A Formal Framework Supporting the Specification of the Interactions between Agents

Farid Mokhati
Département d'Informatique
Université d'Oum El Bouaghi - Algérie
E-mail: Mokhati@yahoo.fr

Mourad Badri and Linda Badri
Département de Mathématiques et d'Informatique
Université du Québec à Trois-Rivières - Canada
E-mail: Mourad.Badri@uqtr.ca, Linda.Badri@uqtr.ca

Keywords: multi-agent systems, RCA, Maude, translation, behavior, interactions, formal specification, verification and validation.

Received: May 24, 2005

In this paper we present a formal framework supporting the translation of interactions between agents (the interactions are described with the help of the RCA formalism) in a Maude specification. Based on rewriting logic, the formal and object-oriented language Maude supports formal specification and programming for a wide range of applications. The main motivations of our work are essentially: (1) to formally specify the behavior of multi-agent systems and (2) to provide a solid basis for their verification and validation. The translation process is illustrated by means of a real case study.

Povzetek: Opisano je prevajanje interakcij med agenti.

1 Introduction

In Multi-Agent Systems (MAS), agents interact in order to exchange information, to cooperate and to coordinate their tasks [24]. The usual approach to the description of interactions between agents consists in using protocols [8, 26]. Several agents' interaction protocols (AIP) have been proposed in the literature [7]. They constitute an important part of MAS's infrastructures. However, most of the protocols published in the literature are semi-formal, vague or contain errors as mentioned in [23]. Knowing that AIP play a crucial role in MAS development [30], their formal specification as well as their verification constitute essential tasks [11]. In the field of agents' behavior specification, three major approaches emerge in the literature: state-charts based approaches [27, 22], Petri Nets based approaches [5, 1], and approaches representing an adaptation of object-oriented specification methods [19, 20].

Among the agents' interaction protocols proposed in the literature, we can mention the RCA formalism (Représentation des Comportements d'Agents) [27], which is based on strongly typed states-transitions graphs. The RCA formalism allows describing agents' behaviors graphically. This formalism has been used in the design of several Cooperative Information Systems (CIS) based on informational agents. We can mention, for example, the NetMan project based on the coordination of several agents [4], a project related to the reactive reorganization of production shops and treating the cooperation between agents having to solve a problem in a distributed and cooperative way [28], as well as a project on the hydraulic management of the Camargue ecosystem and based on a negotiation process between agents (Project SIMFONHYC) [18].

One of the strong points of the RCA formalism [18, 28] resides in the modular design of agents' behaviors. Indeed, the use of composite action states makes it possible the overlapping of behavioral plans and therefore a description by successive refinements of agents' behavior. This characteristic comes directly from the notion of composite state of RCA graphs. Nevertheless, some critiques on RCA graphs can be formulated, notably on their formalization and on the sequential aspect of the execution cycle of behavioral plans [28]. Furthermore, this formalism allows the visualization of the synchronization points between dual protocols thanks to the complementarity between communication states and external transition. It is then easy to recognize the coordination points between dual protocols [28]. However, RCA graphs as well as the existing formalisms in the literature describing agents' interaction protocols are not endowed again with a formal semantics [28]. They only offer a semi-formal specification [23] of interactions between agents. These weaknesses can generate several problems in MAS development and verification.

Using formal notations for the description of MAS' behavior offers several advantages. It essentially allows producing rigorous and precise descriptions supporting efficiently their verification and validation process. The Maude language, based on the rewriting logic, seems to us to be an interesting candidate. It offers, through its rich notation, an interesting way for concurrent systems formal specification and programming. Furthermore, it also supports the description of multi-agent interactions [21, 16]. In this paper, we present a formal framework supporting the translation of multi-agent interactions, specified using the RCA formalism, in a Maude specification. The main motivations of our approach are essentially: (1) to specify formally the behavior of multi-agent systems, in particular, the interactions between agents, and (2) to provide a solid basis for their verification and validation process. The Maude specifications, generated in the context of the developed framework, have been validated using the platform supporting the Maude language. The remainder of the paper is organized as follows: Section 2 gives a brief survey on the main related works. We present summarily the RCA formalism in section 3. In section 4, we give the basic concepts related to the rewriting logic as well as the Maude language. Section 5 presents the translation process. The proposed approach is illustrated using a concrete case study in section 6. Finally, section 7 gives some conclusions and future work directions.

2 Related Work

We present briefly in this section three formalisms (AUML, CATN and RCA) supporting the description of agents' interaction protocols. AUML [19, 9] is an extension of the UML language allowing describing interactions between agents. To represent multi-agent interaction protocols, AUML adopts in fact an approach in three layers. It uses, in the first level, packages and templates to represent the protocol in whole. Sequence diagrams, collaboration diagrams, activity diagrams, and states-transitions diagrams are used to represent interactions between agents. Activity diagrams and states-transitions diagrams are also used to capture agents' internal behavior (for more details see [19]). However, AUML only offers a semi-formal specification of the interactions between agents.

The CATN formalism (Coupled Augmented Transition NetWork) [10] is a states-transitions machine, to which a particular goal (or significance) is associated. A CATN can be decomposed in sub-CATNs. Each of these components is a CATN, having its own goal. The components of a CATN are joined together by ad-hoc transitions named "interactions transitions". Among these, we distinguish the non-terminal interactions transitions of those that are terminal. These last correspond to language acts (between agents) or to private actions of agents. This recursive aspect of the CATN allows a top-down design approach, from the most abstract behavior of a group of agents until their most concrete actions (individual terminal actions and communications through the interactions transitions). Each agent can execute in a concurrent way several CATNs depending on the tasks that it has to achieve [10, 25].

The RCA formalism [27, 28], supporting the description of role protocols, is used to describe agents' behavior. It is based on states-transitions diagrams introducing seven types of states and two types of transitions. The seven states are: the initial state, the final state, the elementary action state, the composite action state, the communication state and the waiting states (limited and unlimited). The two types of transitions are the internal transition and the external transition. Using this formalism, it is easy to recognize the coordination points between dual protocols. The RCA formalism is not limited to the description of the exchanges of messages between agents (as the case in the other formalisms). It also allows clarifying the actions that they undertake. In addition, the RCA graphs describe the working of the agents and help thus the design of their interactions. The links that exist between the macro level (i.e. the system's behavior) and the micro level (i.e. the agent's behavior) may be considered in an integrated way [28, 29].

