JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 3, May-June 2010

A Tool for Specifying and Validating Agents' Interaction Protocols: From Agent UML to Maude

Farid Mokhati, Department of Computer Science, University of Oum-El-Bouaghi, Algeria
Brahim Sahraoui, Department of Computer Science, University of Sétif, Algeria
Soufiane Bouzaher, Department of Computer Science, University of Oum-El-Bouaghi, Algeria
Mohamed Tahar Kimour, Department of Computer Science, University of Annaba, Algeria

Abstract

To achieve the multi-agent systems' goals, agents interact to exchange information, to cooperate and to coordinate their tasks. Interaction is generally recognized as an important characteristic of multi-agent systems (MAS). The usual approaches to model agents' interactions consist of describing them as protocols [Hug04]. In the literature, several representation formalisms of agents' interactions have been proposed. AUML is one among the most used formalisms [Hug02]. However, AUML diagrams only offer a semi-formal specification of interactions. Indeed, the lack of formal semantics in AUML, can lead to several incoherencies in the description of a MAS' behaviour. We present, in this paper, a visual tool that essentially allows: (1) translating the description of agents' interaction protocols (AIP), specified by means of AUML formalism, in a Maude specification and, (2) validating the generated formal descriptions through simulation. Based on rewriting logic, the formal and object-oriented language Maude offers an interesting way for concurrent systems formal specification and programming. By an example of multi-agent systems interaction protocol, we illustrate the proposed translation and the developed tool.

1 INTRODUCTION

In recent years, agents' interaction specification has become a central reasearch field. In order to describe such interactions, several formalisms and languages have been emerged in the literature (CATN [Lem03], RCA [Tra01], AUML [Ode01, Hug04], etc). AUML (Agent Unified Modeling Language) is one among the most used formalisms [Hug02]. In fact, it represents the first emerged result 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 extends UML (Unified Modeling Language) [Mul00] for modeling agents and their interactions [Hug04, Pau03]. While adopting a layered approach, AUML offers the agent interaction protocol diagrams to represent agents' interaction on an abstract level. Agent interaction protocol diagrams are defined by the FIPA in AUML. The main advantage of modeling with AUML is its intuitive graphical representation of the architecture and processes. For the usage of the AUML within CASE tools, a well defined semantics is required. However, AUML diagrams only offer a semi-formal specification of interactions. The semantics of agent interaction protocols are usually defined by the semantics of sequence diagrams, which are usually ambiguous and vague. This weakness may generate several problems. Indeed, the lack of formal semantics in AUML [Ast98, Reg99, Mor05], can lead to several incoherencies in the description of a MAS' behavior.

In this context, the use of appropriated formal notations offers several advantages. Especially, it allows for producing rigorous and precise descriptions supporting efficient verification and validation process.

Based on a sound and complete logic, called the rewriting logic [Mes92], the Maude language [Cla99, McC03] seems to be in this context an interesting candidate. It offers, in fact, through its various constructions, an interesting way for concurrent systems formal specification and programming. It offers all the basic elements allowing specifying and verifying formally multi-agent interactions.

In this paper, we present a visual tool that allows designers to construct and describe agents' interaction protocols using AUML diagrams and to generate formal specification in Maude from such AUML interaction diagrams. Moreover, it allows for validating the generated specification by means of simulation techniques supported by Maude platform. In fact, the developed tool implements the approach proposed in our previous work [Mok07].

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 an overview of Maude. In section 5 we present the proposed translation process. We illustrate in section 6, the developed tool. Finally, in section 7, we give some conclusions and future work directions.

2 RELATED WORKS

Multi-agent System technology has impressively emerged as a new paradigm for software development. As autonomous software components, agents can interact through a standard protocol and collaborate with each other to achieve common goals. For capturing interaction protocols, many works adopted the widely-used Agent UML (AUML) notation.

In [Pad07] PDT (Prometheus Design Tool) [Tha05] has been extended with two significant new features: support for Agent UML interaction protocols, and code generation. PDT provides graphical support for the design phases of the methodology, allowing the designer to enter and edit diagrams and descriptors for entities. Once a design for an agent system has been developed, the design needs to be implemented. PDT allows generate skeleton code in the JACK agent-oriented programming language [Bus99]. However, it's so difficult to verify the correctness of the generated code because the specification of interaction protocols has been not developed in a formal way.

In [Aye08], L. Jemni et al. proposed an event-B based approach to reasoning about interaction protocols. The objective of this approach is to show how an event-B model can be structured from AUML protocol diagrams and then used to give a formal semantic to protocol diagrams which supports proofs of their correctness. For describing concurrent object-oriented models, event-B must use an intermediate object-oriented style specification notation [Edm08].

