Chapter 12

THE AUML APPROACH

Marc-Philippe Huget, James Odell and Bernhard Bauer

Abstract Since the earliest work in multiagent system development, the need has existed to have a methodology and a modeling notation. A recent approach chosen by several authors is based on UML. The main advantage of UML is to provide a recognized notation in software engineering methodology and strong tools for the development. The approach presented here is called Agent UML (AUML) that synthesizes a growing concern for agent-based modeling representations with the increasing acceptance of UML for object-oriented software development. This chapter covers the first phase (from 1999 to 2002) in the development of Agent UML with the sequence and agent class diagrams and describes future directions that follows Agent UML via the standardization at the FIPA.

1. Introduction

Since the earliest work in MAS development, the need has existed to have a methodology and a modeling notation, see the seminal work on Agent-Oriented Programming (Shoham, 1991). Since this first work from Shoham, many methodologies and notations appear for MAS development. Some chose temporal logic like (Fisher and Wooldridge, 1997) for Concurrent METATEM. A recent approach chosen by several authors is based on UML. The main advantage of UML is to provide a recognized notation in software engineering methodology and strong tools for the development. As (Odell et al., 2000) indicated, starting over to develop a new modeling language for agents was neither useful nor productive. Instead, MAS could, in part, benefit from an incremental extension of existing, known and trusted methods. Agent UML (AUML) synthesizes a growing concern for agent-based modeling representations with the increasing acceptance of UML for object-oriented software development.

Agent UML – as an extension of UML – allows developers coming software engineering to move smoothly from software development to agent development. Agent UML avoids abrupt steps by using and extending UML with agent-specific features through stereotypes and profiles. Agent UML was first

introduced in 1999 with the proposal on interaction protocols given in (Bauer, 1999). After a first period from 1999 till 2002 where Agent UML community proposes two specifications (sequence diagrams and agent class diagrams), a growing interest is visible and efforts become to be coordinated in order to define Agent UML based on the UML 2.0 (OMG, 2003b) specification and to standardize it at the FIPA association. This chapter covers the first period with the description of the two specifications and gives possible future directions that the Agent UML community may follow.

Following sections present the purpose of Agent UML in AOSE (see section 2), the current version of Agent UML with the sequence diagram and the agent class diagram (see section 3). The remaining of the chapter describes future directions of work for Agent UML community particularly with the creation of new diagrams, of tools, the definition of semantics for Agent UML and the development and documentation of applications that use Agent UML for the analysis, design, and implementation.

2. Agent UML Purpose

MAS are often characterized as extensions of object-oriented systems. This overly simplified view has often troubled system designers as they try to capture the unique features of MAS systems using OO tools. In response, an agent-based unified modeling language (AUML) is being developed.

Like UML on which it is partially based, the purpose of Agent UML is to offer to developers a notation that it is used to analyze, design, and implement MAS. The key idea of Agent UML is to reuse as much as possible diagrams coming from UML when they fit MAS designers' needs and to extend UML – through its extension abilities (stereotypes, tagged values, constraints) – when agents and objects are different and agents cannot be represented by UML diagrams. Such an example of extension is in Agent UML sequence diagrams with the three connectors AND, OR and XOR. The use of Agent UML in the documentation is clearly visible in the FIPA Interaction specification (see `http://www.fipa.org/repository/ips.php3`) since all the interaction protocols are documented with a diagram expressed in Agent UML.

It should be noted, however, that instead of reliance on the OMG's UML, we intend to reuse of UML wherever it makes sense. In other words, AUML should not be restricted to just UML – only want to capitalize on UML where appropriate. The general philosophy, then, is: when it makes sense to reuse portions of UML, then do so; when it does not make sense to use UML, use something else or create something new.

3. Current Work in Agent UML

The initial work on Agent UML focuses on the agent interaction. (Odell et al., 2001) define sequence diagrams (also called in some papers protocol diagrams) to this purpose. Agent UML sequence diagrams became *de jure* the notation to represent FIPA Interaction protocols as sketched in FIPA specifications. This first work on sequence diagrams was quickly followed by a proposal for the representation of agents and agent architectures through the agent class diagrams proposed in (Bauer, 2001) and refined in (Huget, 2002a).

3.1 Sequence Diagrams

Agent UML Sequence Diagrams were initially one of the most used diagrams because they were adopted by FIPA to express agent interaction protocols as sketched in FIPA specifications. Sequence diagrams are defined as follows in UML: *"A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships."* Since Agent UML considers agents and not objects, one must read agents instead of objects in the previous definition. Sequence diagrams in MAS are diagrams which express the exchange of messages through protocols.

Sequence diagrams have two dimensions: *(i)* the vertical dimension represents time; and *(ii)* the horizontal dimension represents different instances or roles. Messages in sequence diagrams are ordered according to a time axis. This time axis is usually not rendered on diagrams but it goes according to the vertical dimension from top to bottom. So, a message defined higher than a second one is sent before. Sequence diagrams do not use sequence numbers as collaboration diagrams to represent the message ordering. Message ordering is performed by the time axis.

