# Generating Code for Agent UML Sequence Diagrams

Marc-Philippe Huget

Agent ART Group
University of Liverpool
Liverpool L69 7ZF
United Kingdom
M.P.Huget@csc.liv.ac.uk
Phone: +44 151 794 6791

**Abstract.** For several years, a new category of description techniques exists: Agent UML [10] which is based on UML. Agent UML is an extension of UML to tackle differences between agents and objects. Since this description technique is rather new, it does not supply tools or algorithms for protocol synthesis. Protocol synthesis corresponds to generate code for a formal description of a protocol. The derived program behaves like the formal description. This work presents first elements to help designers generating code for Agent UML sequence diagrams. The protocol synthesis is applied to the example of English Auction protocol.

*This paper is published as the technical report ULCS-02-020 from the department of computer science, University of Liverpool. Version 2002/06/28*

## 1 Introduction

For several years, a new description technique has emerged in the domain of multiagent systems: Agent UML [10] [11] [1]. It is used to design agents, organizations and interaction protocols among others. Unfortunately, this graphical modeling language is less endowed than other formal description techniques such as finite state machines [13], Petri nets [7] or languages like LOTOS, ESTELLE or SDL [14]. There are few work in validation and in code generation. We only focus, in this paper, on sequence diagram which is the diagram for representing interaction protocols. Huget's work [6] and Koning and Romero's work [8] are the only work on checking properties on sequence diagrams. No paper in the literature deals with code generation for sequence diagrams. Code generation when applied to protocols is called *protocol synthesis* [4], object-oriented community refers to forward engineering [2].

Protocol synthesis is the next stage after validation [4]. The aim of the protocol synthesis stage is to generate a program which behaves like the formal description of the protocol. Generating code is composed of two steps. In a first step which is more or less automated, an algorithm converts the formal description into a program skeleton. This program skeleton represents the structure of the protocol, i.e. the transitions between the different states. This step deals with program skeleton since it is not possible for an algorithm to insert the semantics of the transitions. Generally, this information is not supplied on formal descriptions. Moreover, this task is too difficult to be performed by a machine: protocols have to respect complex standards for instance. Inserting semantics is performed in a second step. This step is realized by hand. Readers refer to [3] for a detailed view of algorithms proposed by communication protocol engineering.

As stated above, Agent UML sequence diagrams is less endowed in comparison with other formal description techniques: there is no algorithm for generating code until now. The aim of this paper is to fill the gap by providing first elements for protocol synthesis. We do not supply an algorithm for protocol synthesis in this paper. It will be the subject of another paper. This paper presents the well-known English Auction protocol and an implementation of this one. The implementation is derived from the algorithm. Thanks to this implementation, designers have several basic elements for generating their own protocols. We use the Java programming language since it is widely used by agent community.

The remainder of this paper is as follows. Section 2 describes the example of English Auction protocol. This is the example for which we generate code. Section 3 describes our implementation of this protocol. Section 4 presents a discussion in order to help designers generating their own protocols based on this work. Section 5 concludes the paper and gives future directions.

## 2 The English Auction Protocol

In English Auction protocol, the auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the proposed price. As soon as one buyer indicates that it will accept the price, the auctioneer issues a new call for bids with an incremented price. The auction continues until no buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a buyer exceeds the auctioneer's reservation price, the good is sold to that buyer for the agreed price. If the last accepted price is less than the reservation price, the good is not sold.

The corresponding protocol in Agent UML is shown on Figure 1. The first message *inform* is sent by the auctioneer and informs participants that the auction is open. The *cfp* message advertises the good to be sold. It also gives the price that participants have to pay to get the good. Participants can answer either by *not-understood* or by *propose*. They answer with *not-understood* if they do not understand the message or the content of the message. They answer with *propose* if they want the auctioneer to know that they accept to pay the announced price.

After a fixed delay, the auctioneer can answer with *reject-proposal* for rejected bids and with *accept-proposal* for accepted bids. The winner of a round is the participant which sent first a *propose* message.