L. Kahloul et al. [Kah05] presented an approach allowing translating Agent UML description into a recursive colored Petri net (RCPN) model. The proposed approach aims to formalize agent interaction in MAS in order to facilate analysis and verifcation processes. However, the validation and verification of the generated specification requires the use of tools supporting RCPN formalism.

Of course, those works have considerably forwarded the domain by creating AUML diagrams and translating them into another form. However, it's important, in this context to use more formal notations which essentially allows for producing rigorous and precise descriptions supporting efficiently the verification and validation process.

We present, in this paper, a visual tool that allows the generation of a Maude formal specification from AUML diagrams. The developed tool essentially allows: (1) translating the description of agents' interactions, specified using AUML formalism, in a Maude specification and, (2) validating the generated formal descriptions through simulation. The formal and object-oriented language Maude, based on rewriting logic, supports formal specification and programming of concurrent systems [McC03, Eke02, Mes03, Cla05]. Maude is a multi paradigm language [Mes03, Cla05] that supports the semantics of concurrency (intra and inter-objects). Furthermore, the Maude language is supported by a tool, which allows validating the generated formal descriptions through simulation. Maude also integrates a model checker supporting the verification of Linear Temporal Logic (LTL) properties [Eke02, Mes03, Cla05].

3 AUML

The UML language [Mul00, OMG05] was widely used in the modeling and design of object-oriented systems (OOS). However, it is not adapted for the modeling 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 where agents are proactive and social [Ode00, Kav03]. To fill those weaknesses, FIPA and OMG established a cooperation to facilitate the penetration of the agent technology in the industry. The first result of that cooperation was the definition of the Agent UML language (AUML) [Ode00, Bau01]. It supports the specification of multi-agent interactions. To represent multi-agent interaction protocols, AUML adopts in fact an approach organized in three layers. It uses, in the first layer, packages and templates to represent the entire protocol. Sequence, collaboration, activity and state-transition diagrams are used to represent the interactions between agents. Furthermore, activity and state-transition diagrams are also used to capture the internal behaviour of agents (for further details, the reader is referred to [Ode00]). This paper presents an alternative in witch we use the concepts of packages and templates to represent an interaction protocol, as well as sequence diagrams to describe the interaction between agents and state-transition diagrams for the specification of the internal behaviour of an agent. The rest of AUML concepts will be considered in a future work.

3.1. Level 1: Representation of agents' interaction protocol

Figure 1 presents an interaction protocol between agents. It describes, using an AUML sequence diagram, the FIPA Contract Net protocol. As soon as it is called, the Initiator agent sends a call-for-proposal to a Participant agent. Before a given limit date (deadline), the Participant agent can send to the Initiator agent a proposal, refuse to send that proposal (refuse) or indicate that it did not understand properly (not-understood). The proposal formulated by the Participant agent can be accepted or refused by the Initiator agent. When it receives a positive answer to its proposal (accept-proposal), the Participant agent informs the Initiator agent of the execution of its proposition. However, the Initiator agent can cancel the execution of the proposition at any given time. Figure 1 also illustrates two fundamental concepts relative to that level:

- **Package:** a conceptual aggregation of interaction sequences. It allows treating the protocol as a reusable entity. The top left corner indicates that the protocol is a package.
- **Template:** represented by a box in the top right corner. The template concept allows a protocol described by a package to be personalized for similar problem domains.

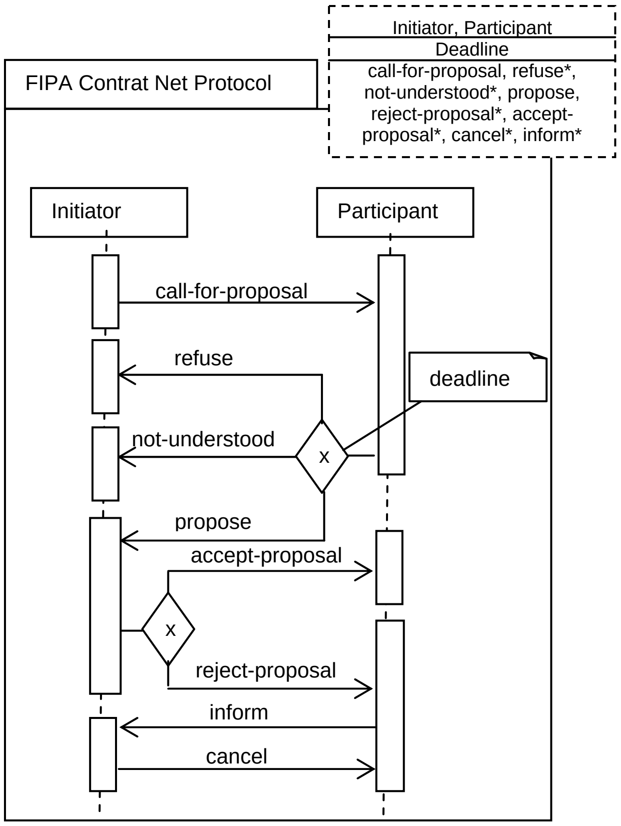


Figure 1. An interaction protocol expressed as a template package.