Several basic components are used within sequence diagrams:

1 Agents and agent roles;

2 Agent lifelines and threads of interaction;

3 Connectors;

4 Messages;

5 Conditions on messages;

6 Multiplicity;

7 Types of message delivery;

8 Nested and interleaved messages; and

9 Protocol templates.

Following sections present these features.

Agents and Agent Roles. Agents can perform various roles within one interaction protocol. For instance, in an auction between an airline and potential ticket buyers, the airline has the role of a seller and the participants have the role of buyers. But at the same time, a buyer in this auction can act as a seller in another auction. UML precises that a role is the behavior of an entity participating in a particular context. One can also add that a role is a specific set of behaviors, properties, interfaces and service descriptions which allow to distinguish a particular role from another one.

Several terms are used in UML for representing the role classification: a *static classification* means that an agent has an unique role during all the execution. A *multiple classification* corresponds for agents to have different roles during the execution. For instance, it is the case for the previous example where buyers can act, in parallel, as buyers and as sellers. Finally, a *dynamic classification* corresponds to the ability to change from one role to another one during the execution. All these classifications appear when designing sequence diagrams.

A protocol can be defined at the level of concrete agent instances or for a set of agents satisfying a distinguished role and/or class. An agent satisfying a distinguished agent role and class is called *agent of a given agent role* and *class,* respectively. The general form of describing agent roles in Agent UML is:

`instance-1,...,instance-n / role-1,...,role-m : class`

denoting a distinguished set of agent instances *instance-1,... ,instance-n* satisfying the agent roles *role-1,... ,role-m* with $n,m \geq 0$ and *class* it belongs to. Instances, roles or class can be omitted, in the case that the instances are omitted, the roles and class are not underlined.

Instances, roles and class are textual strings. The class must correspond to an agent class. Roles are rendered within boxes on sequence diagrams. These boxes are places at the top of the diagram.

In Figure 12.1 the auctioneer is a concrete instance of an agent named *UML-Airlines* playing the role of an *Auctioneer* being of class *Seller.* The participants of the auctions are agents of role *AuctionParticipants* which are familiar with auctions and of class *Consumer.* Further information of the elements of this figure are given all along the chapter.

Agent Lifelines and Threads of Interaction. The agent lifeline in sequence diagrams defines the time period during which an agent exists, repre-

Table 12.1. Different combinations for agent names

syntax	explanation
:C	un-named Agent originating from the Class C
/R	un-named Agent playing the Role R
/R:C	un-named Agent originating from the Class C playing the Role R
A/R	an Agent named A playing the Role R
A:C	an Agent named A originating from the Class C
A/R:C	an Agent named A originating from the Class C playing the Role R
A	an Agent named A

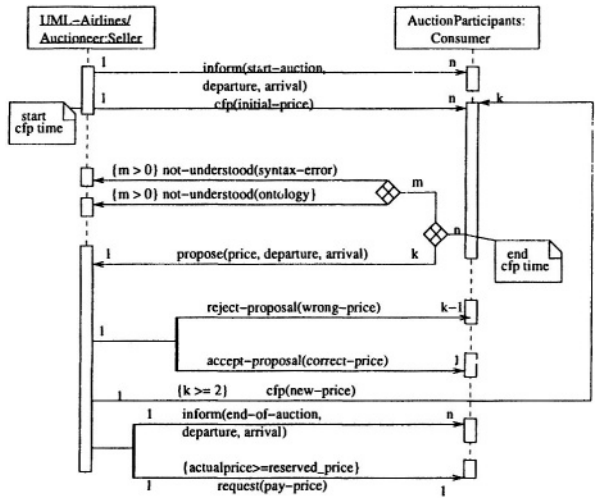

Figure 12.1. English Auction protocol for surplus flight tickets

sented by vertical dashed lines. When a lifeline is created for a role, this role becomes active in the interaction. This lifeline is present as long as the role is active in the interaction.

Added to lifelines, there are threads of interaction – UML uses term *focus of control.* The thread of interaction shows the period during which an agent is performing some tasks as a reaction to an incoming message. It only represents the duration of an action, but not the control relationship between the sender of the message and its receiver. A thread of interaction is rendered as a tall thin rectangle superposed on lifelines. If a metric is defined for the time axis, the

