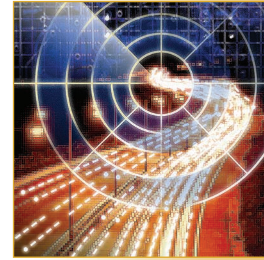

Agent UML Notation for Multiagent System Design

From the earliest days of multiagent system development, the need has existed for both a methodology and a modeling notation that assist in design. The Agent UML community has responded by developing the AUML notation — a UML profile dedicated to agents that tries to simplify the transition from software engineering to multiagent system engineering.

Marc–Philippe Huget
Leibniz–IMAG/MAGMA

Many modeling notations, ranging from temporal and linear logic to UML-based notations such as MESSAGE/UML[1] have emerged for multiagent system development since Shoham's seminal first work.[2] UML[3] offers many advantages because it is widely used in industrial software projects and tools are available for this notation. Moreover, nonmathematician designers can understand it. As Odell and colleagues indicated, starting over with a new modeling language for agents would be neither useful nor productive.[4] Instead, multiagent systems would benefit from an incremental extension of existing trusted methods. Agent UML provides such a solution. The idea behind AUML is to exploit UML extension capabilities such as stereotypes and constraints. AUML crystallizes a growing concern for agent-based modeling representations and lets designers move smoothly from software development to agent development.

AUML first appeared in 1999 in Bauer's proposal on interaction protocols.[5] Between 1999 and 2002, the AUML community proposed three specifications (agent interaction protocols, agent class diagrams, and social structures). Today, the community is pushing to define AUML with the UML 2.0 specification and to standardize it within the Foundation for Intelligent, Physical Agents (www.fipa.org). This article presents the FIPA modeling technical committee's current efforts and future agenda for AUML.

Agent design with AUML is still developing, so we only present it in part. Please visit the AUML Web site (www.auml.org) for the most up-to-date materials.

