# Model Checking Agent UML Protocol Diagrams

Marc-Philippe Huget

Agent ART Group
University of Liverpool
Liverpool L69 7ZF
United Kingdom
M.P.Huget@csc.liv.ac.uk

**Abstract.** Agents in multiagent systems use protocols in order to exchange messages and to coordinate together. Since agents and objects are not exactly the same, designers do not use directly communication protocols used in distributed systems but a new type called interaction protocols encompassing agent features such as richer messages and the ability to cooperate and to coordinate. Obviously, designers consider formal description techniques used for communication protocols. New graphical modeling languages based on UML appeared several years ago. Agent UML is certainly the best known. Until now, no validation is given for Agent UML. The aim of this paper is to present how to model check Agent UML protocol diagrams.

*This paper is published as the technical report ULCS-02-012 from the department of computer science, University of Liverpool. Version 2002/04/16*

## 1 Introduction

Agents in multiagent systems use interaction so as to exchange information as well as to cooperate and to coordinate together. Usual approach is to describe interaction as protocols. Interaction protocols are derived from communication protocols in distributed systems. Actually, it is not possible to use directly communication protocols since several differences exist between agents and processes. Particularly, agents are more autonomous than objects and agents are able to join together in order to cooperate. Moreover, communication between agents is richer than the one in distributed systems. Actually, processes only send data through messages where agents use verbs and contents. Verbs are applied to contents. For instance, *inform(b == 10)* means that the receiver is informed that the sender believes that the value of $b$ is 10. We let readers consult [12] for a more detailed presentation of the differences between agents and objects. Interaction protocol designers have considered formal description techniques used for communication protocols: finite state machines [2], Petri nets [4] and languages such as LOTOS [13] and SDL [11]. Another type of formal description techniques is also considered: logic [6] or language Z [5]. A new approach appeared several years ago: modeling languages based on UML (see [14] for a description and an evaluation of some modeling languages). One particular modeling language draws our attention since it is chosen by the association FIPA[1] in order to describe interaction protocols. This language is called Agent UML [3] [15]. Bauer et al. explain that it is not possible to use UML for two reasons: (1) multiagent system designers are often software engineers who use UML for the description of their systems and (2) agents and objects do not encompass the same features, so it is not possible to represent all the agent features in UML. Interaction protocols in Agent UML are described through protocol diagrams [3].

Two stages are important when designing interaction protocols: formal description and validation. The latter tests if the protocol fulfills some properties and if some other properties are absent such as deadlocks or unreachable states. The validation stage checks if protocols encompass the features defined during requirement analysis. Two approaches exist for the validation: reachability analysis and model checking [7]. The former deals with syntactic properties whereas the latter deals with semantic properties.

These two methods are often used on formal description techniques such as finite state machines, Petri nets or LOTOS. Interaction protocol designers using Agent UML protocol diagrams do not have the same opportunities since until now, it does not exist such validation for these protocol diagrams.

The aim of this paper is to present a validation of Agent UML protocol diagrams by model checking using Promela/SPIN [9].

The reminder of this paper is divided as follows. Section 2 describes Agent UML protocol diagrams. Protocol diagrams are only represented graphically, it is required to provide a textual version of these protocol diagrams for the model checking. We use for that our proposal called AXF [10]. Section 3 presents the translation from the language AXF into Promela. The application of this translation is given in Section 4. The example is the English auction protocol. Section 5 concludes the paper and presents some future work.

## 2 Agent UML Protocol Diagram Description

Several notions are encompassed in protocol diagrams: agents or agents' roles, messages, constraints on messages and the protocol unfolding. All these notions are explained in this section.

### Agents and Agents' Roles

Agents on protocol diagrams can be represented by different manners: (1) they are represented by their identity called *instance* in Agent UML, (2) they are represented by their role in the protocol. For instance, in auction protocols, one has usually two roles: the *Seller* and the participants called *Auctioneer*. (3) finally, they can be represented by both their identity and their role.

[1] http://www.fipa.org

The agent class can be given for completing the definition of the roles. Using roles reduce the size of the diagram. Actually, one does not need several identities but just one role if they all share the same role.