These different approaches certainly offer some elements of answer to some problems related MAS development. However, they only allow a partial formalization of MAS. Furthermore, some authors [6, 5] opposed to the use of formalisms based on state-transition graphs two major arguments: 1) the impossibility to be able to verify the consistency of the protocols thus specified; and 2) the absence of taking into account the concurrent aspects of protocols [28]. In spite of the advantages that it offers relatively to the other formalisms, the RCA formalism only offers a graphic semi-formal description [18]. Furthermore, it is not endowed again with a formal semantics. These weaknesses combined to the complexity of MAS can generate several problems in their development and verification processes. The use of an appropriate formal notation for the description of MAS' behavior offers several advantages. It essentially allows the production of rigorous and precise descriptions supporting efficiently their verification and validation process. Our approach is similar, in terms of objectives, to the previously quoted approaches. It consists, essentially, to support the important stage of the specification of agents' behaviors. However, we preferred to adopt a more formal approach in the specification of agents' behaviors in terms of interactions allowing, among others, to support the verification of consistency (internal and global) in the behavior. Our approach allows translating the interaction protocols described using the RCA formalism in the Maude language. The Maude system consists in a high-level language of programming, specification and modeling based on rewriting logic [2, 15, 21]. It is also endowed with a high performance interpreter. It allows describing concurrent systems and supports the formal specification of distributed systems [14, 29, 12].

3 RCA Formalism

RCA (Représentation des Comportements d'Agents) [27, 28] is a formalism allowing describing an agent's behavior graphically. It is based on a strongly typed graph: seven types of states and two types of transitions (figure 1). The seven states are the initial state (to show the beginning of the graph), the final state (to mark the end of the graph), the elementary action state (that corresponds to the agent's simple action), the composite action state (it is in fact about the call to another protocol), the communication state (sending of message), and the limited and unlimited waiting states (waiting of treatments done by other agents). The two types of transitions are the internal transition (it corresponds to the end of an activity and provokes the passage to another state) and the external transition (it is in fact a reception of a message that provokes, like an internal transition, the change of the agent's activity). An external transition is triggered by a communication state at another agent.

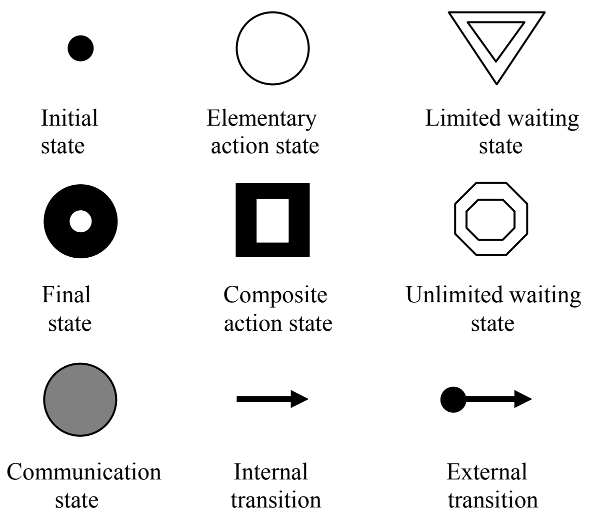


Figure 1 : Convention of representation of the RCA formalism.

The number of internal and external transitions depends on the type of the starting state and its transitions. It can be either null, limited or unlimited (figure 2).

Type of transition's departure state	Authorized internal transitions number	Authorized external transitions number
	[Min..Max]	[Min..Max]
Initial state	[0 .. 1]	[0 .. 1]
Elementary action state	[1 .. ∞]	[0 .. 0]
Composite action state	[1 .. ∞]	[0 .. 0]
Communication state	[1 .. 2]	[0 .. 0]
Limited waiting state	[1 .. 1]	[1 .. ∞]
Unlimited waiting state	[0 .. 0]	[1 .. ∞]
Final state	[0 .. 0]	[0 .. 0]

Figure 2 : Authorized transitions number according to the starting state.

Each states graph starts with a unique initial state and finishes by a unique final state. The internal events are the consequence of the agent's actions represented by action states (elementary or composite). They trigger the internal transitions. The external events result from communication activities of the agents, i.e. a reception of message constitutes an external event and provokes the crossing of an external transition. Of this fact, the type of allowed transition at a precise place of the graph depends exclusively of the origin state type of this transition:

- Initial state : only one transition (internal or external) may quit this state.
- Action state (simple or composite) : the internal transitions are in any number not null after action states.
- Communication state : one or two internal transitions may quit the communication state.
- Limited waiting state : the waiting may stop after the reception of a message (external transition), or if no message has been received beyond the waiting delay (internal transition). Furthermore, only one internal transition may quit a limited waiting.
- Unlimited waiting state : this waiting type remains while that it doesn't occur an external event (reception of message). It is therefore about a blocking state.

4 Rewriting Logic and Maude Language

4.1 Rewriting Logic

The rewriting logic, having a sound and complete semantics, was introduced by Meseguer [14]. It allows describing concurrent systems. This logic unifies all the formal models that express concurrence [13, 15]. In rewriting logic, the logic formulas are called rewriting rules. They have the following form: R:[t] → [t'] 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).

```
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 3 : Example of a portion of the Maude program.

For example, in an object-oriented system the concurrent state that is usually called configuration has the structure of a multi-set of objects and messages. Therefore, we can have configurations constructed by a binary operator applied to binary sets as illustrated in figure 3.

The portion of program illustrated in figure 3 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* its class and *atts* 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 | atts >*. This one is equivalent to the configuration < *O : C | atts* > *M1* because the __ operation is commutative.

4.2 Maude

Maude is a specification and programming language based on the rewriting logic [14, 3]. Three types of modules are defined in Maude. Functional modules allow defining data types and their functions through equations theory. Figure 4.a represents the functional module *Nat* specifying natural numbers. Such a module is imported in the module *FACT* (figure 4.b) to calculate the factorial of natural numbers. System modules define the dynamic behavior of a system. This type of modules extends functional modules by introducing rewriting rules. A maximal degree of concurrence is offered by this type of module. Finally, there are the object-oriented modules that can be reduced to system modules. In relation to system modules, object-oriented modules offer a more appropriate syntax to describe the basic entities of the object paradigm as, among others: objects, messages and configuration. Only one rewriting rule allows expressing 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.

Figure 5.a illustrates the use of a system module *BANK-ACCOUNT* to define an object counts banking *A* and the two operations capable to affect its content *credit* and *debit* while executing the rewriting rules defined in this module. Figure 5.b represents the same *BANK-ACCOUNT* module with a more appropriate object-oriented syntax.

```
fmod NAT is
sorts Zero NzNat Nat .
subsort Zero NzNat < Nat .
***constructors
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 4 : Functional Modules *Nat* and *FACT*.