 1089-7801/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

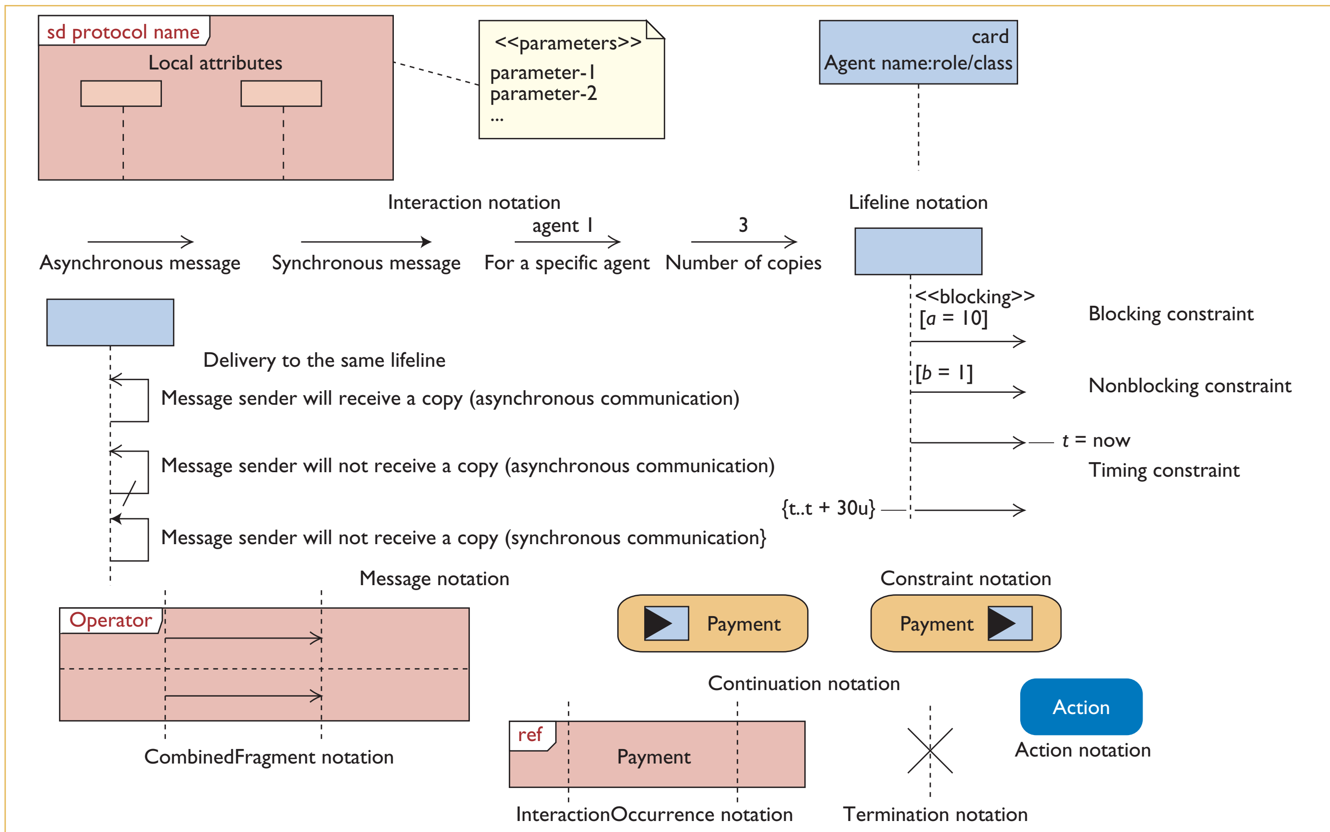

Figure 1. The Agent UML sequence diagram notation. This notation is based on the UML 2.0 Interaction specification. except for the message-with-copy notation, which is from AUML.

Representing Agent Interaction Protocols

Interaction protocols were the first part of multi-agent system design that the AUML community considered.[6] The original idea was to exploit UML 1.x sequence diagrams to represent message exchange between agents; Odell and colleagues refined the model to include agents, roles, and classes in agent communication. An agent is typically depicted via its identity or role within an interaction. The class level prefigures the notion of patterns for agent development — that is, a set of behaviors agents have if they belong to this class. A detailed description of the first version of AUML sequence diagrams appears elsewhere.[6] This first version is used in the FIPA specifications.

UML 2.0 and its new package for interactions, developed by OMG and released in 2003, have changed how we define message exchange. Four kinds of diagrams are now available:

- *Sequence diagrams* focus on message sequence.
- *Interaction overview diagrams* generalize the control flow based on activity diagrams.
- *Communication diagrams* focus on object relationships in which message passing is central.
- *Timing diagrams* (also called *interaction diagrams*) show how the interaction changes in terms of state and condition over linear time.

Figure 1 summarizes the different notations used in interaction diagrams.

Interaction

Figure 2 shows an example AUML sequence diagram. As in UML 2.0, it is organized around an *interaction* frame, which is represented by a rectangle with a pentagon in the upper-left-hand corner. An interaction frame is defined as a unit of behavior and contains the protocol name (prefixed by the keyword `sd` for *sequence diagram*), the related set of objects, and the sequence of messages between those objects. A sequence diagram can represent a protocol instantiation or a protocol template. (In the latter case, the protocol name is prefixed with `<<template>>`).

Contrary to UML 2.0, parameters in AUML are not written in the pentagon in the upper-left-

hand corner of the interaction frame; rather, they appear outside the sequence diagram in a comment. (The comment for parameters is prefixed with `<<parameters>>`, see example in Figure 1, upper-left notation.) This placement reduces the sequence diagram's size: protocols can have many parameters, such as the content language for messages, the agent communication language, or the ontology.

Lifeline

A *lifeline* in AUML describes a participant or role's appearance in an interaction. It is then possible to retrieve the messages sent and received by this lifeline: they correspond to this lifeline's outgoing or incoming transitions. A participant that enters the interaction later than other participants has a shorter lifeline, as does one that leaves earlier. The lifeline notation symbol is a rectangle followed by a vertical line that represents the participant's lifeline (see the lifeline notation in Figure 1). The rectangle can contain the agent's identity, its role, its group, and cardinality. (Cardinality constrains the number of participants in open interactions.)

Message

Agents interact via messages sent from their lifeline to the receiver's lifeline, both of which can be identical. If this is the case, we can verify that the sender will not receive a copy of the message by crossing it out. (A labeled, direct arc from the sender's lifeline to the receiver's lifeline depicts a message.) Most messages in agent communications are sent asynchronously, but messages in AUML can be sent synchronously as well (see the notation on Figure 1).

Constraint

Constraints depict a condition that must be evaluated to true to execute the associated actions (send a message, receive a message, perform another turn of a loop, and so on). AUML has two kinds of constraints: *blocking/nonblocking* and *timing*. Blocking constraints keep the lifeline from participating until the constraints are satisfied (see the blocking constraint notation in Figure 1). A blocking constraint is prefixed with `<<blocking>>`; a nonblocking constraint doesn't block execution in the interaction. If the constraints are invalid, the associated messages are not sent or received. Blocking and nonblocking constraints are written in square brackets; timing constraints are equivalent to those in UML 2.0, meaning that timing constraints correspond to a delay between two messages (see the example between the `cfp` message and the alternative in Figure 2).