Agents are described on protocol diagram by a box and a text defined as follows: instance "/" role ":" class. For instance, for auction protocols, one could have Smith/Auctioneer. Figure 1 gives two roles: *Initiator* and *Participant*. The vertical bar specifies messages which are sent from or are sent to this role.

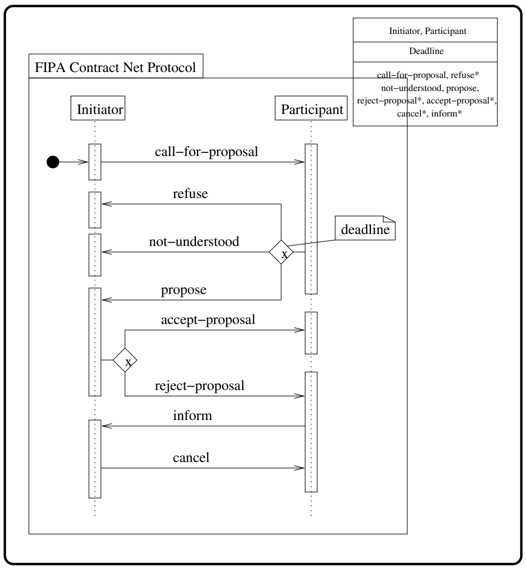


**Fig. 1.** Contract Net Protocol in Agent UML

### Lifelines

Agents are represented by a lifeline which corresponds to their participation in the protocol. When a lifeline is created for a role, this role becomes active for the protocol. This lifeline is present as long as the role has actions in the protocol. Lifelines are defined by a vertical bar drawn on the vertical bar of roles or linked to the vertical bar (see Figure 4 with the messages *not-understood*). Lifelines can be splitted or merged when several alternatives are available in the protocol or several paths are merged in one. For instance, it is the case on Figure 1 where several answers are available: *propose*, *refuse* or *not-understood*.

### Agent UML Connectors

Interaction protocol designers have three logical connectors for describing alternatives and choices (see Figure 2). The connector AND (see Figure 2a) means that messages have to be sent concurrently. The connectors OR (see Figure 2b) and XOR (see Figure 2c) mean that a choice between several messages has to be done. The connector OR means that zero or several messages can be chosen and the connector XOR means that one message has to be chosen. The term $CA$ refers to communicative acts [1].

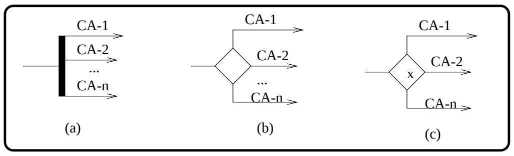


**Fig. 2.** Agent UML Connectors

### Various Notions

Several other notions are present in protocol diagrams:

**Conditions:** It is possible to put some conditions on sending messages. For instance, for the English auction protocol (see Figure 4), a condition is given for the messages *not-understood*. The variable $m$ must have a value greater than 0 if one wants to use this message. Conditions are given inside curly brackets.

**Cardinality:** Designers have to have the ability to send several copies of the same message (for advertising the opening of an auction for instance). In protocol diagrams, the multiple sending of messages has the meaning of multicast. Two numbers are defined: the first one is close to the vertical bar of the role which sends the message. The number is 1. The second number is close to the vertical bar of the role which receives the message. The value corresponds to the number of copies.

**Type of message:** Normally, messages are sent asynchronously (see Figure 3a). It is also possible to sent messages synchronously (see Figure 3b). The last possibility is when messages are not received instantaneously but after some delay (see Figure 3c).

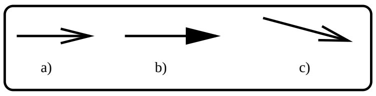


**Fig. 3.** Type of Messages in Agent UML

**Comments:** It is possible to insert some comments on diagrams, for instance *start cfp time* on the diagram of the Figure 4.

**Repetition:** One example of repetition is given on Figure 4 for the message requesting the payment. The curly brackets contain two numbers which represent how many times the message has to be sent. Here, the values range from 0 to 1 since the current price might be less than the reserved price, thus no payment is requested.

**Nested and interleaved protocols:** Since protocols are reusable, it is possible to link protocols together. Two solutions are available in Agent UML protocol diagram: nested protocols and interleaved protocols. Nested protocols correspond to protocols which are within another protocols. This case is used when one needs to repeat several times a protocol. Nested protocols continue while conditions hold.