We note, that after the execution of the unconditional rule [credit], the message *credit(A, M)* is consumed and the content of the account is increased. In the same way, the execution of the conditional rule [debit] requires that the condition (N>=M) be verified. The execution of such rule generates the consumption of the message *debit(A,M)* and the reduction of the content of the account.

```
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 MACHINE-INT .
class Account | bal : MachineInt .
msgs credit debit :   Oid MachineInt -> Msg .
var A : Oid .
vars M N : MachineInt .

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 5 : The same *BANK-ACCOUNT* module in system module and O.O module forms.

5 Translating RCA Descriptions in Maude

We developed a formal framework allowing the formal specification of role protocols described using RCA formalism. The framework is composed, as illustrated by figure 6, of several modules: an object-oriented module (*ROLE-PROTOCOLE*) and several functional modules (the remainder of modules).

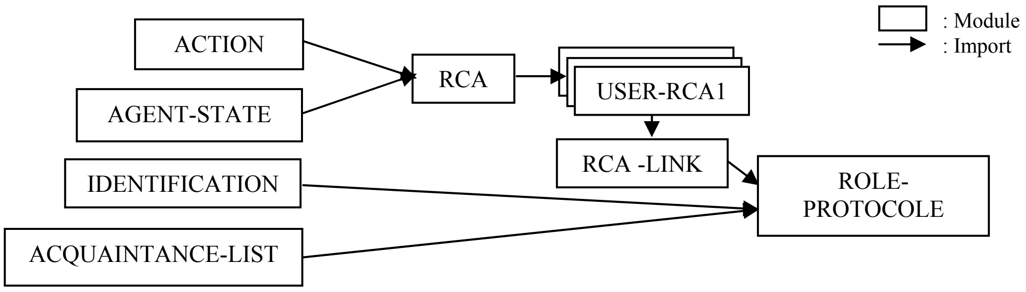


Figure 6 : *RCA-Maude* frameworks' architecture.

```
(fmod AGENT-STATE is
sorts AgentState KindAgentState NameAgentState .                               ***[1]

ops initial final communication elementary composite
                    limitedWaiting UnlimitedWaiting : -> KindAgentState .      ***[2]

op AgentState : NameAgentState KindAgentState -> AgentState .                  ***[3]
op IsInitial : AgentState -> Bool .                                            ***[4]
op IsFinal : AgentState -> Bool .                                              ***[5]
op IsOfCommunication : AgentState -> Bool .                                    ***[6]
op IsElementary : AgentState -> Bool .                                         ***[7]
op IsComposite : AgentState -> Bool .                                          ***[8]
op IslimitedWaiting : AgentState -> Bool .                                     ***[9]
op IsUnlimitedWaiting : AgentState -> Bool .                                   ***[10]

var k : KindAgentState . var ns : NameAgentState .

eq  IsInitial(AgentState(ns, k)) = if  k == initial then true                       ***[11]
            else false fi .
eq  IsFinal(AgentState(ns, k)) = if  k == final then true                           ***[12]
            else false fi .
eq  IsOfCommunication(AgentState(ns, k)) = if  k == communication then true        * **[13]
            else false fi .
eq  IsElementary(AgentState(ns, k)) = if  k == elementary then true                 ***[14]
            else false fi .
eq  IsComposite(AgentState(ns, k)) = if  k == composite then true                   ***[15]
            else false fi .
eq  IslimitedWaiting(AgentState(ns, k)) = if  k == limitedWaiting then true         ***[16]
            else false fi .
eq  IsUnlimitedWaiting(AgentState(ns, k)) = if  k == UnlimitedWaiting then true     ***[17]
            else false fi .
endfm)
```

Figure 7 : The functional module *AGENT-STATE*.

The functional module *AGENT-STATE* (figure 7) contains the different necessary type declarations for the definition of a state (line [1]) and, on the other hand, the definition of operations used for the construction and the manipulation of a state (lines [2, 3, 4, 5, 6, 7, 8, 9, 10]), as well as equations implementing these operations (lines [11, 12, 13, 14, 15, 16, 17]).

In the *ACTION* module (figure 8), in addition to the type Action, we define the two functions *IsSendingToOnlyOne* and *IsSendingToAll.* The first function determines if an action is destined to only one agent's acquaintance, on the other hand the second function indicates if it is necessary to send a message to all agent's acquaintances. To describe the identification mechanism of agents, we define the functional module *IDENTIFICATION* (figure 9). Furthermore, an agent must be endowed with a list of its acquaintances allowing it to exchange messages with the other agents. We define for it the functional module *ACQUAINTANCE-LIST* to manage the lists of the agents' acquaintances . Due to imitation of space and a considerable size of this last module, we don't present it in this paper.

```
(fmod ACTION is
protecting BOOL .
sort Action .
op IsSendingToAll : Action -> Bool .
op IsSendingToOnlyOne : Action -> Bool .
endfm)
```

Figure 8 : The functional module *ACTION*.

```
(fmod IDENTIFICATION is
 sort AgentIdentifier  .
 subsort  AgentIdentifier < Oid .
 endfm)
```

Figure 9 : The functional module *IDENTIFICATION*.

To define an RCA diagram, we propose the *RCA* module (figure 10). This module reuses the *AGENT-STATE* and *ACTION* modules. It includes the definition of two operations: *TargetState* that determines the target state according to a state source and an action, and the *FeedBack* operation used in the case where the treatment accomplished by the agent takes place while toppling between two states during a limited length. To each event coming from a state source, such a function determines the appropriate action that should be executed from the target state as a feedback.

```
(fmod RCA is
protecting ACTION .
protecting AGENT-STATE .
op TargetState : AgentState Action -> AgentState .
op FeedBack : Action -> Action .
endfm)
```

Figure 10 : The functional module *RCA*.

For the construction of an RCA diagram for an application, we propose to extend the *RCA* module in another *USER-RCA* module (figure 11). In this module, the user must: mention all states constituting the RCA diagram, define all possible actions, attach the actions in the states using the *TargetState* function, determine the actions constituting feedbacks using the *Feedback* function, and finally specify for every communication action whether it is sent to all (using the *IsSendingActionToAll* function) or to only one (using *IsSendingActionToOnlyOne*). An *USER-RCA* module (figure 11) is associated with every category of agents (playing the same role).

```
(fmod USER-RCA is
extending RCA .

***User part***
endfm)
```

Figure 11 : The functional Module *USER-RCA*.

To respect the interaction protocol used between agents, we propose to realize a sort of link between the RCA diagrams of the different agents. Basing on the synchronization points, main characteristic of this formalism, such a link consists in guaranteeing that at the moment of the reception of a message, an agent can't consume such a message except if it is in the corresponding state of the state of the sender agent. An agent that is in a communication state generates an external event that causes an external transition at the agent receiver. To receive such an event, this last must be in a waiting state (limited or unlimited). Indeed, the sending actions accomplished by a sender agent represent events for receiver agent. Thus, there is a correspondence between the sending actions of the sender and the events received by the receiver. For it, the user must develop the *RCA-LINK* module (figure 12) that contains the correspondence on the one hand, between the different states of agents and, on the other hand, between the events generated by the sender and the events received by the receiver.