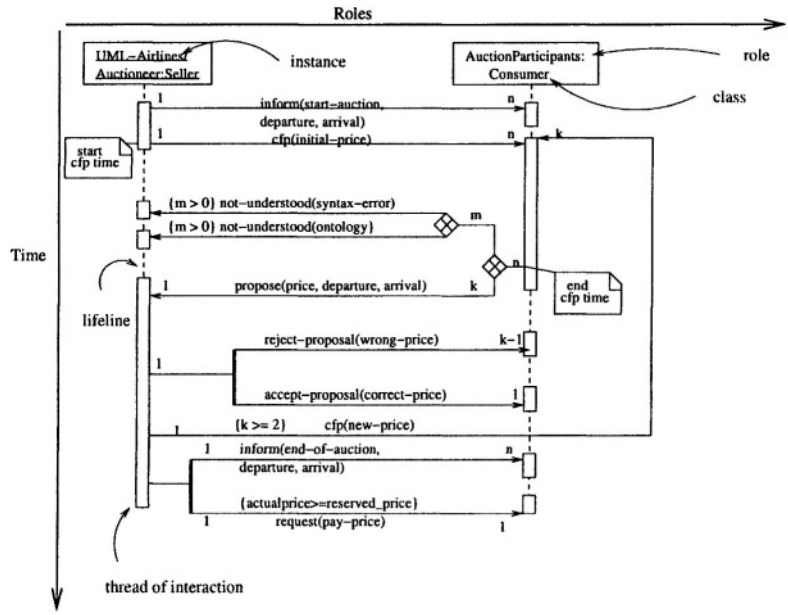

Figure 12.2. Sequence Diagram

thread of interaction corresponds exactly to the time needed to perform actions triggered by the incoming messages.

In this way, it is possible to follow the path in the interaction from the initial interaction state to the final interaction state. Actually, one has just to read the first message on the sequence diagram (messages are ordered from top to bottom), follows the transitions and uses the thread of interaction in order to find the next message. An example is shown on Figure 12.2.

The first message is the *inform* message. This message arrives to the *AuctionParticipants* but no outgoing messages are available for the thread of interaction. In fact, the next message belongs to the thread of the *Auctioneer.* This is the *cfp* message. This time, there are outgoing messages for this message. Several message traces are defined from this point. *AuctionParticipant* can answer either with *not-understood* or *propose* (see Figure 12.2).

Connectors. The lifelines may be split in order to demonstrate two kinds of behaviors: parallelism and decisions. Three connectors are supplied for these features (shown on Figure 12.3). The connector AND is rendered as a thick vertical line as shown on Figure 12.3a. It means that messages have to be sent concurrently. On Figure 12.3a, *CA-1, CA-2, CA-3* are sent in parallel. The connectors OR is rendered as a diamond as shown on Figure 12.3b and

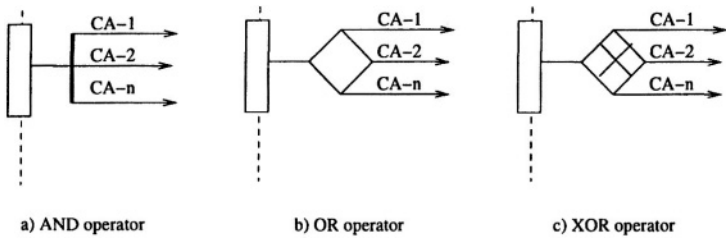

Figure 12.3. Agent UML Connectors

XOR is rendered as a diamond and a cross within it as shown on Figure 12.3c. They mean that a decision between several messages has to be done. When considering the connector OR, zero or several messages is chosen: a subset of the set {*CA-1, CA-2, CA-3*}. In the case of several messages are selected, the messages are sent in parallel. The connector XOR also represents a decision but in this case, one and only one message is chosen, it is either *CA-1* or *CA-2* or *CA-3*.

Messages. Messages in sequence diagrams are sent from a sender role to receiver roles. A message is rendered as a directed solid line. The arrowhead points out the receiver role of this message. Messages are adorned with several information as shown on Figure 12.4:

1 A textual string which is the message. If designers use FIPA ACL, messages could be the communicative act and the content.

2 Conditions which must be satisfied in order to enable an associated transition to fire. Conditions may be written as free-form text. Formal conditions are written with OCL as depicted in (OMG, 2003c). Conditions are nested by curly braces.

3 Multiplicity with two numbers placed near the sender role and the receiver role. These numbers correspond to the number of messages that are sent by the sender role and the number of instances in the receiver role that receive each message.

Several examples are given on Figure 12.1, for instance, the first message from the role *Auctioneer* to the role *AuctionParticipants* where the message is *inform*(*start-auction, departure, arrival).*

Conditions. The connectors OR and XOR are examples where a decision has to be performed to choose the next messages. A casual solution for tackling the non determinism of the sequence diagrams is to use conditions. As a consequence, a message can be sent if and only if the conditions attached to this

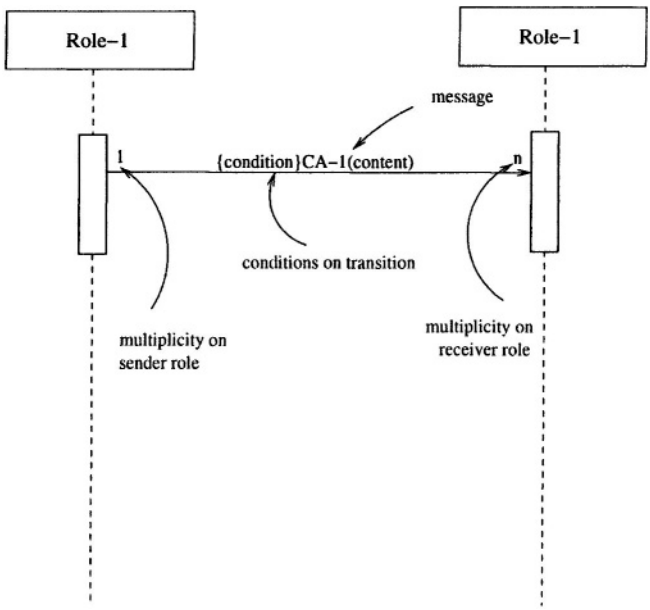

Figure 12.4. Messages in Agent UML

message are satisfied. Conditions on sequence diagrams are rendered a textual string nested by curly braces as shown on Figure 12.1 for the *not-understood* messages. The textual string can be written according to a free format or designers can use OCL *(Object Constraint Language)* defined for this purpose as depicted in (OMG, 2003c). Conditions are written just before the message on sequence diagrams as shown on Figure 12.4.