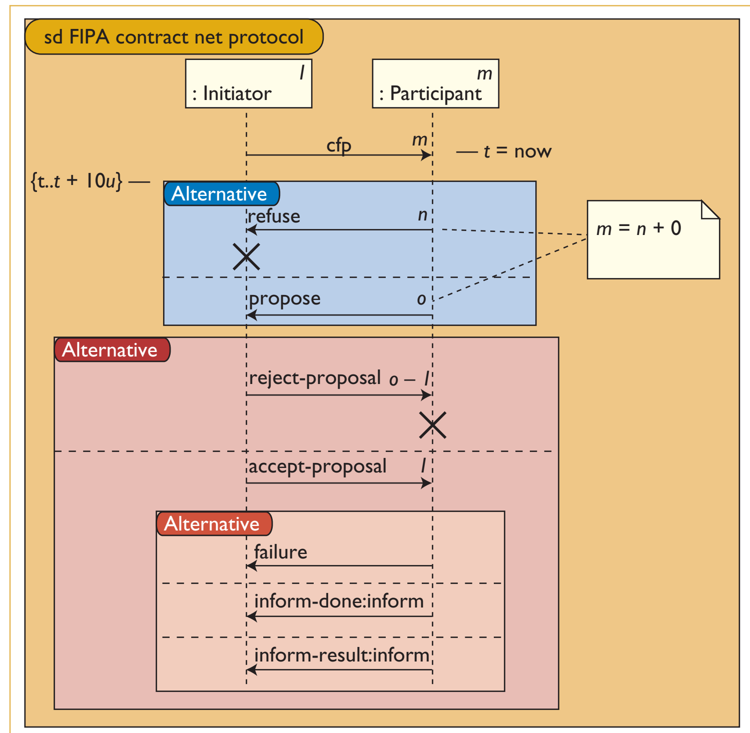

Figure 2. The contract net protocol. In this protocol, an agent (the submitter) is unable to perform a task. It offers it to other agents, which make proposals. The submitter awards the task to an agent based on its proposal and requests it to do the task.

CombinedFragment

UML 2.0's CombinedFragment class offers a concise way to describe multiple traces. It appears in the sequence diagram as a rectangle with a rounded-rectangle in the upper-left-hand corner that contains an InteractionOperator in it (see Figure 1). A dashed line separates each trace. The semantics of the CombinedFragment depends on the InteractionOperator used: alternative, option, break, parallel, weak sequencing, strict sequencing, negative, critical region, ignore, consider, assertion, or loop. Detailed descriptions of these operators appear elsewhere.[7]

Continuation

A continuation syntactically represents the different branches of a CombinedFragment. Continuations are intuitively similar to labels, representing intermediate points in a control flow. The notation is a rounded rectangle with a name in it that corresponds to the label. A continuation in which a

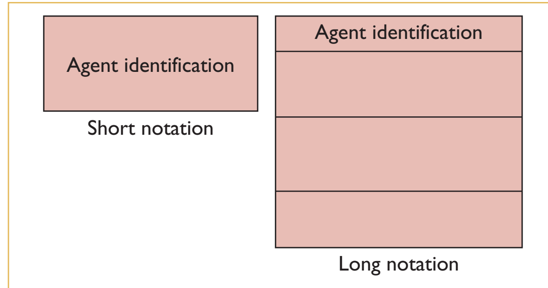

Figure 3. Agent UML agent diagram. The figure presents the two notations for the diagram: short (just identification) and long (all parts of the agents).

black-filled triangle appears before the label indicates the continuation's entrance. If the triangle appears after the label, it corresponds to a call to a continuation.

InteractionOccurrence

An InteractionOccurrence corresponds to a call to another interaction diagram. The caller's interaction diagram resumes when the callee's interaction diagram ends. The InteractionOccurrence's notation is a rectangle with a pentagon in it that says `ref`; the called interaction diagram's name is written inside the rectangle.

Gate

A gate is a connection point that adds frame link messages inside and outside the interaction. Such points on the frame exist on both sides of the message — the incoming and outgoing endpoints.

Termination

A termination denotes the end of a lifeline's participation in the communication and is depicted with an X.

Protocol Template

A protocol template is depicted when `<<template>>` appears between the keyword `sd` and the protocol name. Parameters in protocol templates can be unbound when no value is actually associated to these parameters; in such cases, the templates are prefixed with `<<unbound>>`.

Action

Agents use agent communication languages to interact with other agents. Such languages have a semantics that describes what the agents are supposed to do before sending and after receiving a message. In FIPA's Agent Communication Language, for instance, sending an `inform` message implies that the sender believes the message's contents and that the receiver doesn't know these contents. An action is depicted with a rounded rectangle linked by an association to the message that triggers it (see Figure 1). The action is written as text independent of any programming language.

Designing Agents

Class diagrams typically drive object system design. Although class diagrams can't express all the richness of agent behaviors (in particular, autonomy and goal-driven execution), Bauer proposed enhancing the diagrams to include such features as turning class diagrams into agent-class diagrams.[8]

The AUML community is considering a new way to represent agents: defining an agent shell as a UML classifier and letting designers fill this shell with building blocks that represent specific features such as actions, events, or protocols. The aim is to use this agent shell to design reactive and cognitive agents.