```
(fmod RCA-LINK is
protecting USER-RCA  .
…
op CorrespondingState : AgentState -> AgentState .
op CorrespondingAction : Action -> Action .

***User part***************
…
endfm)
```

Figure 12 : The functional module *RCA-LINK*.

The object-oriented module *ROLE-PROTOCOL* (figure 13) represents the main module. It imports the *RCA-LINK*, *IDENTIFICATION,* and *ACQUAINTANCE-LIST* modules. For the formal description of agents, we propose the class *Agent* (line 2).

The definition of this class has as attributes *PlayRole*, *State,* and *AcqList,* to contain in this order, the agent's actual role, the current state of the agent, and the list of its acquaintances. In addition to different types of states defined in figure 7, we define in this module (figure 13) the type *EventType* (line 1) relative to the two types of events used in this formalism (Internal and External). The appearance of an event is expressed by message *Event* (line 3) having as parameters an agent, a role, the type of the event, the agent's state, and an action.

In the RCA formalism, an agent changes state while doing either an internal transition or an external one. Figure 13 illustrates the necessary rewriting rules we developed modeling the possible cases of transitions (internal and external), while respecting the constraints of this formalism described by the table given in figure 2.

```
(omod ROLE-PROTOCOLE is
protecting RCA-LINK .
protecting IDENTIFICATION .
protecting ACQUAINTANCE-LIST .
sorts Agent Role EventType .

ops Internal External : -> EventType .                                          ***[1]
class  Agent | PlayRole : Role, State : AgentState, AcqList : acquaintanceList  .   ***[2]
Msg Event : Oid Role EventType AgentState Action -> Msg .                       ***[3]

*********************************************************************************************
vars A A1 : Oid . var S : AgentState . vars R R1 : Role .
var Act : Action . var ACL : acquaintanceList .

*******************************Possible cases of internal transition****************************
*******************************************First case*************************************
crl[InternalTransitionCase1] :                                                  ***[4]
       Event(A, R, Internal, S, Act)
       < A : Agent | PlayRole : R, State : S, AcqList : ACL >
      =>
       < A : Agent | PlayRole : R, State : TargetState(S, Act), AcqList : ACL >
       if (IsInitial(S) or IsElementary(S) or IsComposite(S) or IslimitedWaiting(S)) .

*******************************************Second case*************************************
crl[InternalTransitionCase2] :                                                  ***[5]
      Event(A, R, Internal, S, Act)
      < A : Agent | PlayRole : R, State : S, AcqList : ACL >
    =>
      < A : Agent | PlayRole : R, State : TargetState(S, Act), AcqList : TailA(ACL) >
      Event(HeadA(ACL), R1, External, CorrespondingState(S), CorrespondingAction (Act))
      if IsOfCommunication(S) and IsSendingToOnlyOne(Act) .

*******************************************Third case*************************************
crl[InternalTransitionCase3] :                                                  ***[6]
       Event(A, R, Internal, S, Act)
       < A : Agent | PlayRole : R, State : S, AcqList : ACL >
     =>
       < A : Agent | PlayRole : R, State : S, AcqList : TailA(ACL) >
    Event(A, R, Internal, S, Act)
      Event(HeadA(ACL), R1, External, CorrespondingState(S), CorrespondingAction(Act))
      if IsOfCommunication(S) and IsSendingToAll(Act) and ACL =/= EmptyacquaintanceList .

********************Possible case of External transition****************************************
crl[ExternalTransition] :                                                       ***[7]
       Event(A, R, External, S, Act)
       < A : Agent | PlayRole : Initiator, State : S, AcqList : ACL >
     =>
       < A : Agent | PlayRole : Initiator, State : TargetState(S, Act), AcqList : ACL >
       if  IsInitial(S) or IslimitedWaiting(S)  or  IsUnlimitedWaiting(S) .

