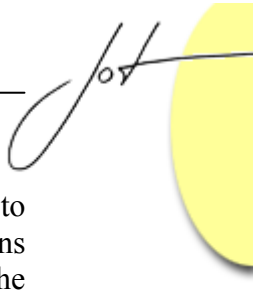
In Multi-Agents Systems (MAS), agents interact to exchange information, to cooperate and to coordinate their tasks. The usual approaches consist of describing their interactions as protocols [Hug02, Hug04]. Several Agents' Interaction Protocols (AIPs) have been proposed in the literature [Gue03, Toi04]. They are used to manage and to control interactions in MAS [Olu05]. Despite the fact that interaction protocols are an important part of multi-agents infrastructures, many of the published protocols are semi-formal, vague or contain errors [Pau03a, Mor05]. Knowing that AIPs play a crucial role in MAS development [Woo00a], their formal specification, as well as their verification, constitute an essential task [Maz02, Gio04]. In the field of agents' behaviour specification, three major approaches emerge in the literature: state-charts based approaches [Tra99, Pau03a],

Cite this column as follows: Farid Mokhati, Noura Boudiaf, Mourad Badri and Linda Badri: "Translating AUML Diagrams into Maude Specifications: A Formal Verification of Agents Interaction Protocols", in Journal of Object Technology, vol. 6, no. 4, May - June 2007, pp 77 - 102 http://www.jot.fm/issues/issue_2007_05/article2

Petri Nets (PN) based approaches [Cos99, Bak00] and, finally, approaches representing an adaptation of object-oriented specification methods [Ode00, Ode01].

Among representations of interaction protocols proposed in the literature, we can quote the AUML formalism (Agent UML) [Ode00, Ode01, Hug04]. It represents, in fact, the first result emerged from the cooperation established between the FIPA (Foundation of Intelligent Physical Agents) and the OMG group (Object Management Group) for facilitating the penetration of the agent technology in the industry. AUML represents an extension of the UML language [Mul00], especially supporting the specification of interactions between agents [Hug04, Pau04]. AUML diagrams adopt, in the description of interactions between agents, an approach in several layers. However, these diagrams only offer a semi-formal specification of interactions. This weakness may generate several problems. Indeed, the lack of formal semantics in AUML [Ast98, Reg99, Mor05], such as in UML, can lead to several incoherences in the description of a MAS's behaviour. It is difficult, especially in the case of complex MAS, to detect this kind of defects. In this context, the use of appropriate formal notations offers several advantages. It essentially allows producing rigorous and precise descriptions supporting efficiently the verification and validation process. Formal verification of software systems constitutes an important and difficult task [Fer04, Cla02]. In the literature, three major classes of formal verification approaches emerge [Sch99]: the proof based techniques [Kri93], the model-checking based techniques [Thi05] and the testing techniques [Sch99].

We focus, in this paper, on formal verification of interaction protocols described by using AUML. The adopted approach is based on model-checking techniques supported by Maude. Model checking offers several advantages relatively to traditional approaches based on simulation, testing techniques, or deductive reasoning. Its main advantage is that, at least for finished-states systems, it can be executed in a completely automatic and efficient way [Cla02]. Based on a sound and complete logic, called the rewriting logic [Mes92a, Mes92b], the Maude language [Mes92b, Man99, McC03], seems to us in this context an interesting candidate. It offers, in fact, through its various constructions, an interesting way for specifying and programming concurrent systems. It offers all the basic elements allowing formally specifying and verifying multi-agent interactions. The rewriting logic and the Maude language, in particular, unify several concurrency models [Mes92b]. Petri nets, state-transitions systems, Lotos, modal logic, temporal logic and other formalisms are integrated in the rewriting logic [Gab04, Ver03]. Our choice of this language was not only based on its expressive power, but also on its possibility to support simulations, therefore allowing formal verification. Verification of a system in Maude is based on the concepts of Model-Checking. Model checking techniques [Cla02, Thi05] take more and more importance in the domain of concurrent systems verification. The Maude model-checker [Eke02] was conceived by combining Maude and the linear temporal logic (LTL), taking advantage of the two formalisms. Moreover, Maude offers a remarkable descriptive power and the LTL offers recent and advanced techniques of model checking. Compared to other model-checkers, like SPIN [Hol97], Maude seems more expressive. It allows easily specifying different kinds of concurrent systems [Eke03].



We explored, in a previous work [Mok04], the feasibility as well as the interest to generate Maude specifications from AUML diagrams. The generated Maude descriptions have been validated by simulation thanks to the tool supporting this language. The present paper focuses on the formal verification of interactions protocols described by AUML diagrams using Maude's model checker. We illustrate the possibility, through the use of the Maude environment, of modelling and evaluating MAS's properties. Some properties depend on the agents' internal behaviour, such as decision-making, and others are relative to the agents' collective behaviour such as the verification of system coherency and termination.

The remainder of the paper is organized as follows. In Section 2, we give a general outline on related work. We summarily present, in Section 3, the AUML formalism. In Section 4, we give a brief preview on the rewriting logic as well as the Maude language. Verification of interactions protocols described by AUML formalism using Maude is presented in Section 6. Finally, in Section 7, we give some conclusions and future work directions.

2 RELATED WORK

Wooldridge et al. have proposed an approach supporting the verification of MAS with MABLE [Woo02], an automatic design and verification language for multi-agents systems. In a MABLE system, agents have a mental state (beliefs, desires and intentions). The verification process is based on the SPIN model-checker [Hol97], a model-checking system for finished-states systems. To chech claims about a MABLE system and to simulate its execution, SPIN uses two descriptions: a description of the MABLE system in PROMELA [Hol97], and a description of the claims to be checked expressed in quantified linear temporal BDI logic called *MORA* (a cut-down version of *LORA* [Woo00b]). Bordini and al. [Bor03] proposed AgentSpeak(F), a variation of the AgentSpeak(L) language [Rao96] that is a logical programming language for BDI agents. The goal of this work is to facilitate the verification of the AgentSpeak(L) systems by using model checking techniques. The key first step in this approach consisted to restrict AgentSpeak(L) to finished-states systems. The result is AgentSpeak(F), a version to finished-states of AgentSpeak(L). The verification of AgentSpeak(F) programs SPIN requires on one hand, the translation of these programs in PROMELA, the model specification language for the SPIN model-checker and, on the other hand, the mapping of the properties expressed in the BDI logic in the temporal linear logic (LTL). These works don't deal with the problem of verification and proof of protocols properties [Gio04].

Our approach is comparable, in terms of objectives, to the previously quoted approaches. Our objective consists, essentially, to support formally the specification and verification of the agents' interaction protocols in a MAS described by using AUML formalism. In the field of the description of agents' interaction protocols, AUML is certainly the best-known language [Hug02]. However, its graphic aspect as well as its lack of formal semantics can generate several incoherences in the description of MAS'

behaviour. The major motivations of this work are: (1) translating the description of agents' interactions, specified using AUML formalism, in a Maude specification and, (2) applying the model-checking techniques supported by Maude to verify some properties of the described systems. Compared to SPIN, Maude seems more expressive. One can easily specify different kinds of concurrent systems using Maude and reason about those specifications using other formal methods and tools [Eke03]. Another advantage of Maude is that the integration of model checking with theorem techniques becomes quite seamless [Eke03].

3 AUML

UML [Mul00, OMG01] is widely used in object-oriented systems (OOS) analysis and design. However, it is not adapted to the modelling of MAS. This is essentially due to the fundamental differences between OOS and MAS. Compared to objects, agents are relatively active and autonomous. Furthermore, objects are reactive while agents are proactive and social [Ode00, Kav03]. To fill these weaknesses, the FIPA and the OMG extended UML by proposing a new structural elements and diagrams allowing increasing the expression power of the basis language, for tacking into account agent-oriented concepts. The result of this cooperation was the definition of the language Agent UML (AUML) [Ode00, Ode01, Hug04]. To represent multi-agent interaction protocols, AUML adopts in fact an approach in three layers. It uses, in the first level, package and template to represent the whole protocol. Sequence diagrams, collaboration diagrams, activity diagrams, and state diagrams are used to represent interactions between agents. Furthermore, activity diagrams and state diagrams are also used to capture the agent's internal behaviour (for more details see [Ode00]).