In this approach, agents are defined via agent diagrams like the one in Figure 3. In this figure, we see an empty shell in which the developer will integrate building blocks to give semantics to the agent. For instance, the agent's behavior is defined by relating events to actions. Two notations are available: the shorter version includes the agent's identity and its groups, roles, and services; the longer one describes the agent's complete structure.

The design process depends on whether we're considering *reactive* or *cognitive* agents. Let's first look at the process of designing reactive agents:

- *Group and role assignment.* Designers assign groups and roles to the agent; the roles will correspond to the agent's specific behavior.
- *Event definition.* Designers extract from the environment the set of events to which the agent can react.
- *Event–action associations.* Designers associate actions to events to react to these events.

Cognitive-agent design follows a similar, but different, process:

- *Group and role assignment.* Designers begin by assigning the agent to groups and defining the different roles the agent plays in them. The groups and roles provide several pieces of

information such as behavior, services, and protocols. This first step provides a template for the agent's overall structure.

- *Service description.* Designers then describe the different services the agent offers — whether from groups, roles, or the agent itself.
- *Protocol description.* Protocols support services for cooperation and coordination or simply facilitate information exchange. The developer extracts protocols from the service definition and from roles.
- *Events.* Agents react to the events in their environments. In this step, the developer gathers all the events to which the agent can react.
- *Goals, plans, and actions.* Processing services, using protocols, and reacting to events imply defining goals, plans, and actions. Thus, the designer attempts to relate goals, plans, and actions to services, protocols, and events.
- *Knowledge.* Cognitive agents have knowledge (information, beliefs, user models, acquaintances, and so on). In this step, the designer fills the agent shell with the knowledge to perform actions and protocols.

In the rest of this section, I'll look more closely at the building blocks for cognitive agent design. I won't go into too much detail about knowledge here, but a more in-depth look at using UML for knowledge and ontologies appears elsewhere.[9] The DAML UML enhanced tool (DUET, http://grcinet.grci.com/maria/www/CodipSite/Tools/Tools.html) lets visitors experiment with translating DAML+OIL specifications into UML class diagrams and vice versa.

Group and Role Assignment

The first step in designing cognitive agents is to assign groups and roles to each agent. The designer defines these groups and roles; the agent has, at its initialization, a compartment of long notation and a complement to the agent identifier in the short notation. The notation in the compartment `Groups,Roles` is `{Group, (Role + )}*`. This means that an agent can play several roles within a group. The short notation, `Agent id:Roles/Group`, can't be used when agents are part of multiple groups; it can represent only a unique group's roles. Figure 4 shows a bookstore example: an agent called Smith belongs to the group Bookstore and takes the role of Seller.

Assigning groups and roles adds information to the agent:

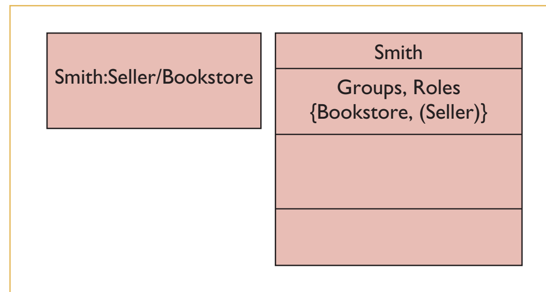

Figure 4. Agents with groups and roles. This figure presents two notations. The short notation reveals only the agent's identity, group, and the roles it plays in this group. The long notation also reveals the parts of the agent via compartments.

- *Services* represent what agents can do and how to request their services.
- *Protocols* support services, service requests, and communication; they are common for a given role or group.
- *Goals* (also called desires) and *plans* are frequently common across agents with the same role or group. All agents with the role Seller aim to maximize the company's income, for example.
- *Knowledge* can be defined at both group and role levels. It corresponds to pieces of knowledge or the agents' capabilities.
- *Norms* can be defined at the group level, such that each agent belonging to the group conforms to the group's norm. In an electronic institution, for example, the norm might be, "the winner of an auction pays the hammer price to get the item."

Groups and roles don't provide a static view of the system because agents can change or stop roles or change from one group to another. We don't integrate the diagram — rather, we just name it and add that name in the agent shell's *role dynamics* compartment. When the developer implements the agent, it refers to this diagram and retrieves the different roles and associated behaviors, services, protocols, and so on to generate agent code.