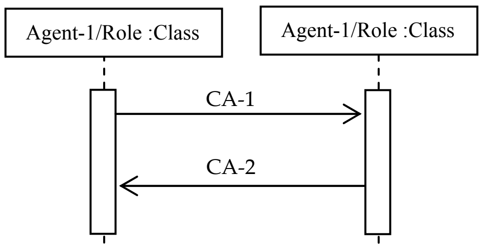


Figure 2. Basic format for agents communication.

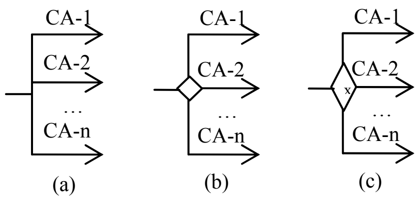


Figure 3. Recommended extensions that support concurrent threads of interaction.

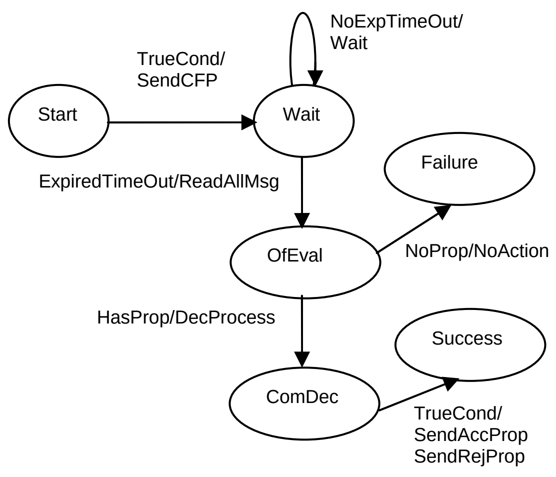
3.2 Level 2: Representation of the interactions between agents

Figure 2 shows the basic format of communication between agents. Instead of the message style defined in UML, AUML uses communication acts (CA).

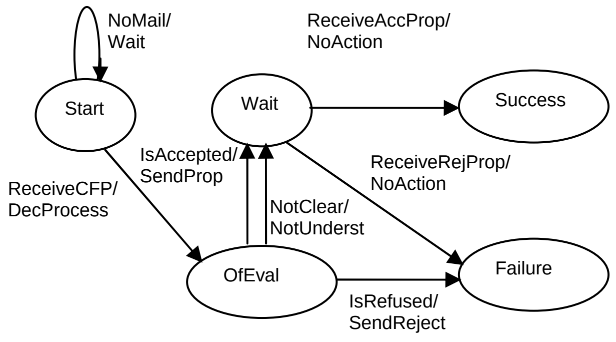
To support the description of concurrent interaction threads, AUML introduced three ways that allow expressing multiple threads (see figure 3). Figure 3(a) indicates that all CA-I (CA-1…CA-n) are transmitted concurrently. Figure 3(b) introduces a decision box indicating which the CAs (0 or more) will be transmitted. Figure 3(c) indicates that a unique CA is to be transmitted. Concurrent communication acts can be sent by a sender agent to an agent playing different roles. They can also be sent to different agents.

3.3 Level 3: Representation of the internal behaviour of agents

AUML offers two alternatives to represent the internal behaviour of an agent. The first consists on using state-transition diagrams (state-charts), and the second uses activity diagrams. As we mentioned earlier, we use state-charts in this paper for the modeling of the internal behaviour of an agent. Figures 4(a) and 4(b) present respectively the internal behaviour of agents Initiator (I) and Participant (P) of figure 1.



(a)



(b)

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

4 MAUDE

Maude is a language for specifying and programming systems and is based on rewriting logic [Cla99, Mes03]. It allows for describing easily the intra and inter-object concurrency. Furthermore, Maude has its own model-checker that is used in checking system's properties.

In rewriting logic, the logic formulas are called rewriting rules. They have the following forms: *R:[t]* → *[t']* or *R:[t]* → *[t'] if C*. Rule *R* indicates that term *t* becomes (is transformed into) *t'*. On its second form, a rule could not be executed except that 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.

Three types of modules are defined in Maude. The functional modules allow defining data types and their functions through equation theory. Figure 5(a) represents the functional module *Nat* specifying the natural numbers. The natural numbers sort hierarchy has top sort *Nat* and (disjoint) sub sorts Zero and NzNat. The sort *Nat* is generated from the constant 0 (of sort Zero) and the successor operator s_.