**********************************************************************************************
…
endom)
```

Figure 13 : The object-oriented module *ROLE-PROTOCOLE*.

An agent doesn't do an internal transition except if it is in one of the following states: initial, elementary, composite, limited waiting or communication (see figure 2). In the first four states, an internal transition is described by the rewriting rule (line 4) of figure 13. Such a rule expresses that at the moment of the appearance of an internal event, the agent consumes the message and changes its state using the *TargetState* function defined in the *RCA* module (figure 10). We treated separately the case of a communication state, knowing that from this state the agent generates an external event (sending of message) allowing its acquaintances that are in waiting to change their states. A message can be sent by an agent to only one agent belonging to its acquaintance list or to all its acquaintances.

The first case is described by the rule of the line 5. Such a rule expresses, on the one hand, the consumption of an internal event, on the other hand, the generation of an external event sent to only one agent (here we adopt the strategy choosing the agent that is at the head of the acquaintances list using the *HeadA* function), if the agent sender is in a communication state. The second case is described by the rule of the line 6. Such a rule presents the sending of a message by the agent *A* to all its acquaintances. It presents a conditional loop. Indeed, it allows browsing the acquaintance list (*ACL*) of the agent, while using the two operations *HeadA* (determines the head of the list) and *TailA* (determines the rest of the list). Such a loop stops when the list is browsed completely. An agent doesn't do an external transition except if it is in a waiting state (limited either unlimited) or sometimes in its initial state (see figure 2). This is expressed by the rewriting rule of the line 7. When it occurs an external event to an agent, this last changes its state while doing an external transition, but the agent must be in an initial or waiting state (limited either unlimited).

6 Case Study : Auction Application

This section illustrates the application of our approach on a concrete example. It is about a simple example of an auction.

We have two kinds of agents: *Auctioneer* and *Bidder*. Each auction involves one Auctioneer and several Bidders.

The Auctioneer has a catalog of products. Before beginning the auction, the Auctioneer sends the catalog to all participants. Then, it begins the auction for all products. The products are proposed sequentially to the participants. Figures 14.a and 14.b describe the representation of the Auctioneer and Bidder roles respectively using the RCA formalism.

6.1 Application of the Translation Process

The formal description of the behaviors of the agents whose roles are described using the RCA formalism implies all defined modules previously with the definition of the *USER-RCA* and *RCA-LINK* modules. Figures 15 and 16 illustrate the defined modules corresponding to the Auctioneer and Bidder roles respectively. The correspondence between these roles is presented in figure 17. Indeed, the two modules *USER-RCA1* (figure 15) and *USER-RCA2* (figure 16) describe the Auctioneer and Bidder roles respectively in the same way. We limit ourselves to detail the *USER-RCA1* module only.

In figure 15, we define the different states of the Auctioneer agent (lines 1 and 2). For example, the state *AgentState(CommitmentDecision, communication)* means that the state named *CommitmentDecision* is a communication state (see figure 14.a). The actions given in figure 14.a are described by line 3. To determine the target state (line 4) according to a source state and a given action, we used the operation *TargetState* defined in figure 10. If the Auctionner agent is in its *CommitmentDecision* state, and the action to execute is *AcceptProposalSent*, the target state of this transition must be the final state *EndI*. To select the conditional rule to execute when the agent is in a communication state (see figure 13, lines 5 and 6), it is necessary to know the type of the action. For example, the line 5 of figure 15 indicates that the *CFP-Sent* action must be sent by the Auctioneer to all Bidders.

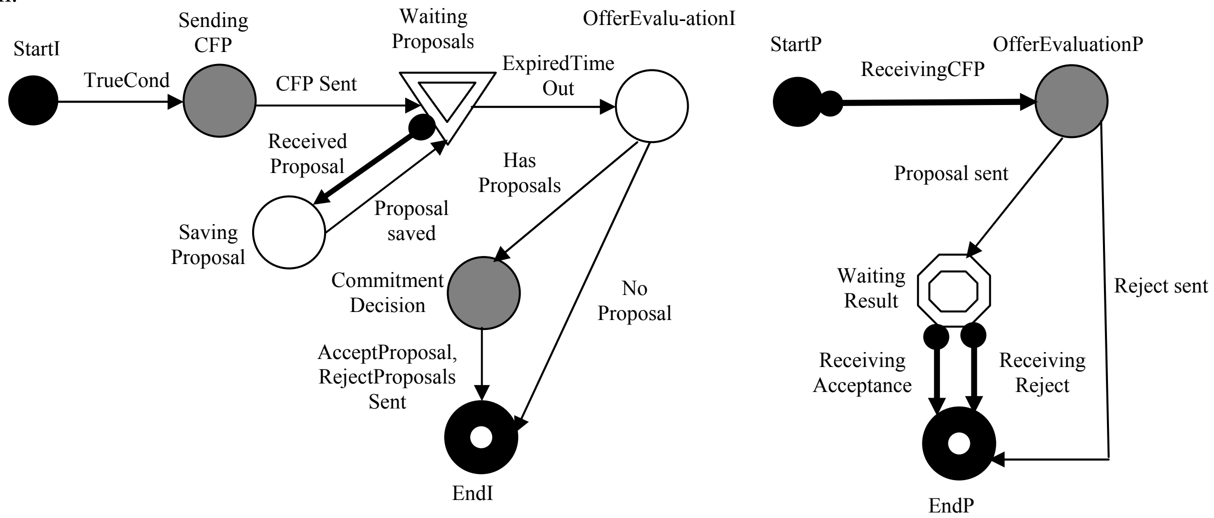


Figure 14 : Representation of the roles, *Auctioneer* and *Bidder* using RCA formalism.

```
fmod USER-RCA 1 is
extending RCA .

****************States of an Auctioneer************************************************
ops StartI SendingCFP WaitingProposals OfferEvaluationI SavingProposal
                                        CommitmentDecision EndI : ->  NameAgentState .       ***[1]

ops AgentState(StartI, initial) AgentState(SendingCFP, communication)
    AgentState(WaitingProposals, limitedWaiting) AgentState(OfferEvaluationI, elementary)
    AgentState(SavingProposal, elementary) AgentState(CommitmentDecision, communication)
                                              AgentState(EndI, final) : -> AgentState .      ***[2]

***************Actions to accomplish by an Auctioneer***********************************
ops TrueCondition CFP-Sent ExpiredTimeOut NoProposal HasProposal  ReceivedProposal
                       ProposalSaved AcceptProposalSent RejectProposalSent : -> Action .      ***[3]

***************Determination of the target state according to a state source and an action *********
eq TargetState(AgentState(StartI, initial), TrueCondition) = AgentState(SendingCFP, communication) .
…
eq TargetState(AgentState(CommitmentDecision, communication), AcceptProposalSent) =
                                                              AgentState(EndI, final) .      ***[4]
eq TargetState(AgentState(CommitmentDecision, communication), RejectProposalSent) =
                                                              AgentState(EndI, final) .

************* Determination of the type of an action ***************************************
eq IsSendingToAll(CFP-Sent) = true .                                                          ***[5]
eq IsSendingToOnlyOne(AcceptProposalSent) = true .

endfm
```

Figure 15 : The module *USE-RCA1* corresponding to the Auctioneer agent.

```
fmod USER-RCA2 is
extending RCA .

****************States of a Bidder****************************************************
ops StartP OfferEvaluationP WaitingResult EndP : -> NameAgentState .

ops AgentState(StartP, initial) AgentState(OfferEvaluationP, communication)
        AgentState(WaitingResult, UnlimitedWaiting) AgentState(EndP, final) : -> AgentState .

***************Action to accomplish by a Bidder****************************************
ops ReceivingCFP ProposalSent RejectSent ReceivingAcceptance  ReceivingReject : -> Action .

****************Determination of the target state according to a state source and an action *****
eq TargetState(AgentState(StartP, initial), ReceivingCFP) = AgentState(OfferEvaluationP, communication) .
…
eq TargetState(AgentState(WaitingResult, UnlimitedWaiting), ReceivingAcceptance ) = AgentState(EndP, final) .
eq TargetState(AgentState(WaitingResult, UnlimitedWaiting), ReceivingReject ) = AgentState(EndP, final) .

*************** Determination of the type of an action************************************
eq IsSendingToOnlyOne(ProposalSent) = true .
eq IsSendingToOnlyOne(RejectSent) = true .

endfm
```

Figure 16 : The module *USER-RCA2* corresponding to the Bidder agent.

The *RCA-LINK* module of figure 17, presents a correspondence on the one hand, between the different states of the agents Auctioneer and Bidder and, on the other hand, between the events they exchange. For example, if the Auctioneer agent is in its communication state *SendCFP*, the Bidder must be in its initial state *StartP* (line 1). In the same way, if the Bidder is in its communication state *OfferEvaluationP* (line 3), the Auctioneer must wait its decision. Indeed, an external event for an agent receiver corresponds to a message sent by a sender agent. For example, when the Auctioneer throws a call-for-proposal (*CFP-Sent*), the Bidder agent receives the call-for-proposal event (*ReceivingCFP*). This is expressed by the rule of the line 2. Also, when the Bidder accepts to propose, it sends its proposition (*ProposalSent*), and of the other side, the Auctionner receives its proposition (*ReceivedProposal*) (line 4).