Multiplicity. Sequence diagrams represent agents either by their instances or by their role in the protocol. When using roles, it is interesting to know the number of agents involved in both the sender role and the receiver role. The cardinality for sender and receiver roles are given by the multiplicity. It is for instance the case on Figure 12.1 for the *propose* message where *k AuctionParticipants* send a *propose* to the *Auctioneer.* Sometimes, it is not possible to give the exact number of messages that are received. For instance, it is the case for the English Auction protocol since it is not possible to know how many *AuctionParticipants* will answer to the bid.

Multiplicity is adorned at both end of the transition. For instance, if the message is sent by one agent role to *n* other agents, the value 1 is written near the lifeline of the sender role and *n* is written near the lifeline of the receiver

Figure 12.5. Types of Message Delivery in Agent UML

role. Obviously, if several instances in sender role send the message, designers have to write the number corresponding to the number of agents instead of 1.

Multiplicity can be represented by a range of value: *0..1* (zero or one), *0.. ** (many) or *1.. ** (one or more). It is also possible to write several ranges of values such as *1..4, 6.. ** which means at least one message and not five messages.

Several examples are given on Figure 12.1. For instance, for the *cfp* message, the multiplicity is 1..*n*. It means that the message is sent to *n Auction-Participants.* The star could be used to denote that zero or more copies of this message are sent.

Types of Message Delivery. Messages in interaction are usually sent asynchronously (the symbol with a stick arrowhead as shown on the Figure 12.5a). It shows the sending of the message without yielding control. It is also possible to sent messages synchronously (the symbol with a filled solid arrowhead as shown on the Figure 12.5b). It shows the yielding of the thread of control (wait semantics), i.e., the agent role waits until an answer message is received and nothing else can be processed. Normally, messages are drawn horizontally. This indicates the duration required to send the message is "atomic," i.e., it is brief compared to the granularity of the interaction and that nothing else "happen" during the message transmission. If the messages require some time to arrive, for instance for mobile communication, during which something else can occur. The message is shown on Figure 12.5c.

Nested and Interleaved Protocols. Because protocols can be codified as recognizable patterns of agent interaction, they become reusable modules of processing that can be treated as first-class notions. For example, Figure 12.6 depicts two kinds of protocol patterns. The left part defines a nested protocol, i.e., a protocol within another protocol, and the right part defines an interleaved protocol, e.g., if the participant of the auction requests some information about his/her bank account before bidding. Additionally nested protocols are used for the definition of repetition of a nested protocol according to conditions. The semantics of a nested protocol is the semantics of the protocol. If the nested protocol is marked with some conditions then the semantics of the nested protocol is the semantics of the protocol under the assumption that the conditions evaluate to true, otherwise the semantics is the semantics of an empty protocol, i.e., nothing is specified.

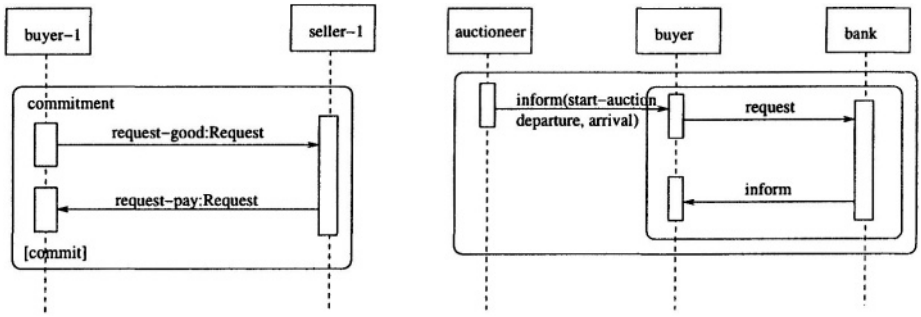

Figure 12.6. Nested Protocol and Interleaved Protocol

Protocol Templates. The purpose of a protocol template is to create a reusable pattern for useful protocol instances. For example, Figure 12.7 shows a template for the FIPA-English-Auction Protocol from Figure 12.1. It introduces two new concepts represented at the top of the sequence diagrams. First, the protocol as a whole is treated as an entity in its own right. The protocol can be treated as a pattern that can be customized for other problem domains. The dashed box at the upper right-hand corner declares this pattern as a *template* specification that identifies unbound entities (formal parameters) within the package that need to be bound by actual parameters when instantiating package. A parameterized protocol is not a directly-usable protocol because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a protocol. Communicative acts in the formal parameter list can be marked with an asterisk, denoting kinds of messages which can alternatively be sent in this context. This template can be instantiated for a special purpose as shown in Figure 12.1. Figure 12.8 applies the FIPA English Auction Protocol to a particular scenario involving a specific auctioneer *UML-Airlines* of role *Auctioneer* and class *Seller* and *AuctionParticipants* of class *Consumer.* Finally, a specific deadline has been supplied for a response by the seller.