Interleaved protocols correspond to protocols which are called during the execution of a second protocol. Bauer et al. give the example of an auction participant who requests some information about her/his bank account [3].

## 3 AXF-Promela Translation

Agent UML protocol diagrams are only represented graphically, it is thus important to have a textual version of these diagrams in order to apply tools on them. As far as we are concerned, the only possibility to describe textually Agent UML protocol diagrams is our proposal called AXF (Agent UML eXchange Format) [10]. Due to lack of space, we do not present in depth what AXF is. Readers can consult [10] for the syntax. AXF is a XML-based language.

This section describes how the different elements used in protocol diagrams are translated into both AXF and Promela. The algorithm for the translation is given in appendix.

The first element in an AXF file is the set of roles used in the diagram. Roles (defined between <role> and </role>) correspond to processes in Promela. For instance:

```
<roles>
  <role>
     <instance>UML-Airlines</instance>
     <name-role>Auctioneer</name-role>
     <class>Seller</class>
  </role>
  <role>
     <instance>AuctionParticipants</instance>
     <class>Consumer</class>
  </role>
</roles>
```

We got two processes: $p1$ and $p2$. These names are not exactly the name of the roles since they are too long for Promela. We just use the letter $p$

and an index. It means that $p1$ is UML-Airlines/Auctioneer:Seller and $p2$ AuctionParticipants:Consumer.

The main part of AXF files is the message description used in protocol diagrams. A message is composed of several fields: the sender, the receivers, the content, the multiplicity, the conditions, the repetition, the comments and the kind of message. We only consider here the following fields for the validation with Promela: the sender, the receivers, the content and the conditions. We omit the multiplicity field even if this field can be used in Promela. In this case, we define as much processes as we have copies of this message. The multiplicity allows designers to test the boundedness of channels. Comments are needless for the validation. Finally, we do not consider the type of messages: synchronous, asynchronous and delayed. Actually, some messages can be sent asynchronously and some others synchronously. This feature needs that we consider two kind of channels in Promela. Moreover, it is not possible to represent delayed messages in Promela. Conditions are present but the value of conditions is not checked during execution. In fact, one builds one path where conditions hold and one path where conditions do not hold.

Two other information are important: the state where agents are and state reached when agents have sent or received messages. These information are given by the tags <state>...</state> and <next-state>...</next-state>. If the next state is end-state, the protocol reaches a final state. This final state is distinguished by a state labeled with end.

For instance, for the first message of the English Auction Protocol [10], we have:

```
<state> 1
  <message>
     <from>
        <role>
           <instance>UML-Airlines</instance>
           <name-role>Auctioneer</name-role>
           <class>Seller</class>
        </role>
     </from>
     <to>
        <role>
           <instance>AuctionParticipants</instance>
           <class>Consumer</class>
        </role>
     </to>
     <CA>inform(start-auction, departure, arrival)</CA>
     <cardinality> 1 n </cardinality>
     <type> asynchronous </type>
     <next-state> 2 </next-state>
  </message>
</state>
```

The Promela structure depends on whether the current process sends the message or receives the message. If the process sends the message, we obtain:

```
State_1: printf("inform");
         p12p2!inform;
         goto State_2;
```

The message `printf("inform")` is only given for users and is not useful. Thus, users can have a trace of the execution. If the process receives the message, we obtain:

```
State_1: p12p2?inform;
         goto State_2;
```

The message construction is modified according to the connectors AND, OR or XOR.

If we have the connector AND, we replace it by an ordered set of messages. In fact, this connector means that this is a parallel sending of several messages but this feature is not available in Promela. One example of using of this connector is given in the English Auction Protocol [10]. The following code corresponds to the translation into Promela:

```
end_State_6: printf("inform");
             p12p2!inform;
             if
               ::1 -> printf("request");
                      p12p2!request;
               ::2 -> skip;
             fi;
```

This example allows us to present another feature of the translation: the conditions. They are given between `if...fi`. In fact, we simulate the conditions. In a first case, we consider that the conditions hold (the value 1 in the if) and in a second case, we consider that the conditions do not hold (the value 2 in the if). The conditions are not evaluated but we only consider the two possibilities when evaluating conditions: true or false.