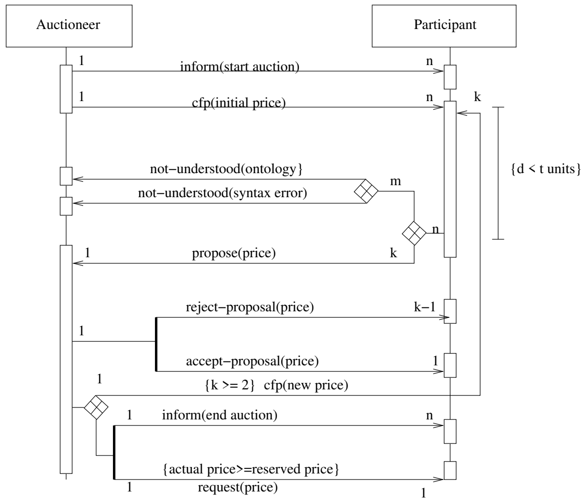


**Fig. 1.** English Auction Protocol in Agent UML

Two choices are then possible: (1) if more than two participants have accepted to pay the price, the auctioneer announces a new round with an incremented price (*cfp* message), (2) the auctioneer closes the auction if less than two participants accept to pay the price (*inform* message). Then, a winner is announced if the hammer price is greater than the privately known reserved price (*request* price).

## 3 Generating Code for English Auction Protocol

Last section described a sequence diagram for English Auction protocol. This section presents an implementation in Java. This implementation is derived from an algorithm not presented here. As a consequence, the corresponding program is limited to transitions and some elements adorned on sequence diagrams.

Generating code for a protocol must not be limited to write the transitions into a program. One has to take into account the agents using the protocol. Figure 1 presents two agent lifelines: *Auctioneer* and *Participant*. A lifeline is any combination of instances, roles and classes so we must be cautious how to define the Java classes. If we have just a class, we define a Java class but if we have roles and classes, we define a base class and several children, as many as we have roles. Figure 2 sums up this point.

Except derived Java classes for agents, we need several other information such as a message server class for exchanging messages. Figure 3 shows the different elements in the derived program. We use the stereotype `«agent»` to distinguish agents from objects.

Agents *Auctioneer* and *Participant* do not interact directly but through the message server. The class *Message* is used to store messages.

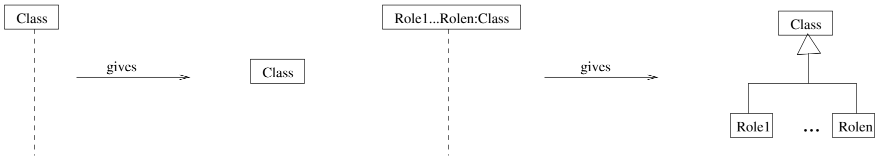


**Fig. 2.** Agent Lifelines Derivation

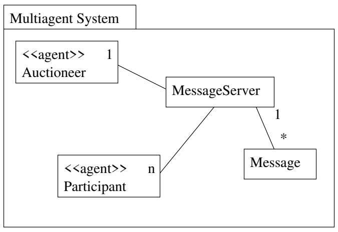


**Fig. 3.** Derived Multiagent Structure

We let apart the classes *MessageServer* and *Message* since they do not have an impact on code generation. A *MessageServer* class is an RMI object which is accessible to agents. Agents can send or receive messages. Sending and receiving messages are done through two operations in the class *MessageServer*.

The algorithm for code generation is a semi-automated algorithm since it needs some information such as the number of instances of each role and class. For instance for the example shown on Figure 1, we have two classes: *Auctioneer* and *Participant* but it does not correspond to the number of agents in the multiagent system. The information is supplied at the beginning of the algorithm. Multiplicity in the multiagent system is rendered on Figure 3: a number is given in the right top corner of agent classes. For instance, designers know that there is exactly one copy of *Auctioneer* in the multiagent system but several copies of *Participant*. The number of copy is not already fixed since it is written $n$. In our implementation, we have one agent for the class *Auctioneer*: *kenny* and three agents for the class *Participant*: *kyle, stan* and *cartman*. A second example where users are requested is for multiplicity for messages. Figure 1 contains $n$, $m$ or $k$ but they have no meaning for the algorithm.