```
fmod RCA-LINK is
protecting RCA1 .
protecting RCA2 .
sort EventType .

ops Internal External : -> EventType .
op CorrespondingState : AgentState -> AgentState .
op CorrespondingAction : Action -> Action .

**********************************Auctioneer Part*********************************

eq CorrespondingState(AgentState(SendingCFP, communication)) = AgentState(StartP, initial) .            ***[1]
eq CorrespondingState(AgentState(CommitmentDecision, communication)) =
                                          AgentState(WaitingResult, UnlimitedWaiting) .
…
eq CorrespondingAction(CFP-Sent) =  ReceivingCFP  .                                                   ***[2]
eq CorrespondingAction(AcceptProposalSent) = ReceivingAcceptance .

**********************************Bidder Part**************************************

eq CorrespondingState(AgentState(OfferEvaluationP, communication)) =
                                                AgentState(WaitingProposals, limitedWaiting) .    ***[3]
…
eq CorrespondingAction(ProposalSent) = ReceivedProposal .                                             ***[4]

endfm
```

Figure 17 : The module *RCA-LINK*.

6.2 Validation of the Generated Description

The rewriting logic offers a great flexibility in terms of simulation of a specification, in particular, concerning the choice of the initial configuration. This choice plays a primordial role in the validation of the description of a system. Using all the system's description, we can validate a part of the system without involving the rest. For a validation of the AIP given by figure 14, we consider two essential cases: the case where there are Bidders that accept to propose and others do not, and the case where all Bidders refuse to propose. For the first case, we propose the following initial configuration :

```
< "Auctioneer" : Agent | PlayRole : Initiator, State : AgentState(StartI, initial), AcqList : ("Bidder1" :
                                                                          ("Bidder2" : "Bidder3")) >
< "Bidder1" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
< "Bidder2" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
< "Bidder3" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
Event("Auctioneer", Initiator, Internal, AgentState(StartI, initial), TrueCondition)
Event("Auctioneer", Initiator, Internal, AgentState(SendingCFP, communication), CFP-Sent)
Event("Bidder1", Participant, Internal, AgentState(OfferEvaluationP, communication), ProposalSent)
Event("Bidder2", Participant, Internal, AgentState(OfferEvaluationP, communication), ProposalSent)
Event("Bidder3", Participant, Internal, AgentState(OfferEvaluationP, communication), RejectSent)  .
```

Figure 18 : Initial configuration.

We define an initial configuration including an agent initiator " Auctionneer ", and three agents participants ("Bidder1", "Bidder2", "Bidder 3"). In the beginning, every agent is in its initial state. From its *OfferEvaluationP* state a Bidder agent can send a proposition as it can refuse to propose. In the configuration of figure 18, Bidder1 and Bidder2 send their propositions whereas Bidder3 refuses to propose while sending a reject. The unlimited rewriting (without indicating the number of the rewriting steps) of this configuration gives the result illustrated by figure 19.

```
< "Auctioneer" : Agent | PlayRole : Initiator, State : AgentState(EndI, final), AcqList :
                                                        ("Bidder1" : ("Bidder2" : "Bidder3") >
< "Bidder1" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
< "Bidder2" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
< "Bidder3" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
```

Figure 19: Auctioneer and Bidders in their final states.

After it sends a call for proposal to all Bidders, the agent Auctioneer begins to receive the proposal from Bidders agents. Once the considered deadline is expired (internal event) the initiator throws its evaluation process while choosing the most appropriate proposition (here we adopt the strategy based on the first proposing).

So, the Auctioneer sends to the chosen Bidder (here "Bidder1") an acceptance, and to the other (here "Biddert2") a reject. Bidder3 is not concerned because it refused to propose and therefore passes to its final state (see figure 14.b). For the second case, we propose the initial configuration of the following figure:

```
< "Auctioneer" : Agent | PlayRole : Initiator, State : AgentState(StartI, initial), AcqList :
                                                        ("Bidder1" : ("Bidder2" : “Bidder3”)) >
< "Bidder1" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
< "Bidder2" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
< "Bidder3" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList : "Auctioneer" >
 Event("Auctioneer", Initiator, Internal, AgentState(StartI, initial), TrueCondition)
 Event("Auctioneer", Initiator, Internal, AgentState(SendingCFP, communication), CFP-Sent)
 Event("Bidder1", Participant, Internal, AgentState(OfferEvaluationP, communication), RejectSent)
 Event("Bidder2", Participant, Internal, AgentState(OfferEvaluationP, communication), RejectSent)
 Event("Bidder3", Participant, Internal, AgentState(OfferEvaluationP, communication), RejectSent) .
```

Figure 20 : Initial configuration.

The configuration of figure 20 looks like the one of figure 18 except that the Bidders refuse to propose. The unlimited rewriting (without indicating the number of the rewriting steps) of this configuration gives the result illustrated by figure 21.