The connector OR means that processes have to choose 0 or several messages. We use for that a loop defined between `do` and `od`. Let us suppose that we have one connector OR for the following messages: $a$, $b$ and $c$, we obtain in Promela:

```
do
  ::(a == 0) -> printf("a");
                p12p2!a;
                a = 1;
                goto ...
  ::(b == 0) -> printf("b");
                p12p2!b;
```

```
               b = 1;
               goto ...
  ::(c == 0) -> printf("c");
               p12p2!c;
               c = 1;
               goto ...
  ::1 -> break;
od;
```

The tests are given in order to avoid that the same message is sent several times. If processes send the message $a$, the value of $a$ is set to 1. So, at most n iterations must be done. The last condition corresponds to the case where processes choose 0 message.

The connector XOR means that one message is chosen among several messages. For instance, for the English Auction Protocol [10], we have the following example:

```
<state> 4
  <xor-message>
     <branch>
        <message>
           <from>
              <role>
                 <instance>UML-Airlines</instance>
                 <name-role>Auctioneer</name-role>
                 <class>Seller</class>
              </role>
           </from>
           <to>
              <role>
                 <instance>AuctionParticipants</instance>
                 <class>Consumer</class>
              </role>
           </to>
           <CA>reject-proposal(wrong-price)</CA>
           <type> asynchronous </type>
           <next-state> 5 </next-state>
        </message>
     </branch>
     <branch>
        <message>
           <from>
              <role>
                 <instance>UML-Airlines</instance>
                 <name-role>Auctioneer</name-role>
                 <class>Seller</class>
```

```
            </role>
          </from>
          <to>
            <role>
              <instance>AuctionParticipants</instance>
              <class>Consumer</class>
            </role>
          </to>
          <CA>accept-proposal(correct-price)</CA>
          <type> asynchronous </type>
          <next-state> 5 </next-state>
        </message>
      </branch>
      <cardinality> k </cardinality>
   </xor-message>
</state>
```

Each message in the connector XOR is described between `<branch>` and `</branch>`. Here, we have two messages *reject-proposal* and *accept-proposal*.
The translation into Promela is:

```
State_4: if
           ::1 -> printf("reject_proposal");
                  p12p2!reject_proposal;
                  goto State_5;
           ::2 -> printf("accept_proposal");
                  p12p2!accept_proposal;
                  goto State_5;
         fi;
```

Since Promela does not accept dashes within names, we replace them by underscores. We use one $if$ structure for representing the different options. The conditions correspond to the index. This index has no link to the messages.

## 4 English Auction Protocol Example

We take the example of the English auction protocol in order to exemplify the use of the language AXF. The meaning of the English Auction Protocol is the following: one agent (the Auctioneer) wants to sell an item, s/he informs all the agents (or just a part of them who can be interested in it) (the Auction Participants) that a new auction begins. The Auctioneer informs the Auction Participants how much is the item. Then, the auction participants can inform the Auctioneer that they accept the bid. If the Auctioneer has more than two Auction Participants who accept the bid, it selects the one who sent first the agreement and informs it that it wins this round. It informs other Auction Participants who sent an offer that their bid is rejected. The auction is opened till it remains only one auction participant who accepts the bid.

When the Auctioneer closes the auction, it compares the current price for this item to the reserved price. If the hammer price is greater than the reserved price, the item is won by the last agent who accepts the bid. Auction Participants are informed that the auction is closed.

The protocol in Agent UML is shown on Figure 4.