Services

A service represents a process that an agent can handle (sell a book, find a restaurant, and so on). We use the service-type definition employed in the AgentCities project (www.agentcities.org) to describe a service:

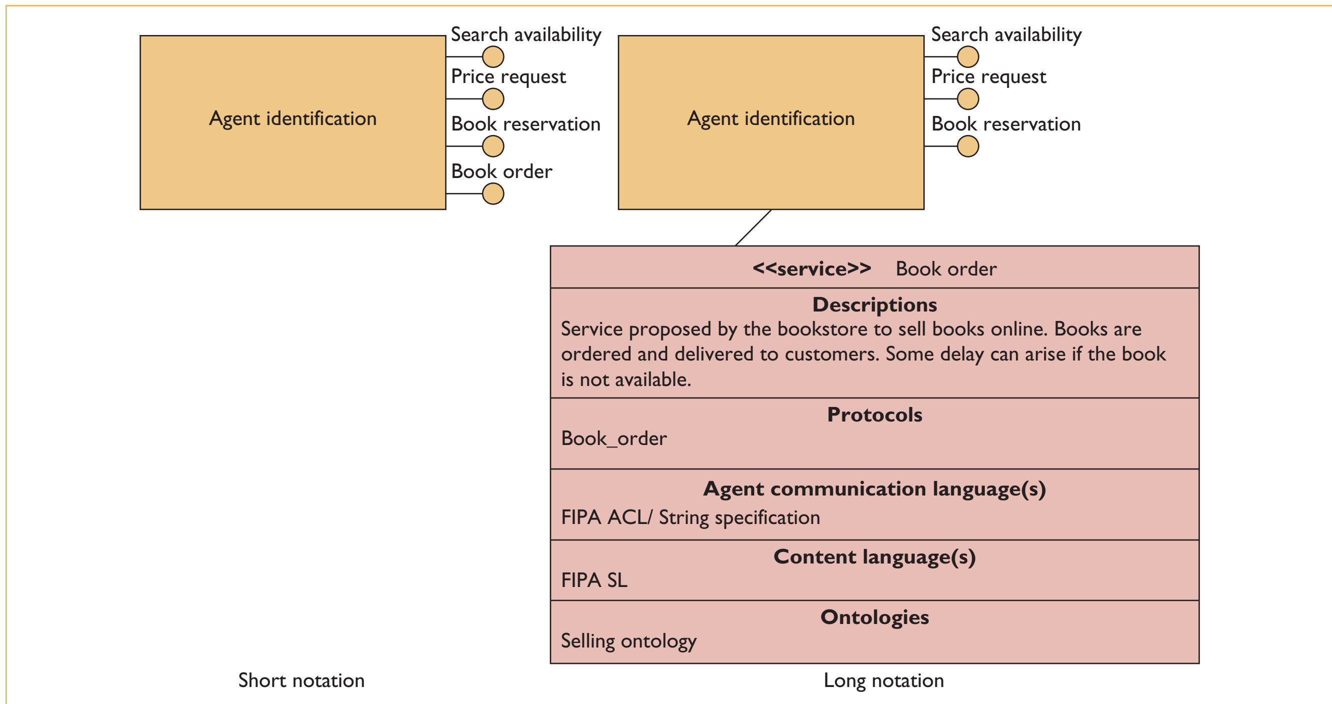

Figure 5. AUML service notation. Services are described via a short notation (a lollipop on the right side of the agent diagram) or via a long notation as a class diagram linked to the agent diagram.

- *Name* is the service's name.
- *Related to* and *sub-type-of* are other services from which this service inherits.
- *Same-type-of* is an equivalent service.
- *Descriptions* are the natural-language description of the service.
- *Protocols* invoke the service.
- *Agent communication languages* are used to interact with this service.
- *Content languages* are used by messages in the protocols.
- *Ontologies* are the ontologies this service uses.
- *DAML-S description* is the DAML-S description of the service.

As Figure 5 illustrates, a service is depicted either by a short notation in which a circle with the service's name is attached to the agent diagram via a solid line or by a long notation in which a service is described as a class diagram attached to the agent diagram via a solid line. Each compartment in the class diagram corresponds to an AgentCities service-type definition.

For our bookstore example, agent Smith has multiple services in its role as Seller: search availability, price request, book reservation, and book order. Each is supported by a protocol of the same name. Figure 5 depicts these services available for agent Smith, but due to space constraints, we see only the book order service in the long notation.

Protocols

A protocol represents the legal sequences of messages between agents. (A complete specification of agent interaction protocols appears elsewhere.[6]) A protocol is arranged around a frame that contains the interaction's participants along with message sequences.

Figure 6 shows two participants from our bookstore example: the agent role Customer and agent Smith, which plays the role of Seller within the group Bookstore. The customer orders a book from the seller, which checks the item's availability (the option box). If the book is unavailable, the seller both informs the customer and orders the book from the wholesaler. Regardless of current availability, the seller sends an invoice, which the customer pays. As soon as the book is available, the seller informs the delivery service to deliver the book.

Events

Whether an agent is reactive or cognitive, it reacts to events in its environment. In our bookstore example, events primarily affect the seller: an

ordered book arrives or a book is no longer available. Figure 7 shows how actions are associated with events and how they correspond to the agent's reaction. Events can also involve time — occurring at regular intervals, for example. Designers can use the hourglass notation, as in UML 2.0, to represent such timing events in AUML.