3.2 Agent Class Diagrams

Class diagram in UML shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams are used to illustrate the static design view of a system. Generally, class diagrams contain three information: class attributes, class operations, and the relationships between classes. It is also possible to insert the class interfaces and other compart-

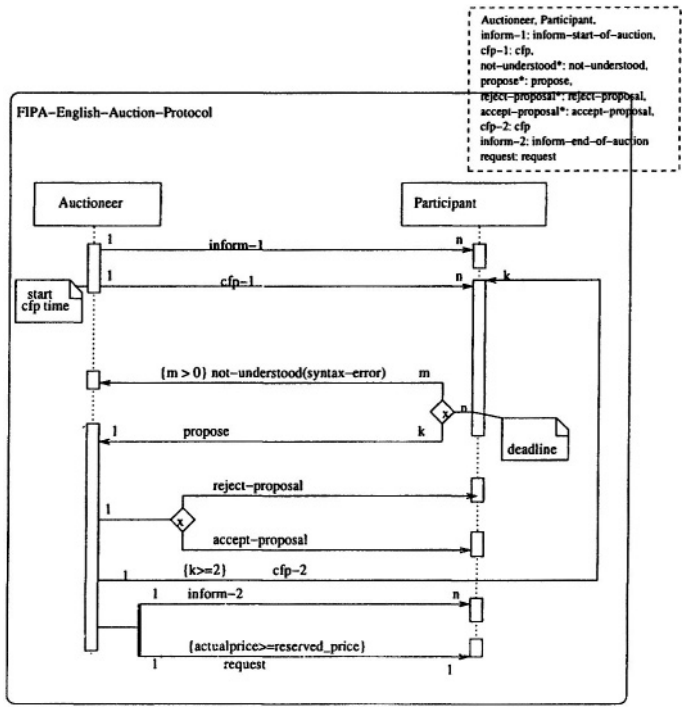

Figure 12.7. Nested Protocol and Interleaved Protocol

FIPA-English-Auction-Protocol
UML-Airlines / Auctioneer : Seller, AuctionParticipants : Consumer
start cfp time + 1 min
inform(start auction, departure, arrival),
cfp(initial-price),
not-understood(syntax error),
not-understood(ontology),
propose(pay-price),
reject-proposal(wrong-price),
accept-proposal(correct-price),
cfp(increased-price),
inform(end-of-auction),
request(pay-price, fetch-car)

Figure 12.8. Instantiation of a Protocol Template

ments in the class view such as responsibilities or exceptions. Due to many differences between agents and objects, see (Odell, 2002) for a description, class diagrams are modified deeply in order to encompass agent features such as knowledge, plans or protocols used. For pointing out the differences with class diagrams in UML, class diagrams in Agent UML are called agent class diagram as depicted in (Bauer, 2001).

UML distinguishes different specification levels, namely the *conceptual,* the *specification* and the *implementation level.* Behind these levels are the notion of abstraction. The abstraction allows designers to focus on some relevant details while ignoring others. Designers can define as many views as they want.

On the *conceptual level,* designers provide an overall view of a system: the different agent classes and their relationships without considering what could be the elements within classes. This level is particularly fitted for a description of the architecture of the system.

On the *specification level* or interface level, an agent class is blueprint for instances of agents. But only the interfaces are described and not the implementation, i.e., the agent head automata describing the behavior of the agent according to incoming messages is missing. Only the internal states and the interface, i.e., the communicative acts supported by the agent, is defined.

The *implementation level* or code level is the most detailed description of a system, showing how instances of agents are working together and how the implementation of a class of agents looks like. On this level the agent head automata has to be defined, too.

UML class diagrams are not defined here. Detailed description of UML class diagrams are given in (Booch et al., 1999).

Agent class diagrams as defined in (Bauer, 2001) contain several elements:

- Agent name;
- State description;
- Actions;
- Methods;
- Capabilities, service description, supported protocols;
- Organization belonging; and
- Agent head automata.

Agent Name. Agents are different from objects so it is necessary to make distinction when agents and objects are both used on the same diagram. This

is the case when agents are defined as a set of objects or when they use objects to perform their tasks. The stereotype «agent» prefixes agent name.

Three information may be supplied for an agent name: instance, role and class. Instances, roles and classes correspond to three levels when considering agents. Class is the most general one. A class is a set of objects that share the same set of attributes, operations and relationships and have the same semantics in UML. The definition is extended for Agent UML in order to take into consideration plans, knowledge or protocols. A role is the behavior associated to an entity into a particular context. For instance, two roles are usually described in auctions: seller and bidders. The role defines how agents react to events. Two different roles have two different behaviors. Instances are the most accurate information. Instances give the name of each agent involved. Instances are unique, i.e., if one has two instances A and B, it is not possible to use the instance A instead of the instance B and vice-versa. The general form of describing agent roles in Agent UML is:

`instance-1, ..., instance-n / role-1, ..., role-m : class`

denoting a distinguished set of agent instances *instance-1,... ,instance-n* satisfying the agent roles *role-1,...,role-m* with $n, m \geq 0$ and *class* it belongs to. Instances, roles or class can be omitted, in the case that the instances are omitted, the roles and class are not underlined.