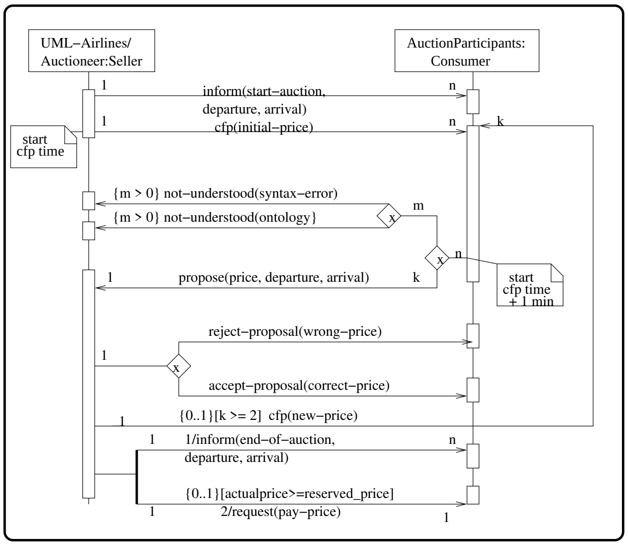


**Fig. 4.** The English Auction Protocol in Agent UML

The first message is sent to all agents by multicast. This situation is repeated once for the second message which details the item and the price. The deadline for the bidding is given as a comment. Then, auction participants can answer either by a *not understood*[2] message or by a *propose* message. If auction participants send a *propose* message, they inform the auctioneer they accept the bid. When the deadline is passed, the Auctioneer informs the Auction Participants if they have won this round (by a *accept proposal* message) or if they have lost this round (by a *reject proposal* message). The winner is the auction participant who sent first the acceptance of the bid. If the auctioneer has more than two agents which accept the price, it proposes once again the item at a higher price (message *cfp(new-price)*.

[2] FIPA defines in its specifications that all FIPA agents have to be able to send the message *not understood* if they do not understand the content of the message or the message.

If the auctioneer has only one agent, it closes the auction and informs all the auction participants (message *inform(end-of-auction)*). If the current price is greater than the reserved price, it informs the last auction participant who accepts the bid that it wins the item (message *request(pay-price)*).

The AXF description of this protocol is given in [10]. Due to lack of space, we do not give it here.

The translation into Promela is as follows:

```
mtype = {inform, cfp, not_understood, propose,
         reject_proposal, accept_proposal, request}

chan p12p2 = [2] of {mtype};
chan p22p1 = [2] of {mtype};

init {
  run p1(); run p2();
}

proctype p1() {

  mtype ca;

  State_1: printf("inform");
           p12p2!inform;
           goto State_2;

  State_2: printf("cfp");
           p12p2!cfp;
           goto State_3;

  State_3: p22p1?ca;
           if
             ::(ca == not_understood) -> goto end;
             ::(ca == not_understood) -> goto end;
             ::(ca == propose) -> goto State_4;
           fi;

  State_4: if
             ::1 -> printf("reject_proposal");
                    p12p2!reject_proposal;
                    goto State_5;
             ::2 -> printf("accept_proposal");
                    p12p2!accept_proposal;
                    goto State_5;
           fi;
```

```
      State_5: if
                 ::1 -> printf("cfp");
                        p12p2!cfp;
                        goto State_3;
                 ::2 -> goto State_6;
               fi;

      State_6: printf("inform");
               p12p2!inform;
               if
                 ::1 -> printf("request");
                        p12p2!request;
                        goto end;
                 ::2 -> goto end;
               fi;

      end: skip;
}

    proctype p2() {
      mtype ca;

      State_1: p12p2?inform;
               goto State_2;

      State_2: p12p2?cfp;
               goto State_3;

      State_3: if
                 ::1 -> printf("not_understood");
                        p22p1!not_understood;
                        goto end;
                 ::2 -> printf("not_understood");
                        p22p1!not_understood;
                        goto end;
                 ::3 -> printf("propose");
                        p22p1!propose;
                        goto State_4;
               fi;

      State_4: p12p2?ca;
               if
                 :: (ca == reject_proposal) -> goto State_5;
                 :: (ca == accept_proposal) -> goto State_5;
               fi;
```

```
85    State_5: p12p2?ca;
               if
                 :: (ca == cfp) -> goto State_3;
                 :: (ca == inform) -> goto end_State_6;
               fi;
90
      end_State_6: p12p2?request;
                   goto end;

      end: skip;

}
```

The first line gives all the possible messages that processes can use. Lines 4 and 5 define two channels, from the process $p1$ to the process $p2$ and the second one from the process $p2$ to the process $p1$. The process *init* (lines 7 to 9) is used for running the processes. Then, it is defined the two processes: $p1$ for the seller and $p2$ for the consumer. The process $p1$ matches exactly the protocol diagram in Figure 4 but it is not the case for the process $p2$. The states *State_ 5* and *State_ 6* are modified since the process $p2$ does not know in advance if the process $p1$ will send a message *cfp* or a message *inform*.