The *NAT* module is imported in the *FACT* module (figure 5(b)) to calculate the factorial of natural numbers using the operator *!*. Such an operator uses the successor operator *s_* in a recursive way.

```
fmod NAT is
sorts Zero NzNat Nat .
subsort Zero NzNat < Nat .
op 0 : -> Zero .
op s_ : Nat -> NzNat .
….
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 5. Functional Modules *Nat* and *FACT*.

The system modules allow defining the dynamic behaviour of a system. This type of module augments functional modules by the introduction of rewriting rules. A maximum degree of concurrency is offered by this type of module. Finally, object-oriented modules can, in fact, be reduced to system modules. This last type of module offers more explicitly the advantages of the object paradigm.

```
mod BANK-ACCOUNT is
protecting INT .
including CONFIGURATION .
op Account : -> Cid.
op bal :_ : Int -> Attribute .
ops credit debit: Oid Nat
   -> 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 .
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 6. The same *BANK-ACCOUNT* module in system module and OO module forms.

Compared to system modules, an object-oriented module offers a more appropriate syntax to describe the basic entities of the object paradigm such as objects, messages and configurations, for example. A single rewriting rule can express the consumption of certain floating messages, the sending of new messages, the destruction of objects, the creation of new objects, state changes of certain objects, etc. all at the same time. Figure 6(a) shows a system module called *BANK-ACCOUNT* to define a bank account object *A* and two operations that can influence its contents: *credit* and *debit* by the execution of the rewriting rules of that module. Figure 6(b) represents the same module *BANK-ACCOUNT* with a more appropriate object-oriented syntax. We can then conclude that after the execution of unconditional rule [credit], the *credit(A, M)* message is consumed and the content of the account is augmented. Also, starting the conditional rule [debit] needs the condition (N>=M) to be true before it can begin. The execution of a rule brings the consumption of message *debit(A, M)* and the reduction of the content of the account.

5 TRANSLATION PROCESS

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.

5.1 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 7. 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-understood propose reject-proposal
    accept-proposal cancel inform : -> Act .
endfm
```

Figure 7. 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.

5.2 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 8, 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]).

```
(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 [Preceiningmsgl]: 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 8. Example of modeling of *Package* in Maude

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.

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.