State Description. The state description looks similar to the attributes compartment in class diagrams except that we introduce a distinguished class *wff* for *well formed formula* for all kinds of logical descriptions of the state, independent of the underlying logic. With this extension we have the possibility to define as well BDI agents.

In the case of BDI semantics, one can define four instance types, named *beliefs, desires, intentions* and *goals,* each of type *wff.* These fields can be initialized with the initial state of BDI agents.

Attributes follow the same construction as for attributes in UML:

```
[visibility] name [multiplicity] [:type]
   [= initial-value] [property-string]
```

Visibility defines how an attribute can be seen and used by others. Three cases are available: *public, private* and *protected.* The public visibility means that other classes can access this attribute. Public visibility is denoted by the symbol '+'. The private visibility means that only attributes and operations in the same class of this attribute can access it. Private visibility is denoted by the symbol '-'. The protected visibility means that the class itself and all descendant of this class can access it. Protected visibility is denoted by the symbol '#'.

Name is the name of the attribute. It is a textual string. The name must be unique within the class.

Multiplicity is used when it is necessary to represent several copies of the same attribute. There are two methods for describing multiplicity: either with a number or with a range of values. Designers use numbers or ranges of values whether they know exactly the number of copies or not.

Type represents the type of the attribute.

Initial-value describes the initial value of this attribute.

Property-string defines how attributes can be used: *changeable* is the default value and means that it is possible to update the value of this attribute, *add-only* is used for lists and means that only the insertion is possible, it is then not possible to update or to delete values in the list, *frozen* corresponds to constants, the value of the attribute cannot be modified, *persistent* denotes that the value is persistent.

Actions. Two kinds of actions can be specified for an agent: pro-active actions (denoted by the stereotype «`pro-active`») that are triggered by the agent itself, e.g., using a timer, or a special state is reached. It is tested on state changes of the agent (e.g., timer, sensor input) if the pre-condition of the actions evaluates to true. Re-active actions (denoted by the stereotype «`re-active`») are triggered when receiving some message from another agent.

In its full form, the syntax of an action is:

```
[visibility] [pre-conditions] name [(parameter-list)]
                    [post-conditions]
```

Visibility has the same definition as the one shown before. *Name* is a textual string.

Parameter-list contains both the name of the parameter and its type.

Pre-conditions are constraints that must be true when an action is invoked.

Post-conditions are constraints that must be true at the completion of an action.

Pre-conditions and post-conditions may be written as a free-form text or as an OCL expression as expressed in (OMG, 2003c).

Methods. Methods are defined like operations in UML. An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class.

In its full form, the syntax of a method is:

```
[visibility] [pre-conditions] name [(parameter-list)]
```

```
[: return-type] [post-conditions] [property-string]
```

Visibility has the same definition as the one before.
Name is a textual string.
Parameter-list is defined as follows:

```
[direction] name : type [= default-value]
```

Direction may be any of the following values:

- `in`: an input parameter; may not be modified;
- `out`: an output parameter; may be modified to communicate information to the caller; and
- `inout`: an input parameter; may be modified.

Type and *return-type* represent the type of the action and of the parameter respectively. Default-value describes the default value of the parameter.

Pre-conditions are constraints that must be true when an action is invoked.

Post-conditions are constraints that must be true at the completion of an action.

Pre-conditions and post-conditions may be written as a free-form text or as an OCL expression as expressed in (OMG, 2003c).

Capabilities, Service Description and Supported Protocols. Capabilities describe what agents can do. UML does not supply capabilities in its diagrams but capabilities is close to the notion of responsabilities in UML. These responsabilities are represented as a free-form text.

Service description has to be linked to interface in UML. An interface in UML is a collection of operation that are used to specify a service of a class or a component. The operations in the interface do not have attributes. These operations are considered as entry points for the class linked to these operations. One clear advantage of interfaces is that designers do not have to modify classes linked to an interface as soon as this interface is modified. Classes remain the same as long as interfaces keep the same name. Service descriptions are rendered as an interface name and a "lollipop" linked to the class. This rendering does not describe the operations of the service description. Service descriptions are represented with their operations as a class but prefixed by the keyword `«service»`.

Supported protocols are described as a list. Supported protocols are adorned with the roles played by the agent in these protocols.

Group Representation. Agents do not act solely. They belong to at least one group and play one or several roles in this group. The compartment *organization* gives the different groups in which the agent evolves, which roles it plays and under which constraints, it can evolve in these groups. The syntax for this information is the following:

```
[constraint] organization : role
```

Constraints are written as a free-form text or as an OCL expression. Constraints must be satisfied if agents want to belong to this group.