The protocol is now translated into Promela, it is then possible to do some verification with Spin [8]. For instance, we can test if states of this protocol are reachable. We got the following verification from Spin:

```
[mph ]> spin -a english.pmla
[mph ]> cc -o run pan.c
[mph ]> ./run
(Spin Version 3.4.12 -- 18 December 2001)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 32 byte, depth reached 26, errors: 0
      63 states, stored
      13 states, matched
      76 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

```
1.493   memory usage (Mbyte)

unreached in proctype :init:
        (0 of 3 states)
unreached in proctype p1
        (0 of 39 states)
unreached in proctype p2
        (0 of 36 states)
```

The lines:

```
unreached in proctype p1
        (0 of 39 states)
unreached in proctype p2
        (0 of 36 states)
```

means that no states are unreachable for both process $p1$ and process $p2$.

As a matter of interest, we first considered the protocol diagram given in [3] and we translated it into AXF. The translation is done without problems and then, we try to validate that all states are reachable. We were really surprised to notice that the only way to finish the protocol is by sending a *not-understood* message. We realized that the protocol diagram is wrong and *not-understood* is the only way to finish the protocol. Validation gives here a clear example of its interest.

## 5 Conclusion

Formal description and validation are the two main stages in protocol engineering. Actually, it is not ensured that the formal description gets rid of every problems. Validation is a usual stage for formal description techniques such as finite state machines, Petri nets or LOTOS. Recently, designers propose new graphical modeling languages. Agent UML is certainly the most known of these new modeling languages. It is based on UML. UML is not used since several differences exist between agents and objects such as autonomy and cooperation. Interaction protocols are represented through protocol diagrams in Agent UML. Until now, no validation stage is proposed for Agent UML protocol diagrams.

The aim of this paper is to describe such a validation of Agent UML protocol diagrams thanks to Promela and Spin. Since these protocol diagrams are graphical, designers need a textual version in order to use tools. We propose to use our proposal AXF which allows to represent textually protocol diagrams [10].

The algorithm for the translation from an AXF description into Promela is given in appendix. Such validation is important and allows designers to find errors within diagrams. We find one error in Bauer et al's diagram of the English Auction Protocol [3]. The only way to finish the protocol is to use the message *not-understood*.

For the moment, the validation is concentrated on reachability analysis. Our future work try to provide a richer range of validation to users, especially boundedness validation and semantic properties dealing with the purpose of protocols.

## References


1. J. L. Austin. *How to Do Things with Words.* Clarendon Press, 1962.
2. M. Barbuceanu and M. S. Fox. COOL : A language for describing coordination in multiagent system. In *First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, USA, June 1995. AAAI Press.
3. B. Bauer, J. P. Muller, and J. Odell. An extension of UML by protocols for multiagent interaction. In *International Conference on MultiAgent Systems (ICMAS'00)*, pages 207–214, Boston, Massachussetts, july, 10-12 2000.
4. R. S. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Modeling agent conversation with colored Petri nets. In J. Bradshaw, editor, *Autonomous Agents'99, Special Workshop on Conversation Policies*, May 1999.
5. M. d'Inverno and M. Luck. Formalising the contract net as a goal-directed system. In W. V. de Velde and J. Perram, editors, *Agents Breaking Away, MAAMAW 96*, number 1038 in Lecture Notes in Artificial Intelligence, pages 72–85. Springer-Verlag, 1996.
6. M. Fisher and M. Wooldridge. Specifying and executing protocols for cooperative action. In *International Working Conference on Cooperating Knowledge-Based Systems (CKBS-94)*, Keele, 1994.
7. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.
8. G. J. Holzmann. An analysis of bitstate hashing. In K. Academics, editor, *Proceedings of Protocol Specification, Testing and Validation (PSTV 95)*, pages 287–305, Warsaw, Poland, june 1995.
9. G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
10. M.-P. Huget. A language for exchanging agent uml protocol diagrams. Technical Report ULCS-02-009, Department of Computer Science, University of Liverpool, 2002.
11. C. Iglesias, M. Garrijo, J. Gonzales, and J. Velasco. Design of multi-agent system using mas-commonkads. In Springer-Verlag, editor, *Proceedings of ATAL 98, Workshop on Agent Theories, Architectures, and Languages*, volume LNAI 1555, pages 163–176, Paris, France, July 1998.
12. N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In J. Bradshaw, editor, *Handbook in Agent Technology.* MIT Press, 2000.
13. J.-L. Koning. Algorithms for translating interaction protocols into a formal description. In K. Ito, editor, *IEEE International Conference on Systems, Man, and Cybernetics Conference (SMC-99)*, Tokyo, Japan, October 1999.
14. J.-L. Koning, M.-P. Huget, J. Wei, and X. Wang. Extended modeling languages for interaction protocol design. In *Proceedings of Agent-Oriented Software Engineering (AOSE 01)*, Montreal, Canada, May 2001.
15. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, Texas, july, 30 2000. ICue Publishing.

# Appendix: AXF to Promela Translation Algorithm