This paper focuses on the following concepts: *package* and *template* to represent interaction protocols, *sequence* diagrams for the description of interactions between agents and *state* diagrams for specifying agent's internal behaviour. The representation of nested protocol using packages as well as the use of AUML activity diagrams and collaboration diagrams will be considered in a future work.

Level 1: Representing Agents Interaction Protocols

Figure 1 illustrates an example of an agents' interaction protocol. It describes, using an AUML sequence diagram, the FIPA Contract Net protocol [FIP02]. When invoked, the Initiator agent sends a *call-for-proposal* to a Participant agent. Before a given deadline, the The Participant agent can submit to the Initiator agent a proposal (*propose*), refuse to submit a proposal (*refuse*), or indicates that it didn't understand (*not-understood*). The proposal formulated by the Participant agent can either be accepted or rejected by the Initiator agent. When it receives a proposal acceptance (*accept-proposal*), the Participant agent informs the Initiator agent about the proposal's execution. However, the Initiator agent can cancel the execution of the proposal at any time. Figure 1 also illustrates two relative fundamental concepts to this level:

- Package : a conceptual aggregation of interaction sequences. It allows treating the protocol like a reusable entity. The tabbed folder notation at the upper left indicates that the protocol is a package.
- Template : represented by a dashed box at the upper right corner. The template concept allows a protocol described by a package to be personalized for the analogous problem domains.

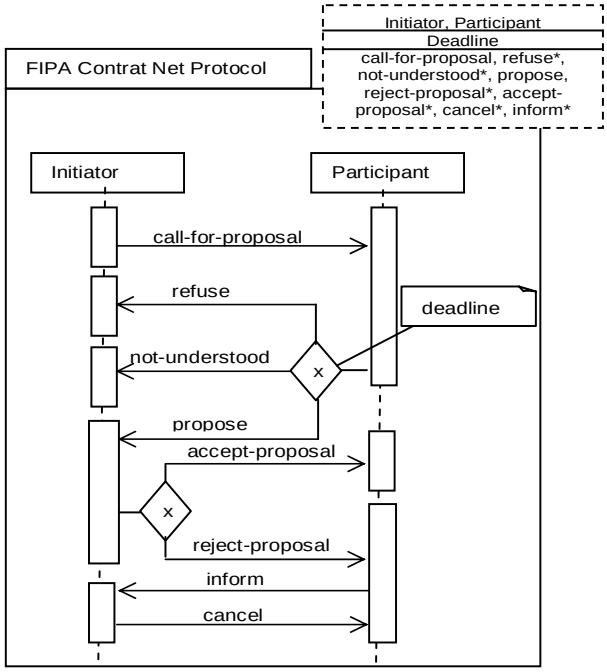


Figure 1: An interaction protocol expressed as a template package

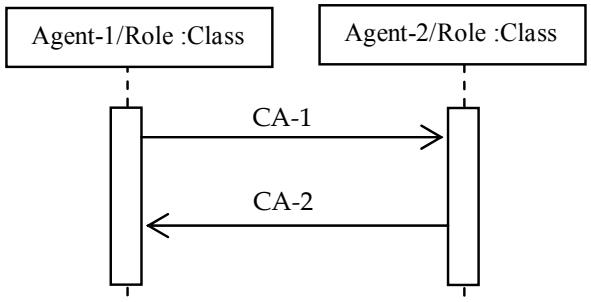


Figure 2: Basic format for agents communication

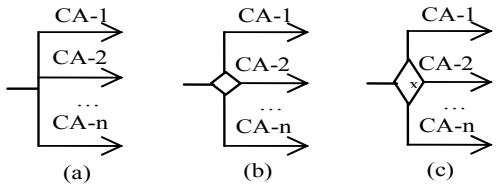


Figure 3: Recommended extensions that support concurrent threads of interaction

Level 2: Representing interactions between agents

We use, in this paper, sequence diagrams to describe interactions between agents. Figure 2 illustrates the basic format of communication between agents. Instead of the message style defined in UML, AUML uses communication acts (CA). To support the description of threads of interaction, AUML introduced three ways allowing expressing the multiple threads (see figure 3). Figure 3(a) indicates that all CA-i (*CA-1, ...CA-n*) are sent concurrently. Figure 3(b) includes a decision box indicating which CAs (zero or more) are going to be sent. Figure 3(c) indicates that only one CA is going to be sent. An agent sender can send concurrent acts to an agent playing different roles. They can, however, be sent to different agents.

Level 3: Representing agent's internal behavior

AUML offers two alternatives to represent an agent's internal behaviour. The first consists in using state diagrams, while the second consists in using activities diagrams. As mentioned previously, we use in this paper state diagrams for modelling the agent's internal behaviour. Figures 4.a and 4.b represent the agent's internal behaviour respectively for Initiator and Participant agents of figure 1.

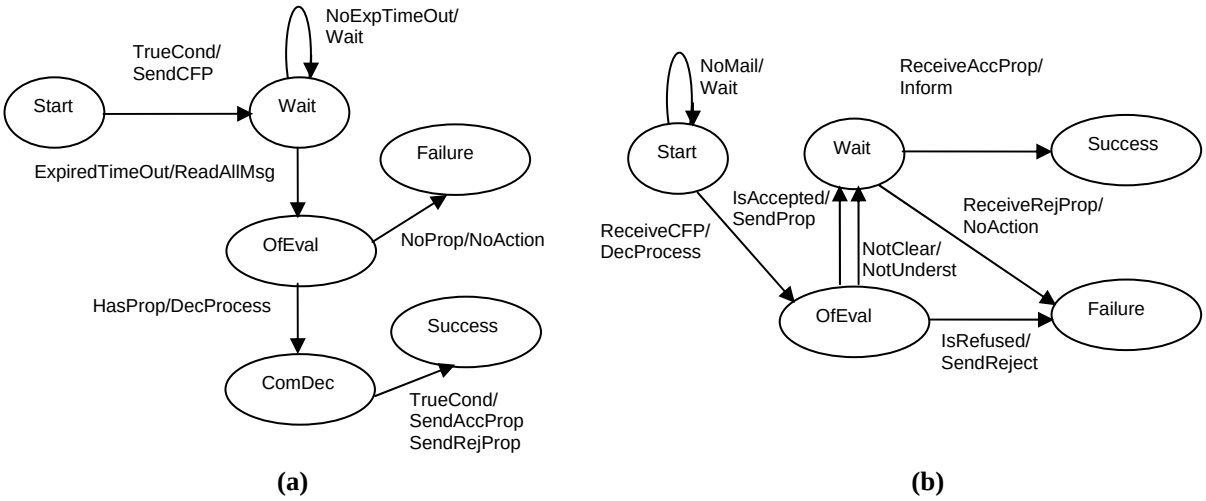


Figure 4: Internal behaviour of the agents Initiator and Participant.