The classes *Auctioneer* and *Participant* are symmetric, i.e. when one sends a message, the second one receives it and vice-versa. However, we present the two classes since the management of messages is not the same whether the agent is sender or receiver.

Agent class diagram of the class *Auctioneer* is shown on Figure 4. The *Auctioneer* class derives from the class *Thread* to show that this class is an active one.

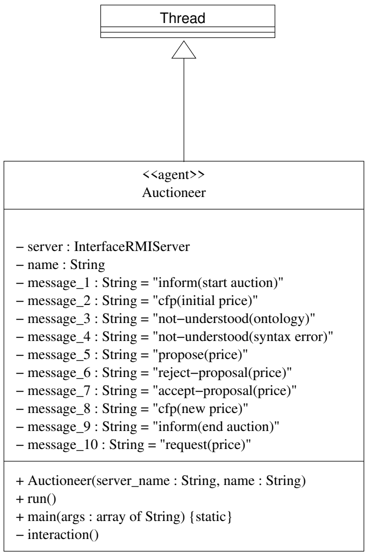


**Fig. 4.** Auctioneer Agent Class Diagram

The attribute *server* corresponds to the link to the message server. The attribute *name* is the agent name. Figure 4 does not present that this attribute is initialized to the name of the agent: *kenny, stan, kyle* or *cartman*. The dash which prepends the attributes means that these attributes are private to the class. The plus used below means that this element is public.

In order to simplify the code, the algorithm stores the messages as strings explaining why ten strings appear in the class diagram.

The operation *Auctioneer* is the constructor of the class. It links the agent to the message server. The operation *run* launches the interaction through the operation *interaction*. The operation *main* is the entry point in the agent. *main* executes the agent.

The main operation is the *interaction* one. It represents the execution of the sequence diagram. The code of the *interaction* operation is given below.

We now present the operation *interaction* which is the main part in the class. It contains the protocol. Usual approach in communication protocol engineering for the protocol synthesis stage is to represent a protocol as an *if...else* structure [4] [9]. For instance, if we have one interaction state and several outgoing arcs for this state. One represents that with several $if$s. With this method, one has two levels of $if$: one level for messages for each interaction states and one nested level corresponding to interaction states.

The interaction states are not implicitly written on sequence diagrams. Interaction states begin at 0 and they are incremented each time one has a new transition on threads of interaction. This increment is done according to sender's point of view. It means that the algorithm does not index states resulting from receiving messages: just one state for the messages *not-understood(ontology)*, *not-understood(syntax error)* and *propose(price)*.

We only present below the code for the operation *interaction*. We remind readers that this code corresponds to the one coming from the algorithm. As a consequence, some information is missing since they are dealing with protocol semantics.

```
private void interaction() throws RemoteException {

  int state = 0;

  while (state != 4) {

    if (state == 0) {
      Vector tmp = new Vector();
      tmp.addElement("stan");
      tmp.addElement("kyle");
      tmp.addElement("cartman");
      server.sendMessage(name, tmp, message_1);
      state = 1;

    } //if

    if (state == 1) {
      Vector tmp = new Vector();
      tmp.addElement("stan");
      tmp.addElement("kyle");
      tmp.addElement("cartman");
      server.sendMessage(name, tmp, message_2);
      state = 2;

    } //if

    if (state == 2) {

      try {

        sleep(d < t units);

      } //try
      catch(InterruptedException e) {}

      Message msg = (Message)server.getMessage(name);
      while((msg.getMessage()).length() != 0)
        msg = (Message)server.getMessage(name);

      state = 3;

    } //if
```

```
if (state == 3) {

  server.sendMessage(name,
                     loser,
                     message_7);

  server.sendMessage(name,
                     winner,
                     message_6);

  state = 4;

} //if

if (state == 4) {

  Random rand = new Random();
  int i = rand.nextInt(2);

  if (i == 0) {
    Vector tmp = new Vector();
    tmp.addElement("stan");
    tmp.addElement("kyle");
    tmp.addElement("cartman");
    if (k >= 2) {
      server.sendMessage(name, tmp, message_8);
      state = 2;

    } //if

  } //if
  else {
    Vector tmp = new Vector();
    tmp.addElement("stan");
    tmp.addElement("kyle");
    tmp.addElement("cartman");
    server.sendMessage(name, tmp, message_9);
    if (actual price >= reserved price)
      server.sendMessage(name, winner, message_10);

  } //else

} //if

} //while

} //interaction
```