```
< "Auctioneer" : Agent | PlayRole : Initiator, State : AgentState(EndI, final), AcqList :
                                                        ("Bidder1" : ("Bidder2" : "Bidder3") >
< "Bidder1" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
< "Bidder2" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
< "Bidder3" : Agent | PlayRole : Participant, State : AgentState(EndP, final), AcqList : "Auctioneer" >
```

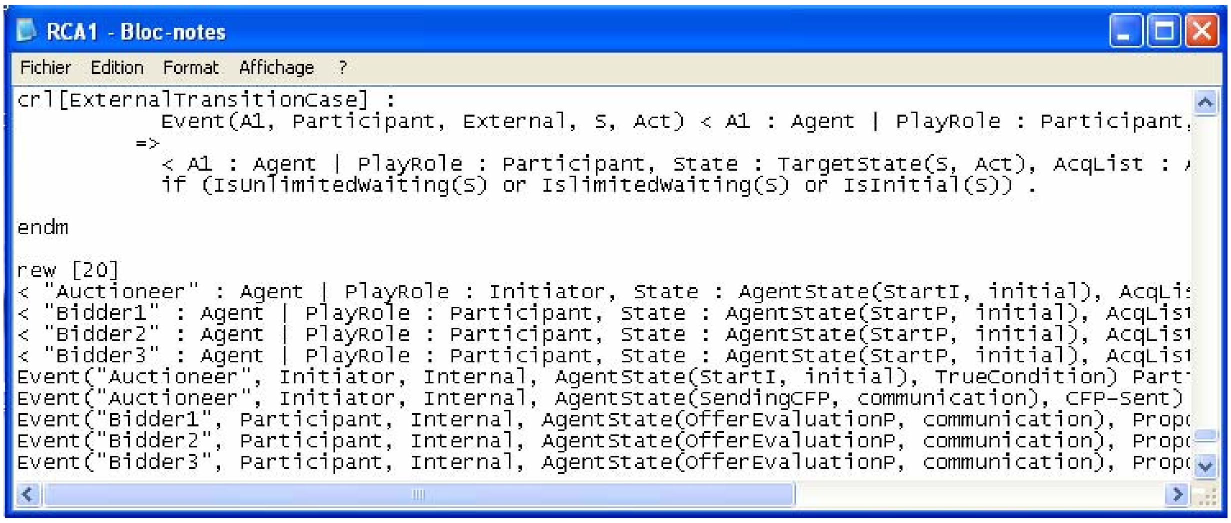
Figure 21: Auctioneer and Bidders in their final states.

Every participant who refuses to propose passes to the *EndP* state (see figure 14.b). In the same way, the initiator waits for the expiration of the deadline and as it doesn't receive any proposition during this interval of time, it passes on its turn in the *EndP* state (see figure 14.a). Indeed, the configuration of figure 21 seems to be the same that the one of figure 19. It is due to the fact that in the RCA formalism an agent can have only one final state. However, such configurations are different (for example, the *EndP* state of agent Bidder1 in figure 19 is a success state, but in figure 21 such a state presents a failure).

6.3 Implementation

Figure 22 illustrates a part of the code we developed. It visualizes the rewriting rule that describes the reception of an external event by the agent *A1* who plays the *Participant* role and exists in the state *S*. This rule also expresses the transition from the state *S* of the agent *A1* to another target state determined by the function

TargetState(S, Act). The triggering of such a transition only takes place if the agent *A1* is in one of waiting (limited or unlimited) or initial states. This is expressed in this conditional rule by the boolean functions *IsUnlimitedWaiting(S)*, *IslimitedWaiting(S)* and *IsInitial(S)* respectively.

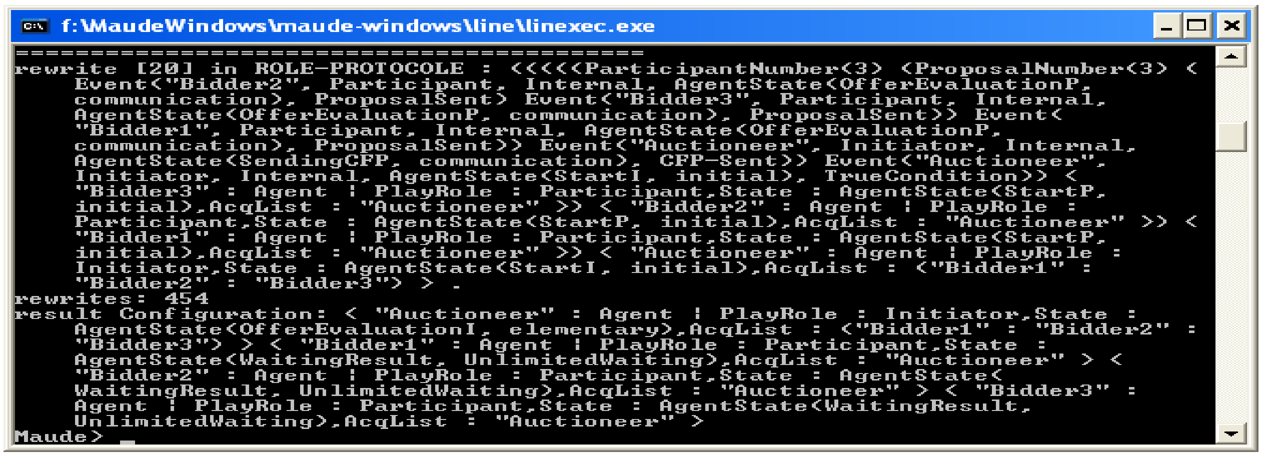


```
crl[ExternalTransitionCase] :
          Event(A1, Participant, External, S, Act) < A1 : Agent | PlayRole : Participant,
        =>
          < A1 : Agent | PlayRole : Participant, State : TargetState(S, Act), AcqList : 
          if (IsUnlimitedWaiting(S) or IslimitedWaiting(S) or IsInitial(S)) .

endm

rew [20]
< "Auctioneer" : Agent | PlayRole : Initiator, State : AgentState(StartI, initial), AcqLis
< "Bidder1" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList
< "Bidder2" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList
< "Bidder3" : Agent | PlayRole : Participant, State : AgentState(StartP, initial), AcqList
Event("Auctioneer", Initiator, Internal, AgentState(StartI, initial), TrueCondition) Part
Event("Auctioneer", Initiator, Internal, AgentState(SendingCFP, communication), CFP-Sent)
Event("Bidder1", Participant, Internal, AgentState(OfferEvaluationP, communication), Prop
Event("Bidder2", Participant, Internal, AgentState(OfferEvaluationP, communication), Prop
Event("Bidder3", Participant, Internal, AgentState(OfferEvaluationP, communication), Prop
```

Figure 22 : Part of the developed code.

Furthermore, figure 22 shows the limited rewriting (after 20 rewriting steps) of an initial configuration. In this configuration, we have the agent " Auctioneer " playing the *Initiator* role, and the three agents " Bidder1 ", " Bidder2 " and " Bidder3 " each playing the *Participant* role. All agents are in the departure in their initial states (*StartI* for agent Auctioneer and *StartP* for the Bidders). We suppose, in this initial configuration, that after the sending of the call for proposal by the Auctionner to all Bidders, these last send propositions in the case where they are in state of evaluation of proposal *OfferEvaluationP*. This state is a communication state (see figure 14).

The result of rewriting of such an initial configuration is illustrated by figure 23. The Auctioneer throws its decision process, and all Bidders wait for an answer from it. The agent Auctioneer is in its elementary state *OfferEvaluationI* and all Bidders are in their unlimited waiting states *WaitingResult*.