Role matches the roles defined for this agent in the agent name.

Agent Head Automata. The agent head automata defines the behavior of an agent's head. Agents are composed of three parts: communicator, head and body.

The *agent communicator* is responsible for the physical communication of the agent. The main functionality of the agent is implemented in the *agent body*. This can be, e.g., an existing legacy software which is coupled to the MAS using wrapper mechanisms.

The *agent's head* is the "switch-gear" of the agent. Its behavior has to be specified with the agent head automata. Especially, this automata related to the incoming messages with the internal state, actions, methods and the outgoing messages, called the reactive behaviors of the agent. Moreover, it defines the pro-active behaviors of an agent, i.e., it automatically triggers different actions, methods and state-changes depending on the internal state of the agent. An example of pro-active behavior is to do some action at a specific time, e.g., an agent migrates at pre-defined times from one machine to another one, or it is the result of some request-when communicative acts.

4. Future Directions in Agent UML

2003 will be the year of deep modifications in Agent UML thanks to the issue of the new UML 2.0 defined in (OMG, 2003b) and the growing interest in Agent UML. Most of the Agent UML work is now being performed within FIPA's Modeling Technical Committee (TC). An important goal of the Modeling TC is to be domain independent. Currently, the TC is examining those area where its members have expertise: Service-Oriented Architecture (SOA), Business Process Management (BPM), simulation, real-time, AOSE, robotics, information systems. Other areas will be examined over time as additional expertise becomes available.

Two diagrams will initially be part of the FIPA specification. The first is the interaction diagram. This diagram consists of sequence diagrams, communication diagrams (previously called collaboration diagrams in UML 1) and the interaction overview diagram. The second is the agent class diagram, which

will inspired by the UML class diagram and extended to express agent-based notions. One can think that this emerging effort around Agent UML is the first step to a better Agent UML comprising several new diagrams and tools to exploit them. This section describes possible future directions of work for Agent UML community.

4.1 Diagrams

Section 3 describes the current version of Agent UML at time of writing this chapter where two diagrams are considered: *sequence diagrams* to represent interaction between agents and *agent class diagrams* to represent agents and agent architectures. From the first work in 1999 till 2002, this specification remains as proposed but the growing interest around Agent UML mostly the last two years gave birth to an effort to update this specification and add new diagrams, some derived from UML diagrams, some specifically tailored for agent needs. The year 2003 is the year of deep modifications in Agent UML partly due to the issue of a new version of UML called UML 2.0 defined in (OMG, 2003b). This new version of UML enhances the dynamic models and particularly the *interaction diagrams* that merge several diagrams and especially, sequence diagrams. These interaction diagrams now more clearly outline the different traces in an interaction through the CombinedFragments described in (OMG, 2003b). As a consequence, Agent UML sequence diagrams are hence called interaction diagrams. The current version of Agent UML interaction diagrams can be found on Agent UML Web site `http://www.auml.org`. Since this is an ongoing work, we here just depict the main modifications in Agent UML interaction diagrams.

Sequence Diagrams The interaction is still enclosed in a frame but this time, a label is added with the protocol name prefixed by the keyword sd as shown on Figure 12.9. Moreover, template protocols are more clearly distinguishable since the stereotype «`template`» is written in the label whereas, in the current version, designers have to figure out that this is a template since there is a dotted box overlapping the diagram in the upper right corner.

Lifelines are deeply modified: in the current version, agents are depicted through their identity and their roles in the interaction, a role can be suffixed by the class of the agent. In the new version, agents are depicted through their identity and their roles but groups replace classes. Groups are attached to roles. One found important to represent that a role can have a different behaviour if it is involved in two different groups. Role cardinality is also included in the box on top of the lifeline. Finally, the main difference about the lifeline is the ability for agents to change of roles or to add and delete roles during the interaction, that is to say an agent winning an auction can move to the role *winner* if needed. Messages are slightly modified to take into consideration when mes-

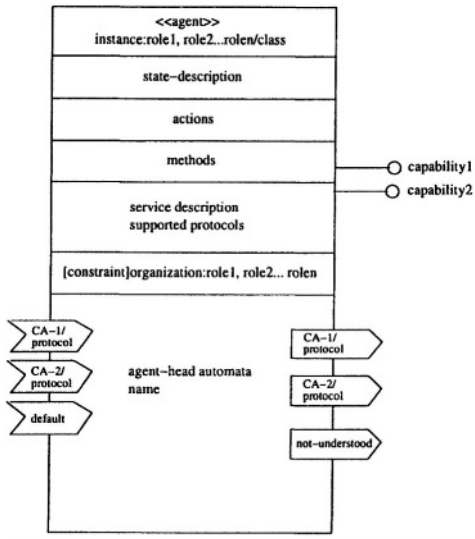

Figure 12.9. Contract Net in AUML 2

sages are sent to a specific agent or when messages are sent to the same lifeline (the sender is whether receiver or not). Timing constraints are now added to interaction diagrams allowing designers to represent deadlines like in Contract Net depicted in (Davis and Smith, 1983). Finally, the main difference between the current version and the new version remains in the splitting/merging path feature. In the current version of sequence diagrams, only three connectors are available: AND (for parallel sending), OR (1 or several messages are sent) and XOR (1 and only 1 is sent). This choice is rather restricted, the new version of Agent UML interaction diagrams also considers the notion of loop, break in the interaction to name a few, and the ability to combine different operators.