4 REWRITING LOGIC AND MAUDE LANGUAGE

Rewriting Logic

The rewriting logic, having a sound and complete semantic, was introduced by Meseguer [Mes92b, Mes00, Mes03]. It allows describing concurrent systems. This logic unifies all the formal models that express concurrence [Mes90]. In rewriting logic, the logic formulas are called rewriting rules. They have the following form: $R:[t] \rightarrow [t'] \text{ if } C$. Rule

R indicates that term t becomes (is transformed into) t' if a certain condition C if verified. Term t represents a partial state of a global state S of the described system. The modification of the global state S of the system to another state S' is realized by the parallel rewriting of one or more terms that express the partial states. The distributed state of a concurrent system is represented as a term whose sub-terms represent the different components of the concurrent state. The concurrent state's structure can have a variety of equivalent representations because it satisfies certain structural laws (equivalence class). Therefore, we can see the constructed configurations by a binary operator applied to binary sets:

```
1. sort Configuration .
2. sort Object .
3. sort Msg .
4. subsort Object < Configuration .
5. subsort Msg < Configuration .
6. op null : -> Configuration .
7. op_ _ : Configuration Configuration -> Configuration [assoc comm id : null] .
```

Figure 5: Example of a portion of the Maude program

The program slice illustrated in figure 5 gives a definition of three types: *Configuration*, *Object* and *Msg*. In lines 4 and 5, *Object* and *Msg* are sub types of *Configuration.* Objects and messages are in fact multi-set configuration singletons. More complex configurations are generated from the application of the union on these multi-set singletons (objects and messages). Where there is neither floating messages nor live objects, we have in this case an empty configuration (line 6). The construction of a new configuration in terms of other configurations is done with line 7's operation. We can note that this operation has no name and that the two sub lines indicate the positions of two parameters of configuration type. This operation, which is the multi-set union, satisfies the structural laws of association and of commutation. It also possesses a neutral element null. For example, if we have a message *M1* that represents a configuration, and an object $<O : C|atts>$ (please note that *O* is an object's identifier, *C* the class to which it belongs and atts is the list of its attributes) that represents in itself another configuration, then we can construct another configuration in terms of those two configurations: $M1 < O : C \mid atts >$. This one is equivalent to the configuration $< O : C \mid atts > M1$ because the __ __ operation is commutative.

Maude

Maude is a specification and programming language based on rewriting logic [Mes92b, Man99, McC03]. In Maude language, two levels of specification are defined [McC03]. The first level is related to the specification of the system whereas the second carries on the specification of the properties.

System Specification

This level is based on rewriting theory. System modules mainly describe it. For a good modular description, three types of modules are defined in Maude. The *functional* modules allow defining data types and their functions through equations theory. Figure 6.a describes the functional module *Nat* specifying the natural numbers. This module is imported in the module *FACT* (figure 6.b) to calculate the factorial of natural numbers.

```
fmod NAT is
sorts Zero NzNat Nat .
subsort Zero NzNat < Nat .
***Constructors
op 0 : -> Zero .
op s_ : Nat -> NzNat . ***Successor
….
endfm
```

(a)

```
fmod FACT is
Including NAT .
op _! : Nat -> NzNat .
var N : Nat .
eq 0 ! = 1 .
eq (s N) ! = (s N) * N !.
endfm
```

(b)

Figure 6: Functional Modules Nat et FACT

The *system* modules define the dynamic behaviour of a system. This type of module extends the functional modules by the introduction of rewriting rules. This type of module offers a maximal degree of concurrency. Finally, the *object-oriented* modules, which can be reduced to system modules. Object-oriented modules explicitly offer the advantages of the object paradigm. In relation to the system modules, the object-oriented modules offer a more appropriate syntax to describe the basic entities of the object paradigm, like, among others, objects, messages and configurations. Only one rewriting rule allows expressing, at a same time: the consumption of certain floating messages, the sending of new messages, the destruction of objects, the creation of new objects, state change of certain objects, etc.

```
mod BANK-ACCOUNT is
protecting INT .
including CONFIGURATION .
op Account : -> Cid.
op bal :_ : Int -> Attribute .
ops credit debit : Oid Nat -> Msg .          ***[1]
var A  : Oid .
vars M N : Int .

rl [credit] :                                ***[2]
  < A : Account | bal : N >  credit(A, M)
  => < A : Account | bal : N + M  > .

crl [debit] :
  < A : Account | bal : N >  debit(A, M)
   => < A : Account | bal : N - M  >
   If N >= M .
endm
```

(a)

```
(omod BANK-ACCOUNT is
protecting INT .
class Account | ba : Int .
op bal :_ : Int -> Attribute .
msgs credit debit : Oid Int -> Msg .

var A : Oid .
vars M N : Int .

rl [credit] :
  < A : Account | bal : N >  credit(A, M)
  => < A : Account | bal : (N + M)  > .

crl [debit] :
  < A : Account | bal : N >  debit(A, M)
   => < A : Account | bal : (N – M)  >
   If N >= M .
endom)
```

(b)

Figure 7: The same *BANK-ACCOUNT* module in system module and OO module forms.

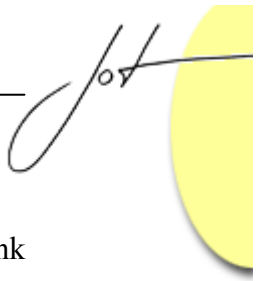


Figure 7.a illustrates the use of a system module *BANK-ACCOUNT* to define a bank account object A and the two operations that can affect its content *credit* and *debit* (line 1) by executing the rewriting rule defined in this module. Whereas figure 7.b describes the same module *BANK-ACCOUNT* with a more appropriate object-oriented syntax. We note for example, after executing the rule [credit] (line 2), the message *credit(A, M)* is consumed and the account's content is changed.

Properties Specification

This specification level defines the system's properties to be checked. The system is of course described using system module. By evaluating the set of states that are reachable from an initial state, Model Checking allows checking a given property in a state or a set of states. Property is expressed in a temporal logic LTL (*Linear Temporal Logic*) or in BTL (*Branching Temporal Logic*). Model Checking supported by the Maude's platform uses LTL logic essentially for its simplicity and the well-defined procedures of decision it offers (for more details, see [McC03]).

```
fmod LTL is
...
*** defined LTL operators
 op _->_ : Formula Formula -> Formula . *** implication
 op _<->_ : Formula Formula -> Formula . *** equivalence
 op <>_ : Formula -> Formula .      *** eventually
 op []_ : Formula -> Formula .      *** always
 op _W_ : Formula Formula -> Formula .  *** unless
 op _|->_ : Formula Formula -> Formula . *** leads-to
 op _=>_ : Formula Formula -> Formula . *** strong implication
 op _<=>_ : Formula Formula -> Formula . *** strong equivalence
...
endfm
```

Figure 8: A module in Maude implementing the operators of LTL logic.

In a predefined module *LTL*, we find the definition of operators for the construction of a formula (property) in linear temporal logic. In figure 8, and by hiding certain implementation details, we find part of LTL operators in Maude syntax. 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). This operator is applied to a formula to give a new formula.

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

Figure 9: A module in Maude implementing the satisfaction operator of a formula in a state.

Furthermore, we need an operator indicating if a given formula is true or false in a certain state. We find such an operator (|=) in the predefined module *SATISFACTION* (figure 9).

The state *State* is generic. After specifying the system's behavior 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 one describing the system's dynamic aspect and the module *SATISFACTION*. Assume for example that *M-PREDS* (figure 10) represents the name of the module describing the predicates on the system's states. *M* is the name of the module describing the system's behavior. The user must specify that the selected state (configuration chosen in this example) for its own system is sub-type of the sort State. At the end, we find the module *MODEL-CHECKER* (figure 11) that 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 since 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
```

Figure 10: A module in Maude containing the predicates defined by the user about a system described by a module M.

```
fmod MODEL-CHECKER is
including SATISFACTION .
…
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult .
…
endfm
```

Figure 11: A module containing the services offered to the user by Maude Model Checking

5 TRANSLATING AUML DIAGRAMS IN MAUDE

We use, in what follows, several examples to illustrate the defined process to support the translation of the AUML specifications in Maude. The hierarchical vision in three layers for describing a system in AUML can be captured by Maude. Using the Maude language, we can define a module whose behavior can be extended by another module.

Translation of the template concept

We define a *TEMPLATE* module in Maude for modelling the template concept of AUML. The different constructions *Act*, *Role* and *DeadlineType* allow us to represent respectively types of acts, of roles as well as the concept of deadline defined in AUML. For example, we translate the AUML template given by the figure 1 in the Maude

functional module described by figure 12. This last defines the *Initiator* and *Participant* roles that are, in fact, constants of type Role. We define, furthermore, *call-for-proposal* like a constant of the proposed type Act.