```
==========================================
rewrite [20] in ROLE-PROTOCOLE : <<<<<ParticipantNumber<3> <ProposalNumber<3> <
    Event<"Bidder2", Participant, Internal, AgentState<OfferEvaluationP,
    communication>, ProposalSent> Event<"Bidder3", Participant, Internal,
    AgentState<OfferEvaluationP, communication>, ProposalSent>> Event<
    "Bidder1", Participant, Internal, AgentState<OfferEvaluationP,
    communication>, ProposalSent>> Event<"Auctioneer", Initiator, Internal,
    AgentState<SendingCFP, communication>, CFP-Sent>> Event<"Auctioneer",
    Initiator, Internal, AgentState<StartI, initial>, TrueCondition>> <
    "Bidder3" : Agent | PlayRole : Participant,State : AgentState<StartP,
    initial>,AcqList : "Auctioneer" >> < "Bidder2" : Agent | PlayRole :
    Participant,State : AgentState<StartP, initial>,AcqList : "Auctioneer" >> <
    "Bidder1" : Agent | PlayRole : Participant,State : AgentState<StartP,
    initial>,AcqList : "Auctioneer" >> < "Auctioneer" : Agent | PlayRole :
    Initiator,State : AgentState<StartI, initial>,AcqList : <"Bidder1" :
    "Bidder2" : "Bidder3"> > .
rewrites: 454
result Configuration: < "Auctioneer" : Agent | PlayRole : Initiator,State :
    AgentState<OfferEvaluationI, elementary>,AcqList : <"Bidder1" : "Bidder2" :
    "Bidder3"> > < "Bidder1" : Agent | PlayRole : Participant,State :
    AgentState<WaitingResult, UnlimitedWaiting>,AcqList : "Auctioneer" > <
    "Bidder2" : Agent | PlayRole : Participant,State : AgentState<
    WaitingResult, UnlimitedWaiting>,AcqList : "Auctioneer" > < "Bidder3" :
    Agent | PlayRole : Participant,State : AgentState<WaitingResult,
    UnlimitedWaiting>,AcqList : "Auctioneer" >
Maude> _
```

Figure 23 : Result of limited rewriting (after 20 steps) of the initial configuration.

7 Conclusions and Future Work

The RCA formalism allows specifying the roles protocols and is used to describe agents' behavior. Compared to others formalisms, RCA allows recognizing the synchronization points between dual protocols. As for the other existing formalisms, RCA is not endowed yet with a formal semantics [28]. Furthermore, it only allows a partial formalization of MAS [17, 22].

In this article, we proposed a formal framework supporting the translation of interactions between agents, specified using the RCA formalism, in a Maude specification. The translation process is based on the RCA graphs. All the concepts used by the RCA formalism are supported by Maude. Based on rewriting logic, the formal and object-oriented language Maude supports formal specification and programming for a wide range of applications. The result of the translation procures a formal description of the interactions between agents preserving the consistency in their behavior. It offers a solid basis for their verification and validation process. The generated Maude specifications are flexible and remain open to extension.

Maude is supported by a tool. This allowed us, as a first experiment, in addition to the modeling, to perform a validation (based on a simulation) of our approach. Furthermore, we work on the extension of our approach in order to integrate the possibilities offered by the Maude language (model-checker) to verify some properties of the interactions between agents described using RCA graphs and translated in Maude.

References

[1] Bakam I., Kordon F., Le Page C., Bousquet F. « 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.

[2] Bruni R., and Meseguer J., « Generalized rewrite theories ». In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003), volume 2719 of Lecture Notes in Computer Science, pages 252-266. Springer, 2003.

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

[4] Cloutier L. «Une approche multi-agents par conventions et contrats pour la coordination de l'entreprise manufacturière réseau », Université de Droit d'Economie et des Sciences d'Aix-Marseille III, DIAM-IUSPIM, Marseille, 1999.

[5] Cost R., and al. «Modeling Agent Conversations with colored Petri Nets», dans Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents'99, Seattle, Washington, mai 1999.

[6] El Fallah-Seghrouchni A., et Mazouzi H., «Une démarche méthodologique pour l'ingénierie des protocoles d'interaction», in. Actes Ingénierie des systèmes multi-agents, JFIADSMA'99, 8-10 novembre 1999, Saint-Gilles, Ile de la Réunion.

[7] Guessoum Z. «Modèles et Architectures d'Agents et de Systèmes Multi-Agents Adaptatifs ». Dossier d'habilitation à diriger des recherches de l'Université Pierre et Marie Curie. Décembre 2003.

[8] Huget M.P., «Model Checking Agent UML Protocol Diagrams ». Technical report ULCS-02-012 from the department of computer science, University of Liverpool. Version 2002/04/16.

[9] Huget M.P., and Odell J. «Representing Agent Interaction Protocols with Agent UML » AAMAS'04, July 19-23, 2004, New York, New York, USA.

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

[11]Mazouzi H., El fallah Seghrouchni A.,, and Haddad S., «Open protocol design for complex interaction in multi-agent systems ». In Proceedings of the first international joint conference on Autonomous agent and multi-agent systems, pages 517-526. ACM Press, 2002.

[12]McCombs T., «Maude 2.0 Primer, Version 1.0». Internal report, SRI International, 2003.

[13]Meseguer J., «Rewriting as a unified model of concurrency» In Proceedings of the Concur'90 Conference, Amsterdam, Pg 384-400, Springer LNCS Vol. 458, 1990.

[14]Meseguer J., «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.

[15]Meseguer J., «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.

[16]Mokhati F., Boudiaf N., Badri L., & Badri M., «Generating Maude Specification from AUML Diagrams: Toward A Systematic Approach». In Proc of CSITeA-04 conference. Cairo, Egypt. December 27-29, 2004.

[17]Mokhati F., Boudiaf N., Badri M., & Badri L., «DIMA-Maude: Toward a Formal Framework for Specifying and Validating DIMA Agents». In Proc of the MOCA'04 conference. Arrhus, Denmark, October 11-13, 2004. pp. 169-187.

[18]Nathalie F., «Modélisation et simulation multi-agents d'écosystèmes antropisés : une application à la gestion hydraulique en grande Camargue», Université de Droit d'Economie et des Sciences d'Aix-Marseille III, IUSPIM-DIAM, Marseille, 2001.

[19]Odell J., Parunak H.V.D., Bauer B., «Representing agent Interaction protocol In UML» conférence AAAI Agents 2000, Barcelone, 3-7 juin 2000.

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

[21]Olveczky P.C., «Modeling and Analyzing Protocols in Maude» 8th Brazilian Symposium on Programming Languages (SBLP'04). May 26-28, 2004.

[22]Paurobally S, Cunningham J., «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.

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

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

[25]Pham V. T., Laurent M., Houari S., «Adaptation dynamique des systèmes multi-agents basée sur le concept de méta-CATN». In Actes de la Deuxième Conférence Internationale Associant Chercheurs Vietnamiens et Francophones en Informatique, Hanoï Vietnam, 2-5 Février 2004.

[26]Toivonen S. and 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

[27]Tranvouez E., Espinasse B., «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.

[28]Tranvouez E., «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

[29]Wooldridge M., and al., «A Methodology for Agent-Oriented Analysis and Design». Proc. 3rd Int. Conf. On Autonomous Agents (Agents99), Seatle, WA, 1999.

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