Goals, Plans, and Actions

The next step in the process is to define and integrate goals, plans, and actions within the agent shell — specifically, to associate actions with services, protocols, and events. A first attempt at formalizing this process, by adapting the UML 2.0 activity diagram for agent goals, plans, and actions, appears elsewhere.[10] Figure 8 summarizes the different notations for AUML goal diagram notation.

The fundamental unit in a goal is an action, which represents a specific task the agent must do. The agent itself invokes actions — such as buy an item, fill an order, ship the order — to achieve its objectives. In Figure 8, we denote each as a rounded rectangle with the action's name inside. Actions are linked via activity edges, which are solid lines that have an open arrowhead from the action performed to the action to perform. If the activity edge is named, the name appears on this line.

Pre- and postconditions can constrain a specific action's execution; they're prefixed with `<<Precondition>>` and `<<Postcondition>>` and can refer to belief, desire, and intention (BDI) modalities. A precondition might be *believe(self (a == 10))*, which means that the agent firing this activity edge needs to believe that *a* equals 10. BDI modalities can also be nested: *believe a (intends b (believe a (q == 10)))* means that *a* believes that *b*'s intention is that *a* believes *q* equals 10.

UML 2.0 activity diagrams differ from UML 1.x activity diagrams in terms of concurrency and synchronization: UML 2.0 activity diagrams resemble Petri nets and allow multiple concurrent flows. Tokens correspond to tokens in Petri nets. If an edge needs three tokens to be traversed, three tokens must be on the incoming end of the edge to traverse it. In Figure 8, tokens appear in brackets on the activity edge with the keyword *weight*; the figure shows that three tokens must be present to fire this edge.

We can use connectors when the activity diagram becomes too complicated and has too many activity edges. Instead of drawing an activity

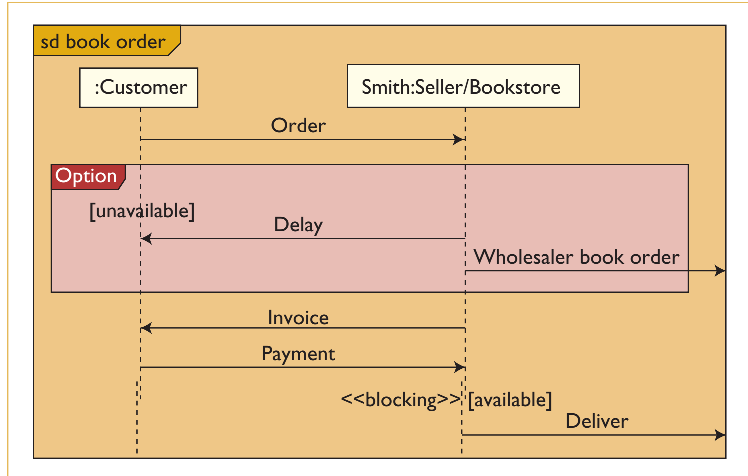

Figure 6. Example protocol. The protocol presents the book order between a customer and a seller, particularly the phase in which there is some delay receiving the book (the option base).

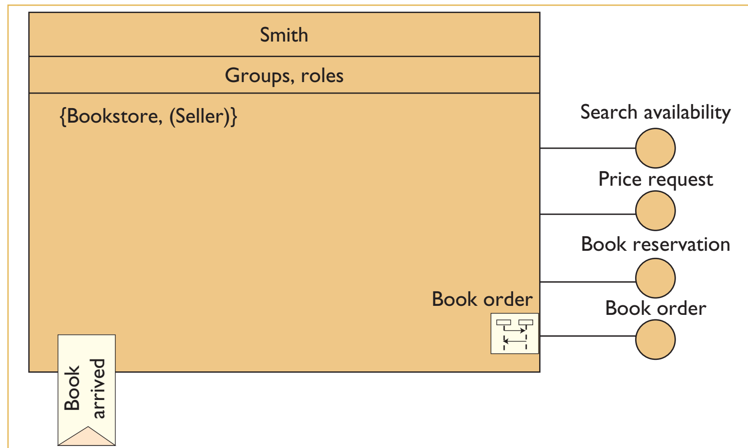

Figure 7. The event notation. The event Book arrived is described as part of this agent, which means the agent Smith can perceive this event.

edge, the developer draws a circle with the edge's name in it at both ends of where the activity edge should be. Every connector with a given label must be paired with an identical label on the same activity diagram.

Control nodes for activities are

- initial node,
- activity final node,
- flow final node,
- decision, or merge, node, and
- fork, or join, node.