```
fmod TEMPLATE-FIPA-CONTRACT-NET-PROTOCOL is

  sorts Act Role DeadlineType .
  ops Initiator Participant  : -> Role .
  op Deadline : -> DeadlineType .
  ops call-for-proposal refuse  not-understand propose reject-proposal
       accept-  proposal cancel inform  :  -> Act .
endfm
```

Figure 12: Modeling *Template* in Maude

This module is generic and remains open to extension. We extend its behaviour by adding a description of the behaviour of its operations in another module that will implement the concept of *Package* of AUML. Before detailing the translation of the package, we give the translation of class and role.

- Basic object-oriented concepts: The basic concepts of the object paradigm (class, object, inheritance, and message) correspond naturally to the defined equivalent concepts in Maude.
- Role: A role in AUML reflects, in fact, an agent's particular behaviour. This behaviour exhibited by the agent controls the type of sent or received messages by an agent. An agent's role is described by a set of rewriting rules. Each time that an agent plays a role, we orient this agent to only execute the rewriting rules of this role. In addition of rewriting rules, we use an attribute to describe a role explicitly. The definition in AUML agent-name/role: class will be described in Maude as follows: <agent-name: class | Play-role: Role,…>, where agent-name is an agent's identifier. We define Role like an enumerated type containing roles values for this agent and Play-role like an attribute that contains the role played by the agent, at a given moment.

Translation of the package concept

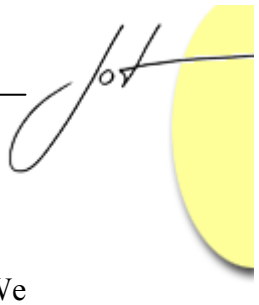
A *Package* in AUML can be described like a module in Maude. This module can encapsulate, as in AUML, a description of aggregation of interaction sequences. Every interaction in AUML, can be described by a rewriting rule. We give in figure 13, a part of an object-oriented module in Maude to describe the template package. In this module, we find the definition of the class *Agent* (line [1]). This class is characterized by the presence of the three attributes: *Play-role* of *Role* type denoting an agent's role, *MBox* of *MailBox* type serving to contain proposals to come, *AcqList* of *AcquaintanceList* type containing the list of the agent's acquaintances and the *State* attribute of *AgentState* type that is a enumerated type containing the set of own state values for an agent. We define the form of the message allowing the exchange of information between Initiator and Participant.

ComingMsg (line [2]) has three parameters expressing in the order the agent sender of the message, the agent receiver and the communication act.

```
(omod PACKAGE-FIPA-CONTRACT-NET-PROTOCOL is

 extending TEMPLATE-FIPA-CONTRACT-NET-PROTOCOL .
  …
 class Agent | Play-Role : Role, MBox : MB, AcqList : AcquaintanceList, State : AgentState .   *** [1]

 msg ComingMsg : Sender Receiver Act -> Msg .                                    *** [2]
 vars I  P : Aoid .                                                              *** [3]
 …
crl [IsendmsgP]:   < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : ACL, State : StartI >          ***[4]
                 =>
                     ComingMsg(I, HeadA(ACL), call-for-proposal)
                    < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : TailA(ACL), State : StartI >
                   if ACL =/= EmptyacquaintanceList .

rl [PreceiningmsgI]:  ComingMsg(I, P, call-for-proposal)                         ***[5]
                    < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : StartP >
                 => Execute(P, DecisionProcess)
                    < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP > .

crl [Pdecision1]:  Execute(P, DecisionProcess) Event(P, Cond)                    ***[6]
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
               => ComingMsg(P, I, propose)
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : WaitP >
                 if Cond = IsAccepted .

crl [Pdecision2]:  Execute(P, DecisionProcess) Event(P, Cond)                    ***[7]
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
               => ComingMsg(P, I, notunderstood)
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : WaitP >
                 if Cond = NotClear .

crl [Pdecision3]:  Execute(P, DecisionProcess) Event(P, Cond)                    ***[8]
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
               => ComingMsg(P, I, refuse)
                 < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : FailureP >
                 if Cond = IsRefused .
…
endom)
```

Figure 13 : Example of modeling of *Package* in Maude

The rule in this figure (line [4]) describes the sending of message *call-for-proposal* on behalf of the agent *I* in the *Initiator* role to all its acquaintances (agents playing the *Participant* role). Note that the variable *I* and *P* are *Aoid* type (line [3]) representing the identification space of the class *Agent* and the *Sender* and *Receiver* types are sub-types of *Aoid*. The execution of this rule requires the presence of the Initiator agent (left part of the rule) and gives as a result the agent itself and the creation of the *ComingMsg(P, I, Propose)* (right part of the rule). During each iteration, a *ComingMsg(P, I, Propose)* message is sent to one of participant agents . The execution of the rule stops when each participant appears in the list of acquaintances receives such a message.



Description of the agents' interaction modes in Maude

The three interaction modes defined in AUML (figure 2) are all supported by Maude. We use only one rewriting rule (see figure 14) to describe the interaction form that is in the diagram (3.a).

```
rl [L] :    < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...>
       => < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...>
            M1(CA-1, ...)    M2(CA-2,...)  ... Mn(CA-n,....)
```

Figure 14: Modelling of the concurrent threads of interaction in Maude.

This clearly indicates the spontaneous sending of the *Mi* messages containing as parameters the communication acts *CA-i.* In the case of ' inclusive or ' and of *'exclusive or',* a "standard" evaluation strategy doesn't exist allowing to choose an alternative among several. This makes it difficult (and impossible in some cases) to propose a precise translation of these two interaction modes in Maude. Their translation remains an open issue. However, thanks to the flexibility of the Maude language, we can recommend some solution directions. To describe the form of interaction concerning the *'inclusive or'*, we propose one of the following solutions to capture this concept in Maude, the use of m rewriting rules ($m \leq n$) of the form (figure 15).

```
crl [Li] : < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...>
 => < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> M1(CA-i1, ...) M2(CA-i2, ...)
                                                                  ...  Mk(CA-ik, ...)  if Ci  .
```

Figure 15: Modelling of the interaction mode concerning 'inclusive or' in Maude using the conditions.

In fact, *i =1,..., m* and *CA-i1 CA-i2... CA-ik* are communication acts among *CA-1 CA-2... CA-n.* They present acts that must appear spontaneously. The condition *Ci* validates the *Li* rule. Therefore, it controls its execution. Instead of using conditions, another considered alternative (figure 16) consists of using a common message to all *Li* rules, in the following manner:

```
rl [Li] : < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> M(Vi)
=> < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> M1(CA-i1 , ...)  M2(CA-i2, ...)
                                                                  ...  Mk(CA-ik, ...)  .
```

Figure 16: Modelling of interaction mode concerning 'inclusive or' in Maude using the messages.

In the context of this solution, only one instance of the *M* message generated in advance allows to launch the execution of only one rewriting rule among the *Lis*. Values *Vi* of parameters of *M* allow selecting the *Li* rule among the other rules. Let's note that values of parameters of *M* are unique for every rule.

The *'exclusive or'* is described in Maude in a similar way to the one of the previous case. In this case, we adopt the solution based on conditions and we get n-rules. The form of these rules is described in figure 17.

```
rl [Li] : < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...>
 => < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> MI(CA-i)   if Ci  .
```

Figure 17 : Modeling of interaction mode concerning 'exclusive or' in Maude using the conditions.

i =1, ..., n. If *CA-i* is the chosen message to be sent, it is therefore necessary that the condition *Ci* be verified for rewrite the appropriate rule Li, while all other conditions *Cj* (*j =1,.. ,n, j≠i*) are false. The solution based on messages in the interaction mode *'inclusive or'*, can be adapted also to implement the *'exclusive or'*. In this case, the execution of every *Li* rule consists in creating only one act *CA-i* (see figure 18).