Class Diagrams A second effort is processed around agent class diagrams in order to represent a wide range of agents and partly due to some critics about the cumbersome of the notation and some inefficient choices as showed in (Huget, 2002a). At time of writing, the agent class diagram specification is not enough advanced to be presented here. The draft version of the new Agent UML class diagrams can be found on Agent UML Web site as well.

It is worth noticing that these two specifications are about to be standardized at FIPA.

Various Diagrams These Interaction and Class diagrams are the first ones that will be standardized at the FIPA. Extensions to the other UML represen-

tations are planned. Currently, these include: packages, templates, activity diagrams, class diagrams, deployment diagrams, and state machines. As mentioned earlier, the FIPA Modeling TC activities are not limited to UML for their inspiration: we intend to reuse of UML wherever it makes sense. There is already some research being conducted to model which UML does not currently address. For instance, roles and groups are now playing a more important role in interaction diagrams. The need for a role diagram that expresses how agents assume and change roles is vital. A second kind of diagram can emerge to deal with goals and planning. These are just two examples among others of possible diagrams in Agent UML. Perhaps all the UML diagrams will be considered and "agentified" if appropriate. The efforts of the FIPA Modeling TC are ongoing and therefore not fully planned. The Modeling TC cannot fully plan, because it is still too early to know exactly what will be required to model agent-based systems. The field of agents and agent-based systems is still in its early stages. As the agent community understands better how to think, communicate, and represent its notions of agents, we will then be develop an richer and more expressive AUML.

4.2 Tools

All specifications are worthless if no tools are provided to help designers representing, exchanging diagrams and generating code from these diagrams. Unfortunately, this point is not really considered for the moment for Agent UML. Two options are possible: *(i)* updating an existing tool; or *(ii)* creating from scratch a tool dedicated to Agent UML.

The former option is certainly the less time consuming. Two kinds of tools are considered for this approach: tools that support UML and tools that simply draw diagrams. In the second case, it is straightforward to propose an extension to support Agent UML; some work is done for the moment around Dia (see `http://www.lysator.liu.se/~alla/dia`) and Microsoft Visio. The main drawback of this approach is the absence of integrated environment to generate code and check diagrams. Problems happen with tools that support UML since UML 2.0 is a young specification and tools do not support it for the moment, as a consequence we cannot extend them to support Agent UML. The advantage of such tools is the generation of code and the diagram validation. OpenTool (see `http://www.tni-valiosys.com`) supports UML and Agent UML in its current version.

Creating a tool from scratch to support Agent UML is certainly the most time-consuming task but it offers a tool that perfectly answers to Agent UML needs. Maybe some tools will appear in near future as soon as projects and applications will consider Agent UML as notation.

Our first work will be to extend current tools that support UML 2.0 in order to support Agent UML as well. As soon as Agent UML will become stronger and used, the solution will be to tailor a specific tool that offers diagram modeling, code generation and validation.

4.3 Algorithms

We have already advocated the needs for tools that consider both diagram modeling, code generation and validation. Code generation and validation of such diagrams in the context of MAS have to be defined. Some work already exist for the current version of sequence diagrams, see (Huget, 2002c; Koning and Romero-Hernandez, 2002), but the same kind of effort has to be applied to the new specification of interaction diagrams. Except the work in (Huget, 2002b) where a Java program is generated from a sequence diagram, there is no work around code generation.

Important work has to be performed around agent class diagrams since nothing for code generation and validation exists for the moment.

As soon as the interaction diagram and the agent class diagram specifications will be accepted, we will move part of our effort to provide such algorithms for the validation of these diagrams and for the generation of code.

4.4 Semantics

A critic frequently encountered both for UML and Agent UML is these two notations are not formal. We understand this concern and are conscious that this can cause some misunderstandings when modeling agents or protocols if this interpretation is not clearly defined. We actively work at the definition of a semantics for Agent UML leveragging the critics on formalism in Agent UML.

4.5 Applications

Applications have to be considered as first-class citizens in the near future of Agent UML. Actually, there is no important application of Agent UML for the moment. The design of applications will arise problems and lacks in the specification and will offer tools for the design of Agent UML.

5. Conclusion

MAS are often characterized as extensions of object-oriented systems. This overly simplified view has often troubled system designers as they try to capture the unique features of MAS systems using OO tools. In response, the agent-based unified modeling language – Agent UML – is being developed. Instead of reliance on the OMG's UML, we intend to reuse of UML wherever it makes sense. We do not want to be restricted by UML; we only want to

capitalize on it where we can. The general philosophy, then, is: when it makes sense to reuse portions of UML, then do it; when it does not make sense to use UML, use something else or create something new. Upon closer inspection many of the new breakthroughs of UML 2.0 reflect the requests of the agent community – making AUML less of an extension to UML 2.0 than UML 1.0.