An *initial node* marks the start of an activity, but a single activity can have multiple initial nodes

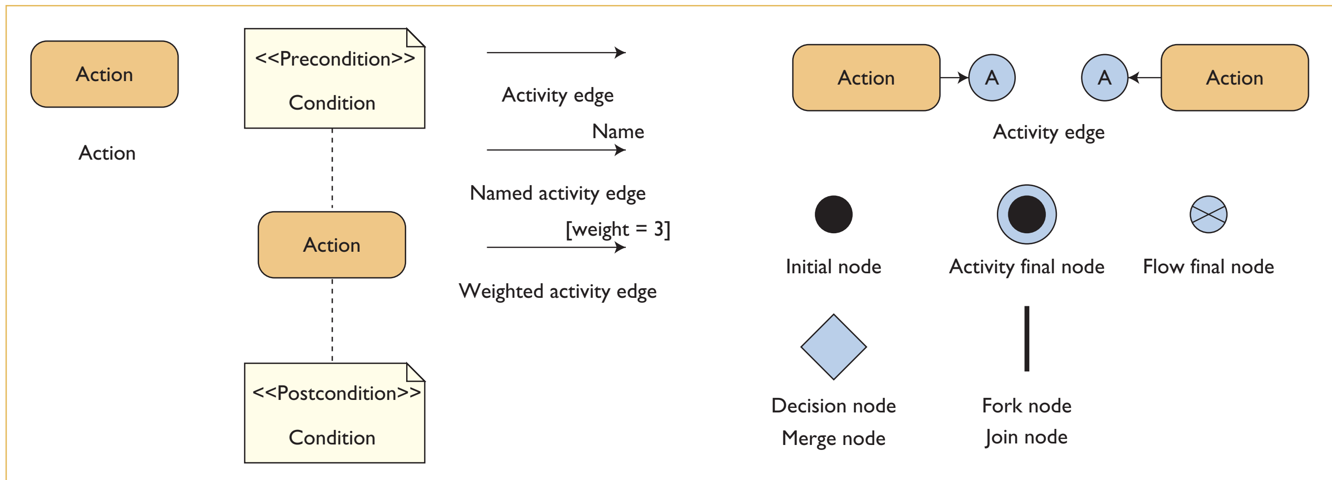

Figure 8. Goals, plans, and actions. The agent goal diagram notation is based on UML 2.0 action notation.

(several flows would be initiated, one per initial node — denoted with solid circles in Figure 8). An *activity final node* marks the end of an activity, but as with the initial node, one activity can have several final nodes. An activity final node means that all processing in the activity has stopped, whereas a *flow final node* means that only a specific flow has ended. Flow final nodes do not stop other processing.

A *decision node* chooses a node among several outgoing activity edges. Constraints can force outgoing activity edges to simplify activity edge selection; constraints also can contain BDI modalities. If a decision node has several incoming activity edges, we call it a *merge node*, but this doesn't change the decision node's semantics: it still chooses only one outgoing activity edge.

A *fork node* splits a flow into multiple concurrent flows; it has one incoming edge and several outgoing edges. A *join node* is essentially a fork node with several incoming activity edges; it is used to synchronize multiple flows.

A more detailed description of the other classes that goal diagrams consider — time management, exception handling, action atomicity, priority, and so on — appears elsewhere.[10]

The Agenda

Several diagrams are already under consideration for AUML (for goals, plans and actions, services, and groups and roles, to name a few), but the road to an efficient notation that meets FIPA's requirements is long. Let's look at some of the items on the AUML agenda.

First, AUML needs formal semantics for the different diagrams. One of UML's great advantages is that it's visual and easy to understand, particularly for software engineers. One of its main drawbacks, though, is the absence of a formal semantics for defining the different classes used in the diagrams. Consequently, it's possible to misinterpret diagrams. A formal semantics helps designers reuse diagrams and conforms tools' appearance to a specification.

AUML also needs to populate a set of diagrams. Initial work on AUML focused on interaction protocols and agent design. The newer version will provide diagrams for goals and social structures as well.

One of AUML's main drawbacks is the absence of tools dedicated to it. Designers working with it use tools such as Dia, Xfig, or Visio, but they can't check diagram consistency with these tools. This makes it easy to misuse the specification.

Finally, it is necessary to challenge the AUML notation against industrial and real-world applications to verify its completeness. Unfortunately, this is the most difficult part because it's hard to find industrial applications. Nevertheless, the AUML community is encouraging companies to participate in its efforts to develop a strong and useful notation.

Acknowledgments

The author thanks all the people working on Agent UML and those who will join the Agent UML community after reading this article.