```
rl [Li] : < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> M(Vi)
 => < A1 :  C1 | PlayRole : R1, ...> < A2 : C2 | PlayRole : R2, ...> Mi(CA-i , ...)  .
```

Figure 18: Modelling of interaction mode concerning 'exclusive or' in Maude using the messages.

In the example of figure 13, after receiving the message *ComingMsg(I, P, call-for-proposal)* (line [5]) the participant launches its decision process by generating an *Execute(P, DecisionProcess)* message. This message is common to three rewriting rules (lines 6, 7 and 8). One of these rules, solely, will be executed. The message *ComingMsg(I, P, call-for-proposal)* will be consumed after the rewriting of this rule, blocking so the rewriting of the two another ones.

Agent's internal behaviour

This behaviour is described by a state-transitions system. This system presents, in fact, a particular case in Maude [Mes03].

6 VERIFYING INTERACTION PROTOCOL USING MAUDE

After generating an AUML-Maude model, by translating AUML concepts in Maude, we can now move on to the application of model-checking. We start by the predicates specification (see Figure 19) related, on one hand, to the agents' internal behaviour and, on other hand, to their collective behaviour.

Individual behavior's properties

The verification of collective behaviour of several agents passes, to our opinion, first by the verification of the individual behaviour of agents implied in the realization of the task to accomplish. We adopted an incremental approach. We define in a first time the properties to be verified for different agents. We propose, in what follows, two properties (properties 1 and 2) related to the internal behaviour of the agent Initiator and two properties (properties 3 and 4) related to the Participant's behaviour. We first specify that all *Configurations* are *State*.

- Property 1:
  []~(NotEvaluation-At-AnyCase(Initial-State)) : This property expresses the fact that we never have *NotEvaluation-At-AnyCase* from the initial state (initial configuration) *initial-state.* From an initial state, the agent *Initiator* can launch its evaluation process in all cases (for example, if no proposition is received, if the deadline is not expired again, etc.).
- Property 2:
  []~(NotWill-be-Chosen-Participant-Proposing-MinimalPrice(Initial-State)) : this property expresses the fact that we never have NotWill-be-Chosen-Participant-Proposing-MinimalPrice from the initial state Initial-Sate. In other words, the agent Initiator always choose the Participant proposing the minimal price corresponding to the proposal being at the head of proposals list in the Initiator's mail box (Initiator adopts an increasing sorting of the proposals according to the price).
- Property 3:
  <> (Participant-Refuse-To-Propose(P)): this property expresses the fact that the participant *P* can eventually refuse to propose.
- Property 4:
  [](Participant-Refuse-To-Propose(P) -> O (Participant-In-Final-State(P))): this property expresses the fact that if the participant *P* refuses to propose, it always passes directly to a final state (see figure 4.b).

Collective behaviour properties

Once the properties related to the internal behaviour for each agent are verified, we consider the verification of collective behaviour's properties. The invalidity of agent's internal behaviour can have a repercussion on the collective behaviour. Two collective behaviour properties have been developed : the system's state coherency and the correct termination of the system (the system's final state must normally be reached).

- Property 5 : **(Coherency)**
  []~(InCoherent-System-State(Initial-State)) : Starting from an initial state *Initial-State*, this property expresses the fact that the system never falls in an incoherent state. All reachable states from such an initial configuration are always coherent.
- Proprerty 6 : **(Termination)**
  []~(Anamalous-End-Of-System(Initia-State)) : this property expresses the fact that from the initial state *Initial-State*, the system always finishes in a normal state.

Figure 19, represents the *PACKAGE-FIPA-CONTRACT-NET-PROTOCOL-PREDICATS* module containing the definition of different properties for agents' behaviour (internal and relational) (lines (1-9) and the equations implementing these properties (lines (10-17)). The three properties related to internal behaviour of agents *Initiator* and *Participant* are described in this module by the operations of the lines (1-3). For the first property (*NoEvaluate-At-AnyCase*) (line 1), we define a configuration (*State*) composed of a message and a state of the agent Initiator (line 10). Such a property is valid in the case where the deadline (here *DecemberEnd*) is not even expired or that no proposition has

been received by the Initiator (empty Mailbox). It allows verifying that the evaluation process triggering of the proposals only makes itself in the case where there are proposals, after the expiration of the deadline. The second property (*NotChoose-Participant-Proposing-MinimalPrice*) (line 2) expresses the fact that at the end of the evaluation process, the agent Initiator chooses the first *Participant* proposing without taking in account the price (line11). Such a property is valid in the case where the price of the first proposal is not the minimal price. The third property of the line 3 (*Participant-Refuse-To-Propose*) expresses the fact that from its making decision state (*OfferEvaluationP*), the Participant launches its decision process from which results a refusal to propose (line 12). The fourth property doesn't appear in the figure 19. It is, in fact, composed of two other properties (lines 3 and 9).

The properties related to the relational behaviour between agents are defined in figure 19, by the operations of the lines 4 and 5. The predicate *InCoherent-System-State* (line 4) is valid if in any configuration, the Initiator's current state doesn't correspond to the Participant's current state (line 13). In order to respect the interaction protocol used by agents, we must recognize their synchronization points. For this, we use the function *CorrespondingState* which have as parameters a communication state (sending message state) of an agent *i* and a waiting state of an agent *j*. Such a function returns the value true if the states correspond to one another, false otherwise. The *Anamalous-End-Of-System* predicate (line 5) is valid if the Initiator finishes in state *success* whereas there is a number of Participants different of one that also finish in state *success* (line 14) (we suppose that in the case where there are some proposals, the Initiator only chooses one participant and sending to it accept-proposal). Note that the second property doesn't constitute a particular case of the first, since a final state is not a communication state, nor a waiting state. The four properties of the figure 19 (lines (6-9)) are defined to be used jointly in order to verify the agents' relational state. The two properties *Initiator-In-OffEval-State*, and *Initiator-In-CommDec*-State indicate if the *Initiator* is in the *OfferEvaluation* and *CommitmentDecision* states respectively. *Participant-In-Waiting-State* is valid in the case where the *Participant* is in waiting state. The other property *Participate-In-Final-State* indicates if the agent *Participant* is in its final state.

```
(omod PACKAGE-FIPA-CONTRACT-NET-PROTOCOL-PREDICATS is
protecting PACKAGE-FIPA-CONTRACT-NET-PROTOCOL . including SATISFACTION .
subsort Configuration < State .

op NotEvaluation-At-AnyCase : Configuration -> Prop .                          ***[1]
op NotWillbe-Chosen -Participant-Proposing-MinimalPrice : Configuration -> Prop .  ***[2]
op Participant-Refuse-To-Propose : Oid -> Prop .                               ***[3]
op InCoherent-System-State : Configuration -> Prop .                           ***[4]
op Anamalous-End-Of-System : Configuration -> Prop .                           ***[5]
op Initiator-In-OffEval-State : Oid -> Prop .                                  ***[6]
op Initiator-In-CommDec-State : Oid -> Prop .                                  ***[7]
op Participant-In-Waiting-state: Oid -> Prop .                                 ***[8]
op  Participant-In-Final-State : Oid -> Prop .                                 ***[9]

********* Confirmation that the evatuation process not lauch at any time and in any case****
ceq    Event(I, ExipredTimeOut(D))                                             ***[10]
       < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : ACL, State : WaitI >
 |=   NotEvaluation-At-AnyCase( Event(I, ExipredTimeOut(DecemberEnd))
          < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : ACL, State : WaitI > ) = true
      if (D =/= DecemberEnd) or (MB == EmptyMailBox) .

******Confirmation that the choiced participant is who having the minimal price****
ceq   < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : SI >   ***[11]
      ComingMsg(I, GetChoicedParticipant(FrontQ(MB)), accept-proposal)
 |=  NotChoose-Participant-Proposing-MinimalPrice(
               < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : SI >
               ComingMsg(I, GetChoicedParticipant(FrontQ(MB)), accept-proposal) ) = true
      if MinimalPrice(MB) =/= GetPrice(FrontQ(MB)) .

******Confirmation that the participant eventually refuses to propose**************
ceq   < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP >   ***[12]
      Execute(P, DecisionProcess) Event(P, Cond)
  |= Participant-Refuse-To-Propose( P) = true
     if (SP = OfferEvaluationP) and (Cond = IsRefused) .

**********Confirmation that the system is always  in a coherent state************
ceq  < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : SI >   ***[13]
     < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP >
|=   InCoherent-System-State(
               < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : SI >
               < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP > ) = true
      if ( CorrespondingState(SI, SP) == false) ..