5.3 Description of the agents' interaction modes in Maude

The three interaction modes defined in AUML (figure 3) are all supported by Maude. We use only one rewriting rule (see figure 9) 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 9. 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',* it doesn't exist a 'standard' evaluation strategy 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 10).

```
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 10. 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 11) 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 11. 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 12.

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

Figure 12. 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 13).

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

Figure 13. Modelling of interaction mode concerning 'exclusive or' in Maude using the messages.

In the example of figure 8, 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.

Note that all rewriting rules proposed in this section are terminating. The left part of each rule is different than the right one.

5.4 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. OUR TOOL SUPPORT APPROACH

We developed, along this work, a tool supporting the generation of Maude specification from an AUML description of an Agents' Interaction Protocol. This tool allows the user creating and manipulating different AUML diagrams.

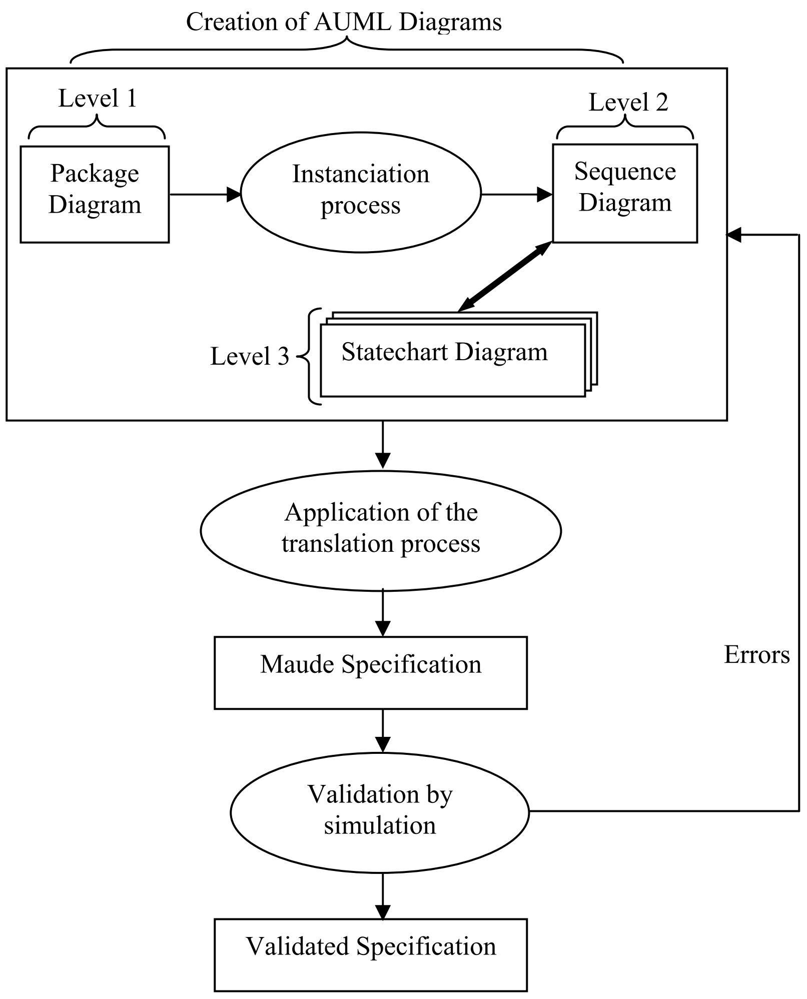


Figure 14. Architecture of the tool.

The developed tool is easy to use. It guides the user during the generation process of the Maude specification. Furthermore, by using this tool, the user can validate the generated specification.