The attribute *state* (line 3) contains the current state in the interaction. Four kinds of messages are on the sequence diagram of the Figure 1:

1. messages without conditions and with multiplicity *inform(start auction)*
2. messages with conditions and multiplicity *cfp(new price)*
3. parallelism with AND connector *reject-proposal(price), accept-proposal(price)*
4. decisions with XOR connector *not-understood(ontology), not-understood(syntax error)*

When messages are sent without conditions, it corresponds to a simple message such as the one in line 12 or in line 22. Lines 8 to 11 and lines 18 to 21 give the list of agents which receive these messages. The algorithm obtains this information when it begins to generate code. The variable $n$ is now linked to the value 3 and to the agents *stan, kyle* and *cartman*.

Code for messages with conditions and with multiplicity are more or less the same. The algorithm adds one *if...else* structure for the conditions. Conditions are written within the $if$ without considering if the presentation conforms Java notation. Two examples are on lines 68 and 81. These conditions respect Java notation but these variables are not defined and not initialized.

Parallelism does not exist in Java for communication, so the algorithm serializes the messages (lines 46 and 80). Decisions with XOR connector need information. This information is not supplied by the sequence diagram. Decisions are for the moment performed through a random function (line 61).

The last point is the deadline management between the *cfp* message and the answers from the participants. This deadline is rendered as a marker bar on sequence diagrams and the duration $d < t$ *units*. We replace this information by a Java *sleep* (lines 29 to 34). Thus, the agent which owns this *sleep* does nothing during the time associated to the *sleep*.

This code is generated without adding semantics to the transition so, some problems arise for the selection of messages or for the conditions. For instance, the auctioneer receives several proposals but there is no way to link these proposals to the two messages *accept-proposal* and *reject-proposal*. Moreover, the generated code only corresponds to read all the messages without doing something on these messages.

The code for the *Participant* class is quite similar. We give below interesting elements for code generation. The following code is when agents receive messages. We also take into account when the message is not the expected one:

```
if (state == 0) {
  Message message = (Message)server.getMessage(name);
  if ((message.getMessage()).length() != 0) {
    if ((message.getMessage()).equals(message_1))
      state = 1;
    else
      System.out.println(name + " unexpected message");
  } //if
} //if
```

The second example that we took from the *Participant* class is when sequence diagrams have a connector: either XOR, OR or AND. Since it is not possible for agents to know in advance what messages will be sent, agents retrieve one message then compare this message to all the possible messages. Once again, only the structure is given not the semantics associated to the sequence diagram.

```
if (state == 4) {

  Message msg = (Message)server.getMessage(name);
  if ((msg.getMessage()).length() != 0) {
    if ((msg.getMessage()).equals(message_8))
      state = 2;
    if ((msg.getMessage()).equals(message_9)) {
      msg = (Message)server.getMessage(name);
      state = 5;
    } //if

  } //if
} //if
```

## 4 Discussion

In this paper, we present the code generation for English Auction protocol. The elements described here can be applied to other examples. Most of the graphical elements on Agent UML sequence diagrams are shown on Figure 1 [11]: agent lifelines, threads of interaction, messages, multiplicity, conditions, AND connector, XOR connector and deadlines. Some few elements are not present in this example: OR connector, nested and interleaved protocols and types of message delivery.

OR connector is derived as an *if...else* structure where each $if$ corresponds to one possible message. A *default* case is also supplied to deal with the case where no message has to be sent.