*********Confirmation that the end of the task is terminating in a coherent state*****
ceq    < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : (P1 : (P2 : P3)), State : SI >   ***[14]
       < P1 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP1 >
       < P2 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP2 >
       < P3 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP3 >
 |=   Anamalous-End-Of-System(
       < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : SI >
       < P1 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP1 >
       < P2 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP2 >
       < P3 : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP3 >   ) = true
      if (SI == SuccessI) and SuccessNumber((SP1 :: (SP2 :: SP3))) =/= 1) .

**********Confirmation that the Initiator is in OffEvauationI state******
eq   < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : OfferEvaluationI >   ***[15]
  |=   Initiator-In-OffEval--State(I ) = true .

**********Confirmation that the Initiator is in  CommitmentDecison  state******

eq   < I : Agent | PlayRole : Initiator , MBox : MB, AcqList : ACL, State : CommitmentDecision >
  |-    Initiator-In-CommDec-State(I ) = true .

********* Confirmation that theParticipant is inWait state ******                ***[16]
eq  < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : WaitP >
  |=  Participant-In-Waiting-state (P ) = true

********* Confirmation that theParticipant is in its final state ******          ***[17]
ceq  < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : SP >
  |=  Participant-In-Final-State(P ) = true
     if (SP == SuccessP) or (SP == FailureP) .
endom)
```

Figure 19: A module in Maude defining properties on the AUML-Maude model

Application of Maude model-checker

We used the Maude function *modelcheck* for verifying some properties related to the agent's internal behaviour and to the relational behaviour between agents. Figure 20 illustrates a part of the code we developed. We visualize precisely the module *PACKAGE-FIPA-CONTRACT-NET-PROTOCOL-CHECK* containing the definition of six initial configurations and the application of the function *modelcheck* on these configurations. We propose the initial states (*Initial1, Initial2* and *Initial3*) to show how the Maude model-checker behaves during verification of an agent's individual behaviour and (*Initial4, Initial5* and *Initial6*) for the one of their collective behaviour (see figure 20).

Verification of individual behaviour's properties

In this section, we consider the verification of some properties related to individual behaviour of the agents Initiator and Participant.

To verify that the agent Initiator doesn't launch its evaluation process that after the expiration of the deadline and that if it received proposals of the participants before, we propose the following property:

[]~(NotEvaluation-At-AnyCase (initial1)).

This property expresses the fact that from an initial state *initial1* (see figure 20), representing the arrival of an event of expiration of the deadline when the *Initiator* is in waiting with an empty mailbox, this last never launches its evaluation process. The launching of the property verification is illustrated by the figure 20. The result obtained of this verification is a *counter-example* (figure 21). Indeed, in this case the *Initial1* configuration itself represents a counter-example knowing that the mailbox is empty. This means that the second part of the rule condition of the line 10 (figure 19) is verified.

To verify that the agent Initiator always chooses the agent Participant proposing the minimal price, we propose the verification of the following property:

[]~(NotWill-be-Chosen-Participant-Proposing-MinimalPrice(initial2))

This property is interpreted in the following way: leaving from the *Initial2* state (figure 20), the *NotWill-be-Chosen-Participant-Proposing-MinimalPrice* property is always false. Such a state represents agent *Initiator* in its evaluation state with a mailbox containing proposals in an increasing order according to the price. In other words, agent *Initiator* always chooses the *Participant* proposing the minimal price corresponding to the proposal being at the head of the proposals list in the Initiator's mail box (the *Initiator* adopts an increasing sorting of the propositions according to the price). The launching of the verification of this property is illustrated by the figure 20. The result of this verification is *true* (figure 21). The validity of this property indicates that the price

proposed by the participant, being at the head of the mail box (while using the *GetPrice(FrontQ((MB))* function (line 11 of the figure 19)), is always the lowest price being in the mail box MB (By using the *MinimalPrice(MB)* function (line 11 of the figure 19)).

To verify that the agent Participant can eventually refuse to send a proposal to the Initiator, we propose the following property:

$$<> \textit{(Participant-Refuse-To-Propose(P))}$$

This property expresses the fact that from the initial state *Initial3* (figure 20), the participant refuses eventually to propose. Such a state describes the launching of the decision process by the agent *Participant* being in its taking decision state. The launching of the verification of this property is illustrated by the figure 20. The result of this verification is *true* (figure 21).

As mentioned in the figure 4.b, each *Participant* that refuses to propose passes to its final state *Failure.* To verify this behaviour, we propose to verify the following property:

$$[]\textit{(Participant-Refuse-To-Propose(P) -> O(Participant-In-Final-State(P))}$$

Starting from the initial state Initial3, if it refuses to propose, the participant passes to a final state. The result of this verification is *true*.

The strategy we adopt for verifying agents interaction protocols allows limiting the space of localization of anomalies. It consists of, in fact, an incremental verification process. We start with the verification of different properties related to the internal behaviour of every agent, and then we pass to the system's global behaviour.

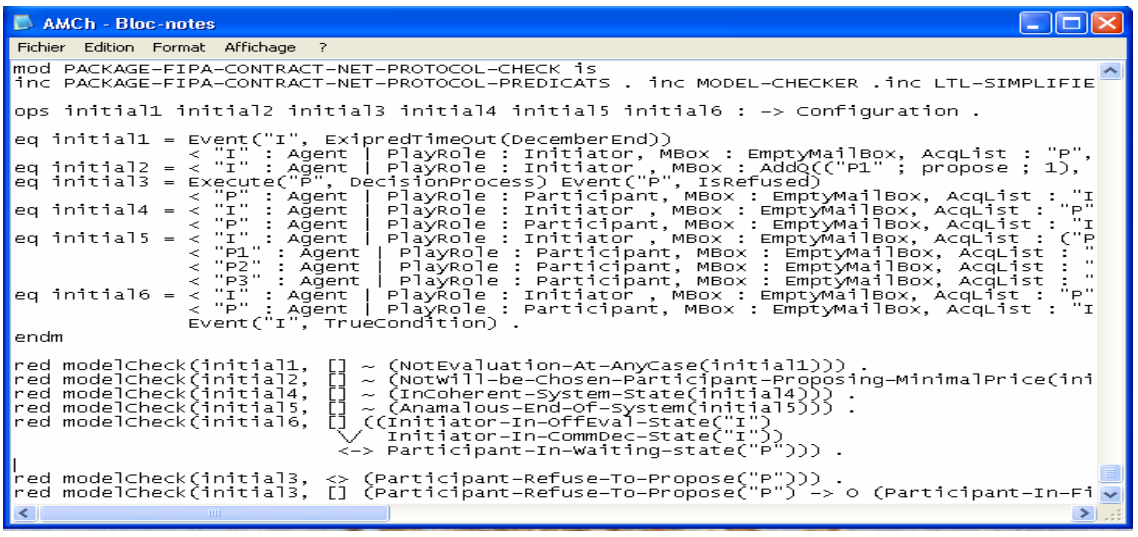


Figure 20: A part of code (example of initial states and launching of properties verification) of AUML-Maude model.

Verification of collective behavior's properties

After verifying the properties related to individual behaviour of different agents, we move on to the verification of the system's collective behaviour.

To verify the system coherency, we propose the following property:

[]~(InCoherent-System-State(initial4))

This property expresses the fact that from the initial state *Initial4* (figure 20), we never end into a system's incoherent state. Such a state represents the agents *Initiator* and *Participant* in their initial states with empty mailboxes. Starting from *Initial4*, all reachable states from such an initial configuration always are coherent. The obtained result is the value *true* (figure 21). This expresses the fact that there is always a correspondence between interacting agents states (using the function *CorrespondingState* (line 13 of figure 19)).