References

1. C. Caire et al., *Message: Methodology for Agent-Oriented*

Software Engineering, tech. report, Eurescom, 2001, www.eurescom.de/~public-webspace/P900-series/P907/D1/.
2. Y. Shoham, "AGENT0: A Simple Agent Language and Its Interpreter," *Proc. 9th Nat'l Conf. Artificial Intelligence*, AAAI Press/MIT Press, 1991, pp. 704–709.
3. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
4. J. Odell, H. Van Dyke Parunak, and B. Bauer, "Extending UML for Agents," *Proc. Agent-Oriented Information Systems Workshop, 17th Nat'l Conf. Artificial Intelligence*, G.Wagner, Y. Lesperance, and E. Yu, eds., ICue Publishing, 2000.
5. B. Bauer, "Extending UML for the Specification of Interaction Protocols," submission for the 6th Call for Proposal of FIPA and revised version of FIPA 99, 1999.
6. M.-P. Huget, J. Odell, and B. Bauer, "The AUML Approach," *Methodologies and Software Engineering for Agent Systems*, Kluwer, 2004 (to be published).
7. M.-P. Huget and J. Odell, "Representing Agent Interaction Protocols with Agent UML," *Proc. 5th Int'l Workshop on Agent-Oriented Software Eng.*, P. Giorgini, J. Müller, and J. Odell, eds., Springer-Verlag, 2004.
8. B. Bauer, "UML Class Diagrams Revisited in the Context of Agent-Based Systems," *Proc. Agent-Oriented Software Eng.*, LNCS 2222, M. Wooldridge, P. Ciancarini, and G. Weiss, eds., Springer-Verlag, 2001, pp. 1–8.
9. S. Cranefield and M. Pruns, "UML as an Ontology Modeling Language," *Proc. Workshop on Intelligent Information Integration, 16th Int'l Joint Conf. Artificial Intelligence*, CEUR Publications, 1999.
10. M.-P. Huget, "Representing Goals in Multiagent Systems," *Proc. 4th Int'l Symp. Agent Theory to Agent Implementation* (AT2AI–4), P. Petta and J. Mueller, eds., Austrian Soc. for Cybernetic Studies, 2004, pp. 588–593.

Marc–Philippe Huget is a postdoctoral fellow at Leibniz–IMAG/MAGMA. His research interests include multiagent system design and interaction in multiagent systems. He received a PhD in computer science from Paris 9. He is a member of the IEEE and the ACM. Huget is the editor of *Communication in Multi-Agent Systems* (Springer-Verlag, 2003). Contact him at Marc–Philippe.Huget@imag.fr.

ADVERTISER / PRODUCT INDEX JULY/AUGUST 2004

Advertiser	Page Number
Charles River Media	15
John Wiley & Sons, Inc.	Cover 4
MIT Press	6
SAP Labs	9
WebSec 2004	Cover 2

Advertising Personnel

Marion Delaney
IEEE Media, Advertising Director
Phone: +1 212 419 7766
Fax: +1 212 419 7589
Email: md.ieeemedia@ieee.org

Marian Anderson
Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org

Sandy Brown
IEEE Computer Society,
Business Development Manager
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Advertising Sales Representatives

Mid Atlantic (product/recruitment)
Dawn Becker
Phone: +1 732 772 0160
Fax: +1 732 772 0161
Email: db.ieeemedia@ieee.org

New England (product)
Jody Estabrook
Phone: +1 978 244 0192
Fax: +1 978 244 0103
Email: je.ieeemedia@ieee.org

New England (recruitment)
Robert Zwick
Phone: +1 212 419 7765
Fax: +1 212 419 7570
Email: r.zwick@ieee.org

Connecticut (product)
Stan Greenfield
Phone: +1 203 938 2418
Fax: +1 203 938 3211
Email: greenco@optonline.net

Midwest (product)
Dave Jones
Phone: +1 708 442 5633
Fax: +1 708 442 7620
Email: dj.ieeemedia@ieee.org

Will Hamilton
Phone: +1 269 381 2156
Fax: +1 269 381 2556
Email: wh.ieeemedia@ieee.org

Joe DiNardo
Phone: +1 440 248 2456
Fax: +1 440 248 2594
Email: jd.ieeemedia@ieee.org

Southeast (product)
Bob Doran
Phone: +1 770 587 9421
Fax: +1 770 587 9501
Email: bd.ieeemedia@ieee.org

Midwest/Southwest (recruitment)
Darcy Giovingo
Phone: +1 847 498-4520
Fax: +1 847 498-5911
Email: dg.ieeemedia@ieee.org

Southwest (product)
Josh Mayer
Phone: +1 972 423 5507
Fax: +1 972 423 6858
Email: josh.mayer@wageneckassociates.com

Northwest (product)
Peter D. Scott
Phone: +1 415 421-7950
Fax: +1 415 398-4156
Email: peterd@pscottassoc.com

Southern CA (product)
Marshall Rubin
Phone: +1 818 888 2407
Fax: +1 818 888 4907
Email: mr.ieeemedia@ieee.org

Northwest/Southern CA (recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Southeast (recruitment)
Jana Smith
Phone: +1 404 256 3800
Fax: +1 404 255 7942
Email: jsmith@bmmatlanta.com

Japan
Sandy Brown
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sbrown@computer.org

Europe (product/recruitment)
Hilary Turnbull
Phone: +44 1875 825700
Fax: +44 1875 825701
Email: impress@impressmedia.com