Figure 14 shows the principal steps to be followed in order to achieve the task. For that, the user has to:

1. Create AUML diagrams (package, sequence, and state chart diagrams),
2. Apply the translation process in order to generate a Maude Specification describing the created AUML diagrams,
3. validate the generated specification by means of simulation,

The obtained results must to be analysed by the user (designer) in order to detect errors. If errors have been detected, the user must return back to AUML diagrams for correcting them.

Figure 15 presents the result of the creation of AUML Sequence diagram by instantiating package diagram. It illustrates the diagrams describing the Contract Net protocol.

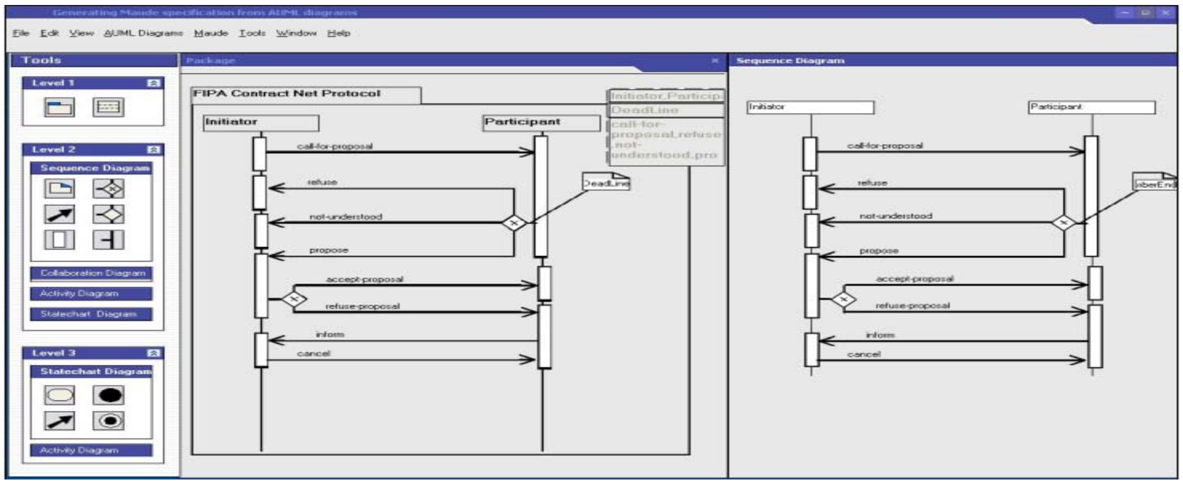


Figure 15. Creation of AUML Sequence diagrams.

To describe agents' internal behavior, users have to create a state chart diagram for each agent role. Figure 16 shows two state chart diagrams associated to the Initiator and Participant roles.

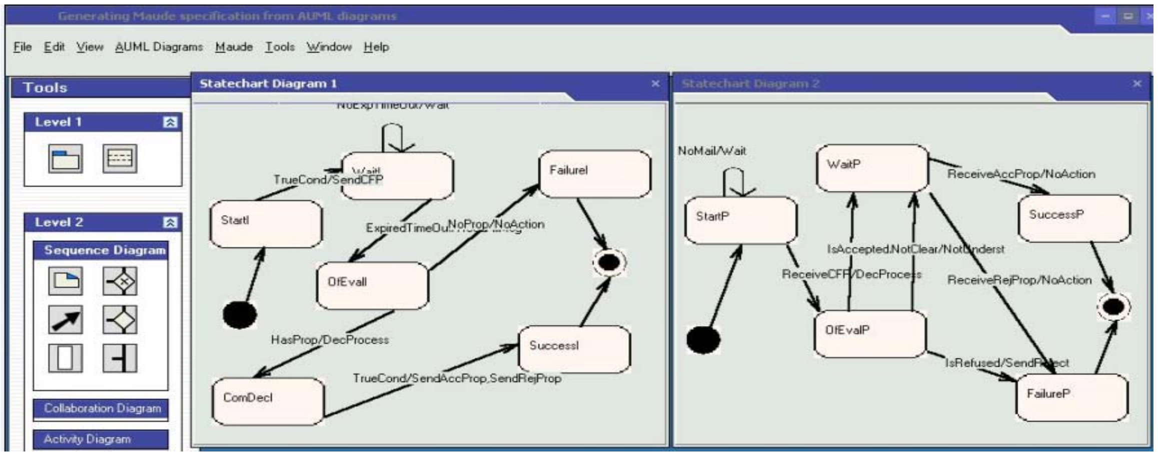
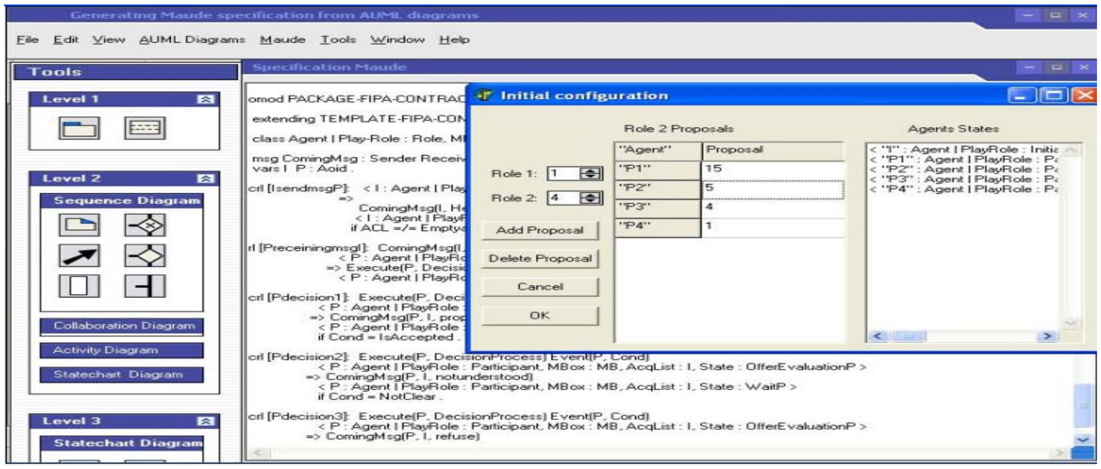


Figure 16. Creation of AUML state chart diagrams.

Once created, the AML diagrams will be translated into a formal Maude specification. Figure 17, illustrates the generated Maude specification, as well as, the initial configuration used to validate this specification. The initial configuration shows, all agents (one initiator (I) and four participants (P1, P2, P3, and P4) in their initial states. It also shows, the different cost proposed by the participants to accomplish the call for proposal.