In the same way, to verify that the system finishes in a normal state, we propose the following property:

[] ~(Anamalous-End-Of-System(initia5))

This means that from the initial state *Initial5* (figure 20), we never have *Anamalous-End-Of-System*. It means that always the system end is normal. The result of the verification of this property from the state *Initial5* is a *counter-example* (figure 21). This counter-example asserts that *Initial5* itself is abnormal. The *Initiator* is in *Success* state and there are two Participants being on their turn in *Success* state.

We want in this case to illustrate a particular case of the system's state coherency that seems important to us. Furthermore, if the *Initiator* is in one the following states: the proposals evaluation or engagement decision making, the *Participant* must be in waiting state and vice-versa. For it we propose the following property:

[]((Initiator-In-OffeEval--State(I) \/ Initiator-In-CommDec-State(I)) <-> Participant-In-Waiting-state(P)).

The launching of the verification of this property is illustrated by figure 20. The result of the verification of this property from the state *Initial6* is the value *true* (figure 21). Such a state describes the agents *Initiator* and *Participant* in their initial states with the event initializing the interaction protocol. The validity of this property indicates that the correspondence between the states (*OfferEvaluation* and *CommitmentDecision*) of the agent *Initiator* and the waiting state (*Wait*) of the agent *Participant* is always respected.

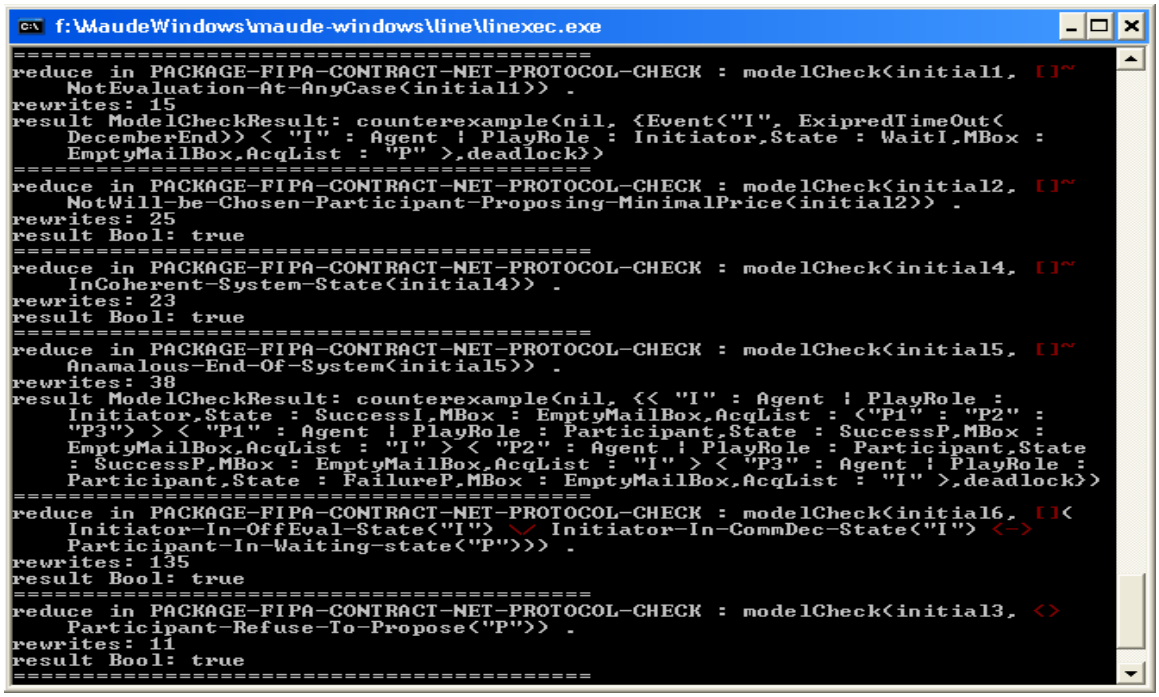


Figure 21: Properties verification results

7 Conclusions and future work

Specification and verification of interacting agents' behaviour represent ones of the main issues in the domain of multi-agents systems. These last years, several works carrying on the specification and the verification of MAS have been achieved. We mention, among others, the work of Wooldridge et al. in [Woo02]. They presented an imperative multi-agent programming language called MABLE, and a formal semantics for this language in terms of a BDI logic called LORA. They also described an implementation of this language, and described how claims about MABLE systems, expressed in a quantified linear temporal BDI logic called MORA, can be automatically checked by translating them into the form used by the SPIN model checking system. Furthermore, Bordini et al. [Bor03] proposed AgentSpeak(F), a variation of the BDI logic programming language AgentSpeak(L) intended to permit the model-theoretic verification of multi-agent systems. These two approaches nearly adopt the same way. They implement MAS using MABLE and AgentSpeak languages respectively, and then translate this implementation in PROMELA. The result of such a translation will be used by the model-checker SPIN to check claims about these systems. Although AIP represents an interesting part of MAS' infrastructures, these works don't deal with the problem of verification and proof of AIP's properties [Gio04].

We presented in this paper an approach to formally specifying and verifying agents interaction protocols (AIPs) described by using AUML formalism. Although AUML is the best-known language [Hug02], it only offers a semi-formal specification of interactions. This weakness can lead to several incoherencies in the description of a

MAS' behaviour. Our approach is based on the formal and object-oriented language Maude. It integrates at the same time the description power (specification and programming), the possibility of simulation for validation and in particular the formal verification based on the model-checking techniques. Compared to other model-checkers, like SPIN, Maude is effectively more expressive. Using Maude, one can easily specify different kinds of concurrent systems in it and can also reason about those specifications using other formal methods and tools [Eke03]. Furthermore, Another advantage of Maude is that integration of model checking with theorem proving techniques becomes quite seamless [Eke02].

This paper focused on the formal verification of interactions protocols described by AUML diagrams using Maude's model checker. Some properties have been verified, such as decision-making concerning the agent internal behaviour, and the verification of system coherency and termination related to the agents' collective behaviour. In a future work, we plan to exploit the power as well as Maude's flexibility to specify and to verify open multi-agents systems.

REFERENCES

[Ast98] E. Astesiano. UML as "Heterogeneous Multiview Notation. Strategie for a Formal Foundation". In L. Andrade, A. Moreira, A. deshpande, and S. Kent, editors, Proc. of the Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA'98) – Workshop on Formalizing UML. Why ? How ?, Canada 1998.

[Bak00] 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". First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000. FAABS 2000.

[Bau01] B. Bauer, J. P. Muller, J. Odell. ''Agent UML: A Formalism for Specifying Multiagent Interaction'', Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 91-103, 2001.

[Bor03] R. Bordini, M. Fisher, C. Pardavila and M. Wooldridge. "Model Checking AgentSpeak". AAMAS 2003, pp. 409-416, 2003.

[Cla02] E. M. Clarke, O. Grumberg, and D. A. Peled. ''Model Checking''. MIT Press, 2002.

[Cos99] R. Cost and al. "Modeling Agent Conversations with colored Petri Nets", Working Notes of the Workshop on Specifing and Implementing Conversation Policies, Autonomous Agents'99, seattle, Washington, May 1999.

[Eke02] S. Eker, J. Meseguer, A. Shridharanarayannan. ''The Maude LTL model checker''. In: proc. WRLA'02. Volume 71 of ENTCS., Elsevier (2002).

[Eke03] S. Eker, J. Meseguer, A. Shridharanarayannan. ‘’The Maude LTL model checker and its Implementation’’, T. Ball and SK Rajamani (Eds.): SPIN 2003, LNCS 2648, pp. 230–234, 2003.

[Fer04] S.N. Freund and S. Qadeer. ’’Checking Convise Specifications For Multithreaded Software’’. in Journal of Object Technology, vol 3, no 6, June 2004, Special issue: ECOOP2003 Workshop of FTfJP, pages 81-101.

[FIP02] Foundation for Intelligent Physical Agents. ‘’FIPA Contract Net Interaction Protocol Specification’’. December 2002.

[Gab04] C.Gabriel, Dorel Lucanu: ‘’Specification and Verification of Synchronizing Concurrent Objects’’. IFM 2004: 307-327.

[Gio04] L. Giordano, A. Martelli, C. Schwind. “Verifying Communicating Agents by Model Checking in a Temporal Action Logic”. José Julio Alferes, Joao Alexandre Leite (Eds.): Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings. LNCS 3229 Springer 2004, ISBN 3-540-23242-7.

[Gue03] Z. Guessoum. “Modèles et Architéctures d’Agents et de Systèmes Multi-Agents Adaptatifs”. Dossier d’habilitation à diriger des recherches de l’Université Pierre et Marie Curie. Décembre 2003.

[Hol97] G. Holzmann. “The Spin model checker”, IEEE Trans. on Software Engineering, 23(5):279–295, May 1997.