Nested and interleaved protocols are directly inserted into the program, i.e. the corresponding code for these protocols are generated then it is inserted where needed. If nested protocols have a condition in order to do iterations, a loop with these conditions nests the program skeleton for this nested protocol.

Agent UML allows designers to use several kinds of message delivery: synchronous, asynchronous and delayed. The example presented here only uses asynchronous messages. Synchronous messages use a different version of the operation *sendMessage*. The operation *sendMessageSync* sends message synchronously. It means that the operation in the MessageServer does not return till the receiver does not acknowledge receipt. Delayed messages are described as asynchronous messages since the delay does not have to appear in the code.

Reusing this implementation for new protocols is quite straightforward. The classes *MessageServer* and *Message* do not have to be modified. Agents defined in this example can be reused: only names and the *interaction* operation have to be modified. Informally speaking, the algorithm corresponds to read sequence diagrams from top to bottom. First, one reads the different roles and classes used. The structure of the multiagent system is then derived. Then, for each agent lifeline, one considers if it is a sending message or a receiving message, if there are some conditions, if there is multiplicity. The two last pieces of information modify how the interaction state is generated.

## 5 Conclusion

Protocol synthesis is one stage of interaction protocol engineering [5] or communication protocol engineering [4]. It deals with the generation of a program which behaves like the formal description. This stage is performed in two steps: a first step uses an (semi-) algorithm which generates a program skeleton. The second step is realized by designers and implements semantics. Until now, there is no algorithm for generating code for sequence diagram.

The aim of this paper is to give some elements for generating code for sequence diagrams. We present one implementation of English Auction protocol. This implementation is obtained by an algorithm not presented here. The implementation allows designers to seize how to generate code.

Except deriving a program from a formal description, this protocol synthesis presents some interest since it is then possible to find how accurate is the model. This representation of English Auction protocol as shown on Figure 1 lacks some elements. For instance, it is not written how to link the *propose* message to the messages *reject-proposal* and *accept-proposal*. It might be interesting to add on the sequence diagram that all the proposals are stored in a bag, then the *accept-proposal* is sent to the first received proposal. This operation could be done with OCL [12]. A second problem arises for the condition $\{k \geq 2\}$. It is not possible to link this variable to the number of *propose* messages.

Future directions would be improving the quantity of generated code. Communicative acts in FIPA's ACL are supplied with semantics. It would be interesting to generate the code corresponding to the communicative acts. For instance, if an agent receive an *inform* message, the algorithm could generate code which updates the agent knowledge base. This feature has a counterpart since it needs more information from users thus reducing the automation.

**Acknowledgements**. This research was supported by the UK government under EPSRC project GR/R27518 (Verifiable Languages and Protocols for Multiagent Systems).

## References

1. 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.

2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.
3. P.-Y. M. Chu. *Towards Automating Protocol Synthesis and Analysis*. PhD thesis, Ohio State University, 1989.
4. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
5. M.-P. Huget. *Une ingénierie des protocoles d'interaction pour les systèmes multi-agents*. PhD thesis, Université Paris 9 Dauphine, June 2001.
6. M.-P. Huget. Model checking Agent UML protocol diagrams. In *ECAI Workshop on Model Checking Artificial Intelligence (MoChArt)*, Lyon, France, July 2002.
7. K. Jensen. *High-Level Petri Nets, Theory and Application*. Springer-Verlag, 1991.
8. J.-L. Koning and I. Romero-Hernandez. Generating machine processable representations of textual representations of AUML. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *AAMAS Workshop on Agent-Oriented Software Engineering (AOSE)*, Bologna, Italy, July 2002.
9. M. Narasimhan. An object oriented framework for composing communication protocols. Master's thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, january 1997.
10. 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.
11. J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of First International Workshop on Agent-Oriented Software Engineering*, Limerick, Ireland, june, 10 2000. Springer-Verlag.
12. OMG. UML 1.4. Technical report, OMG, 2001.
13. A. Salomaa. *Theory of Automata*. Pergamon Press, 1969.
14. K. J. Turner. *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons, Ltd, 1993.