```
omod PACKAGE-FIPA-CONTRAC
 extending TEMPLATE-FIPA-CON
 class Agent | Play-Role : Role, MI
 msg ComingMsg : Sender Receiv
 vars I  P : Aoid .
crl [IsendmsgP]:   < I : Agent | Play
              =>
                    ComingMsg(I, He
                   < I : Agent | PlayF
                 if ACL =/= Emptya
rl [Preceiningmsgl]:  ComingMsg(I,
                  < P : Agent | PlayRo
               => Execute(P, Decisi
                  < P : Agent | PlayRo
crl [Pdecision1]:  Execute(P, Deci
             < P : Agent | PlayRole :
          => ComingMsg(P, I, prop
             < P : Agent | PlayRole :
             if Cond = IsAccepted .
crl [Pdecision2]:  Execute(P, DecisionProcess) Event(P, Cond)
             < 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)
             < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
          => ComingMsg(P, I, refuse)
```

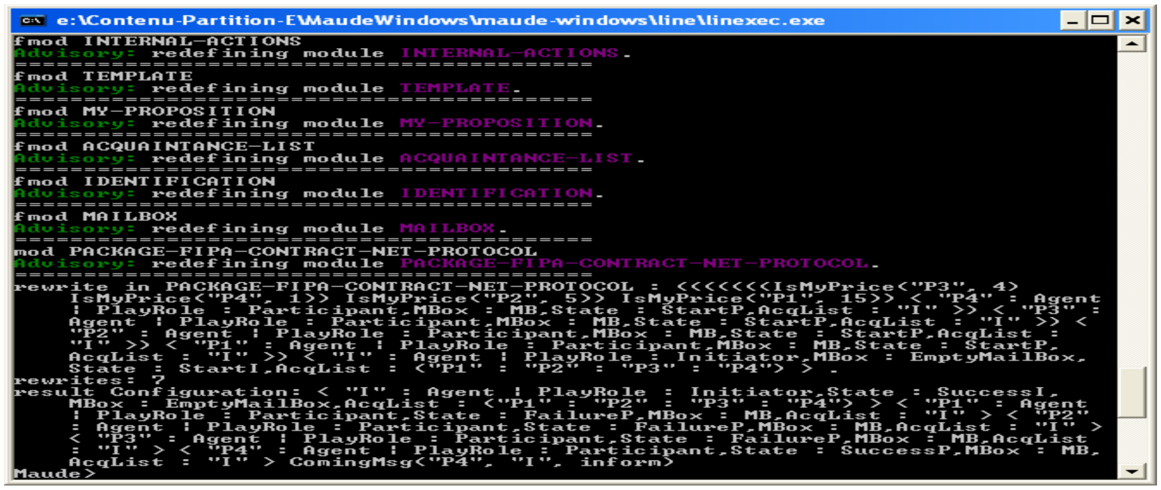
Role 1: 1
Role 2: 4

"Agent"	Proposal
"P1"	15
"P2"	5
"P3"	4
"P4"	1