[Hug02] M.P. Huget. ‘’Model Checking Agent UML Protocol Diagrams’’. In ECAI Workshop on Model Checking Artificial Intelligence (MoChArt), Lyon, France, July 2002.

[Hug04] M.P. Huget and J. Odell, ‘’Representing Agent Interaction Protocols with Agent UML’’. In AAMAS’04, pp.1244-1245, New York, NY, USA, 2004.

[Kav03] K. kavi and al. ‘‘Extending UML for Modeling and Design of Multi-Agent Systems’’. Proc. of ICSE'03 Workshop on Software Engineering for Large Multi-Agent Systems (SELMAS'03), Portland, Oregon, May 3--4, 2003.

[Kri93] J.L. Krivine. “Lambda-calcul, types et modèles", Masson, Paris (1990). English translation : Lambda-calculus, types and models. Ellis Horwood (1993).

[Man99] M. Clavel and al. “Maude : Specification and Programming in Rewriting Logic”. Internal report, SRI International, 1999.

[Maz02] H. Mazouzi, A. F. Seghrouchni, and S. Haddad. ‘‘Open protocol design for complex interaction in multi-agent systems’’. In proceeding of the first international joint conference on Autonomous agent and multi-agents systems, pages 517-526. ACM Press, 2002.

[McC03] T. McCombs. “Maude 2.0 Primer, Version 1.0”. Internal report, SRI International, 2003.

[Mes90] J. Meseguer, “Rewriting as a unified model of concurrency”. In Proceedings of the Concur’90 Conference, Amsterdam, P. 384-400, LNCS Vol. 458, 1990.

[Mes92a] J. Meseguer, “Conditional Rewriting Logic as a unified model of concurrency”. Theoretical Computer Science, 1992.

[Mes92b] J. Meseguer, “A Logical Theory of Concurrent Objects and its Realization in the Maude Language”. In G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992.

[Mes00] J. Meseguer, “Rewriting Logic and Maude : a Wide-Spectrum Semantic Framework for Object-Based Distributed Systems” In S. Smith and C. L. Talcott, editors, Formal Methods for Open Object-Based Distributed Systems, FMOODS2000, 2000.

[Mes03] J. Meseguer, “Software Specification and Verification in Rewriting Logic”. In M. Broy and M. Pizka, editors, Models, algebras and logic of engineering software, pages 133-193. IOS Press, 2003. ISBN 1-58603-342-5.

[Mok04] F. Mokhati, N. Boudiaf, L. Badri et M. Badri. ‘’Generating Maude Specification from AUML Diagrams: Toward A Systematic Approach’’. In Proc. of CSITe-A04, Caire, Egypte, Decembre 2004.

[Mor05] M. MORGE. "Système dialectique multi-agents pour l’aide à la concertation". Thèse de doctorat. Ecole Nationale Supérieure des Mines. SAINT-ETIENNE. 20 juin 2005.

[Mul00] P.A. Muller et Nathalie Gaertner. “Modélisation objet avec UML”, Deuxième Edition 2000 Paris.

[Ode00] J. Odell, H. V. D. Parunak, B.Bauer, “Representing agent Interaction protocol In UML”, AAAI Agents 2000, Barcelone, 3-7 juin 2000.

[Ode01] J. Odell, H. V. D. Parunak, B.Bauer, “Representing agent Interaction protocol In UML”, Agent Oriented Software Enginering, Paolo Ciancarini and Michael Wooldridge (eds.), Springer-Verlag, Berlin, 2001, pp. 121-140.

[Olu05] A. Oluyomi and L. Sterling. ‘’A Dedicated Approach for Developing Agent Interaction Protocols’’. M.W.Barley and N.Kasabov (Eds.): PRIMA 2004, LNAI 3371, pp. 162-177, 2005.

[OMG01] The Object Management Group. ‘’OMG Unified Modeling Language Specification’’, version 1.3, March, 2001

[Pau03a] S. Paurobally, J. Cunningham, “Achieving Common Interaction Protocols in Open Agent Environments”, 2nd international workshop on Challenges in Open Agent Environments, AAMAS 2003, Melbourne, Australia 14-18th July 2003.

[Pau03b] S. Paurobally, J. Cunningham, and N. R. Jennings, “Developing Agent Interaction Protocols Using Graphical and Logical Methodologies”, in Proc.

of AAMAS03 PROMAS Workshop on Programming Multi-Agent Systems , 2003.

[Pau04] S. Paurobally, Cunningham J., and Jennings, N. R., "Verifying the contract net protocol: a case study in interaction protocol and agent communication semantics". In Proceedings of the 2nd International Workshop on Logic and Communication in Multi-Agent Systems , pages pp. 98-117, Nancy, France 2004.

[Reg99] G. Reggio and R. Wieringa. "Thirty one Problems in the Semantics of UML 1.3 Dynamics". In Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA'99) – Workshop "Rigorous Modeling an Analysis of the UML Challenges and Limitations'', 1999.

[Rao96] A. S. Rao. ''AgentSpeak(L): BDI agents speak out in a logical computable language''. In W. Van de Velde and J. Perram (eds), Proc. Seventh Workshop on Modelling AutonomousAgents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, number 1038 in LNAI, pages 42–55. Springer-Verlag, 1996.

[Sch99] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit, "Vérification de logiciels : Techniques et outils du model-checking", Vuibert, 1999.

[Thi05] Y. Thierry-Mieg."Techniques pour le model-checking de spécifications de Haut-niveau". Séminaire Méthodes de Conception, Vérification et Réalisation. Application à la Répartition et au Temps Réel, 21 Janvier 2005.

[Toi04] S. Toivonen. et al. «Using Interaction Protocols in Distributed Construction Processes». In Seruca, I., Filipe, J., Hammoudi, S., and Cordeiro, J. (Eds.): Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'04), Porto, Portugal, April 2004, pp. 344—349

[Tra99] E. Tranvouez, B. Espinasse, "Protocoles de coopération pour le réordonnancement d'atelier". in actes des journées francophones d'Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'99) à Saint-Gilles, île de la Réunion, novembre 1999, Gleizes J.-P., Marcenac P., Ed. Hermès, 1999.

[Ver03] A. Verdejo, I. Pita, and N. Marti-Oliet. ''Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic''. Formal Aspects of Computing, 14(3): 228-246, 2003. 26

[Woo00a] M. Wooldridge et al, "The gaia methodology for agent-oriented analysis and design". Autonomous Agent and Multi-aget Systems, 3(3):285-312, 2000.

[Woo00b] M. Wooldridge. ''Reasoning about Rational Agents''. The MIT Press: Cambridge, MA, 2000.

[Woo02] M. Wooldridge, M. Fisher, M.P. Huget and S. Parsons. "Model Checking Multi-Agent Systems with MABLE". In AAMAS'02, pp. 952-959, Bologna, Italy, 2002.

About the author

Farid Mokhati (Mokhati@yahoo.fr) is an assistant professor of computer science at the Department of Computer Science of the University of Oum El-Bouaghi in Algeria. He holds a Ph.D. in computer science (Distributed Artificial Intelligence) from the University of Annaba in Algeria. His main areas of interest include object and agent-oriented software engineering, and formal methods.

Noura Boudiaf (Boudiafn@yahoo.com) is an assistant professor of computer science at the Department of Computer Science of the University of Oum El-Bouaghi in Algeria. She holds a Ph.D. in computer science from the University of Constantine in Algeria. Her main areas of interest include object-oriented software engineering, Petri net analysis methods, and formal methods.

Linda Badri (Linda.Badri@uqtr.ca) is professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. She holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. Her main areas of interest include object and aspect-oriented software engineering, software quality attributes, maintenance, and web engineering.

Mourad Badri (Mourad.Badri@uqtr.ca) is professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, and formal methods.