```
//write the message type
Mess is the list of message types
WRITE("mtype = " + Mess)

//write the channels
FOR each role i DO
  FOR each role j DO
    IF (i != j) THEN
      WRITE("chan " + i + "2" + j + "= [2] of {mtype};")
      WRITE("chan " + j + "2" + i + "= [2] of {mtype};")
    END IF
  END FOR
END FOR

//write the init process
WRITE("init {")
FOR each role i DO
  WRITE("run i();");
END FOR
WRITE("}")

//write each process
FOR each role i DO
  WRITE("process " + i + "() {")
  WRITE("mtype " + ca + ";")

  //take the AXF version of the diagram for this role
  WHILE(the protocol diagram is not completely read) DO {
    take the first state
    retrieve the label of this state
    WRITE("State_" + label + ":");
    IF (connector == AND) THEN
      FOR each message m DO
        retrieve the communicative act ca of m
        WRITE("printf(" + ca + ");")
        retrieve the sender role sr
        retrieve the receiver role rr
        retrieve the processus name of these roles p_sr and p_rr
        IF (sr is the current role) THEN
          WRITE(p_sr + "2" + p_rr + "!" + ca;)
        ELSE
          WRITE(p_rr + "2" + p_sr + "?" + ca);
        END IF
      END FOR
```

```
  retrieve the next state ns of m
  IF (ns equals "end-state") THEN
    WRITE("goto end;");
  ELSE
    WRITE("goto State_" + ns)
  END IF
END IF

IF (connector == OR) THEN
  WRITE("do")
  FOR each message m DO
    retrieve the communicative act ca of m
    WRITE("::(" + ca + "==0) ->");
    WRITE("printf(\"" + ca + "\");")
    retrieve the sender role sr
    retrieve the receiver role rr
    retrieve the processus name of these roles p_sr and p_rr
    IF (sr is the current role) THEN
      WRITE(p_sr + "2" + p_rr + "!" + ca;)
    ELSE
      WRITE(p_rr + "2" + p_sr + "?" + ca);
    END IF
    WRITE(ca + " = 1;")
    retrieve the next state ns of m
    IF (ns equals "end-state") THEN
      WRITE("goto end;");
    ELSE
      WRITE("goto State_" + ns)
    END IF
  END FOR
  WRITE("::1 -> break;")
  WRITE("od;")
END IF

IF (connector == XOR) THEN
  WRITE("if")
  FOR each message m DO
    retrieve the communicative act ca of m
    WRITE("::" + index + "-> printf(\" " + ca + "\");")
    retrieve the sender role sr
    retrieve the receiver role rr
    retrieve the processus name of these roles p_sr and p_rr
    IF (sr is the current role) THEN
      WRITE(p_sr + "2" + p_rr + "!" + ca;)
    ELSE
```

```
          WRITE(p_rr + "2" + p_sr + "?" + ca);
        END IF
        index <- index + 1
        retrieve the next state ns of m
        IF (ns equals "end-state") THEN
          WRITE("goto end;");
        ELSE
          WRITE("goto State_" + ns)
        END IF
      END FOR
    END IF
  END WHILE
  WRITE("end: skip;");
  WRITE("}");
END FOR
```