```
< "I" : Agent | PlayRole : Initia
< "P1" : Agent | PlayRole : Pa
< "P2" : Agent | PlayRole : Pa
< "P3" : Agent | PlayRole : Pa
< "P4" : Agent | PlayRole : Pa
```

Figure 17. The generated Maude specification.

After it sends a call-for-proposal to all participants, the initiator agent begins to receive the proposal on behalf of participant agents. It arranges the received proposals in its mailbox in an increasing order according to the proposed costs.



```
e:\Contenu-Partition-E\MaudeWindows\maude-windows\line\linexec.exe
fmod INTERNAL-ACTIONS
Advisory: redefining module INTERNAL-ACTIONS.
==========================================
fmod TEMPLATE
Advisory: redefining module TEMPLATE.
==========================================
fmod MY-PROPOSITION
Advisory: redefining module MY-PROPOSITION.
==========================================
fmod ACQUAINTANCE-LIST
Advisory: redefining module ACQUAINTANCE-LIST.
==========================================
fmod IDENTIFICATION
Advisory: redefining module IDENTIFICATION.
==========================================
fmod MAILBOX
Advisory: redefining module MAILBOX.
==========================================
mod PACKAGE-FIPA-CONTRACT-NET-PROTOCOL
Advisory: redefining module PACKAGE-FIPA-CONTRACT-NET-PROTOCOL.
==========================================
rewrite in PACKAGE-FIPA-CONTRACT-NET-PROTOCOL : <<<<<<<IsMyPrice("P3", 4>
    IsMyPrice("P4", 1>> IsMyPrice("P2", 5>> IsMyPrice("P1", 15>> < "P4" : Agent
    | PlayRole : Participant,MBox : MB,State : StartP,AcqList : "I" >> < "P3" :
    Agent | PlayRole : Participant,MBox : MB,State : StartP,AcqList : "I" >> <
    "P2" : Agent | PlayRole : Participant,MBox : MB,State : StartP,AcqList :
    "I" >> < "P1" : Agent | PlayRole : Participant,MBox : MB,State : StartP,
    AcqList : "I" >> < "I" : Agent | PlayRole : Initiator,MBox : EmptyMailBox,
    State : StartI,AcqList : <"P1" : "P2" : "P3" : "P4"> > .
rewrites: 7
result Configuration: < "I" : Agent | PlayRole : Initiator,State : SuccessI,
    MBox : EmptyMailBox,AcqList : <"P1" : "P2" : "P3" : "P4"> > < "P1" : Agent
    | PlayRole : Participant,State : FailureP,MBox : MB,AcqList : "I" > < "P2"
    : Agent | PlayRole : Participant,State : FailureP,MBox : MB,AcqList : "I" >
    < "P3" : Agent | PlayRole : Participant,State : FailureP,MBox : MB,AcqList
    : "I" > < "P4" : Agent | PlayRole : Participant,State : SuccessP,MBox : MB,
    AcqList : "I" > ComingMsg("P4", "I", inform)
Maude>
```

Figure 18. Result of the unlimited rewriting of the initial configuration of the figure 17

Once the deadline considered is expired, the initiator launches its evaluation process by choosing the most appropriate proposal (here we adopt the strategy based on the minimum cost). So, the initiator sends to the chosen participant (here P4) an acceptance (accept-proposal), and to the remainder of participants (here P1, P2, and P3) a reject (reject-proposal). The chosen participant must inform the initiator of the execution of the proposition by sending it a message (inform). It expresses the fact that the participant consumed the message accept-proposal.

7. CONCLUSION AND FUTURE WORK

Modeling languages have received a lot of attention since they offer important modeling facilities by using diagrammatic notations. They enable developers to view large systems via diagrammatic notations, which can increase understanding [Tal03]. Agent UML is one among the modeling languages in multi-agent systems area. It offers several diagrams allowing describing interactions between agents. However, the lack of formal semantics in AUML, can lead to several incoherencies in the description of a MAS' behavior.

We presented, in this paper, a visual tool supporting our approach proposed in a previous work [Mok07]. We focused on the translation of AUML diagrams into Maude specification. Using the developed tool, designers can construct their own interaction models via AUML diagrams and translating them into Maude specification. Furthermore, they may validate the generated specification by means of simulation techniques using the Maude platform.

As future work, we plan to extend our tool for, on the one hand, taking into account agents' static aspects using class diagrams, and on the other hand, verifying the generated descriptions by using Maude's incorporated model checker.

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.

[Aye08] L.J. Ben Ayed, Fatma Siala, "From AUML Protocol Diagrams to Event B for the Specification and the Verification of Interaction Protocols in Multi-agent Systems," compsac, pp.581-584, 2008 32nd Annual IEEE International Computer Software and Applications Conference, 2008

[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.

[Bus99] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents --- components for intelligent agents in Java. AgentLink News, Issue 2, 1999.

[Cla99] M. Clavel and al. "Maude : Specification and Programming in Rewriting Logic". Internal report, SRI International, 1999.

[Cla05] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Mesenguer, and C. Talcott. Maude Manual (Version 2.1.1), April 2005.

[Edm08] A. Edmunds, M. Butler. "Linking Event-B and Concurrent Object-Oriented Programs". Electr. Notes Theor. Comput. Sci. 214: 159-182 (2008)

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

[Hug02] M.F. Huget. ''Model Checking Agent UML Protocols Diagrams''. Technical report ULCS-02-012. Department of Computer Science, University of Liverpool. Version 16/04/2002.

[Kah05] L. Kahloul, K. Barkaoui, Z. Sahnoun, "Using AUML to derive formal modeling agents interactions," aiccsa, pp.109-vii, ACS/IEEE 2005 International Conference on Computer Systems and Applications (AICCSA'05), 2005.

[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.

[Lem03] C. Lemaître, X. Prat, L0 Magnin, L. et A. Dury "Description, programmation et validation d'interactions par Coupled Augmented Transition Network(CATNs)''. Dans les Actes des Secondes Journées Francophones sur les Modèles Formels d'Interactions (MFI'03). Lille, France, 20-23 mai 2003.

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

[Mes92] 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.

[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.

[Mok07] 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

[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", conférence 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.

[OMG05] Object Modeling Group. "Unified Modeling Language Specification", version 2.0, July 2005.

[Pad07] L. Padgham, J. Thangarajah and M. Winikoff."AUML Protocols and Code Generation in the Prometheus Design Tool".AAMAS'07, May 14–18, 2007, Honolulu, Hawai'i, USA.

[Pau03] S. Paurobally, J. Cunningham, and N. R. Jennings "Developing Agent Interaction Protocols Using Graphical and Logical Methodologies", in Proc. AAMAS03 PROMAS Workshop on Programming Multi-Agent Systems , 2003.

[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.

[Tal03] A. Taleghan I, J. Ostrof. "Bon Development Tool". In Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange. Pages: 10-14. 2003.

[Tha05] J. Thangarajah, Lin Padgham, Michael Winikoff: Prometheus design tool. AAMAS 2005: 127-128.

[Tra01] E. Tranvouez. ''IAD et ordonnancement: une approche coopérative du réordonnancement par systèmes mulit-agents''. Thèse de doctorat. Université de Droit, d'Economie et des Sciences d'Aix-Marseille III. 2001.

About the authors

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 an HDR (Habilitation à Diriger des Recherches), 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.

Brahim Sahraoui (BSahraui@yahoo.fr) is a Master's Degree student at the University of Setif in Algeria. His main areas of interest include agent-oriented software engineering, and natural language processing.

Soufiane Bouzaher (SBouzaher@yahoo.fr) is a Software Development Engineer. His main areas of interest include agent-oriented software engineering and formal methods.

Mohamed Taher Kimour (Kimour@yahoo.fr) is an assistant professor of computer science at the Department of Computer Science of the University of Annaba in Algeria. He holds a Ph.D. in computer science (Software Engineering) from the University of Annaba in Algeria. His main areas of interest include object and agent-oriented software engineering, and embedded systems.