# Extending UML for Agents


| James Odell | H. Van Dyke Parunak | Bernhard Bauer |
|---|---|---|
| James Odell Associates | ERIM Center for Elec. Commerce | Siemens |
| 3646 W. Huron River Dr. | P.O. Box 134001 | ZT IK 6 |
| Ann Arbor, MI 48103 | Ann Arbor, MI 48113 | D-81730 München |
| Tel: +1 (734) 994-0833 | Tel: +1 (734) 623-2509 | Tel: +49 (89) 6 36-5 06 54 |
| jodell@compuserve.com | vparunak@erim.org | bernhard.bauer@mchp.siemens.de |
| www.jamesodell.com | www.erim.org/~vparunak | |



## ABSTRACT

Gaining wide acceptance for the use of agents in industry requires both relating it to the nearest antecedent technology (object-oriented software development) and using artifacts to support the development environment throughout the full system lifecycle. We address both of these requirements in this paper by describing some of the most common requirements for modeling agents and agent-based systems—using a set of UML idioms and extensions. This paper illustrates the approach by presenting a three-layer AUML representation for agent interaction protocols and concludes by including other useful agent-based extensions to UML.



## Keywords

Agents, UML, interaction protocols, AUML, design artifacts, software engineering.


## 1. INTRODUCTION

Successful industrial deployment of agent technology requires techniques that reduce the risk inherent in any new technology. Two ways that reduce risk in the eyes of potential adopters are:

- to present the new technology as an incremental extension of known and trusted methods, and
- to provide explicit engineering tools that support industry-accepted methods of technology deployment.

We apply both of these risk-reduction insights to agents.

To leverage the acceptance of existing technology, we present agents as an extension of active objects, exhibiting both dynamic autonomy (the ability to initiate action without external invocation) and deterministic autonomy (the ability to refuse or modify an external request). Thus, our basic definition of an agent is "an object that can say 'go' (dynamic autonomy) and 'no' (deterministic autonomy)." This approach leads us to focus on fairly fine-grained agents. More sophisticated capabilities can also be added, such as mobility, BDI mechanisms, and explicit modeling of other agents. Such capabilities are extensions to our basic agents, that is, they can be applied where needed, but are not diagnostic of agenthood.

Accepted methods of industrial software development depend on standard representations for artifacts to support the analysis, specification, and design of agent software. Three characteristics of industrial software development require the disciplined development of artifacts throughout the software lifecycle. The scope of industrial software projects is much larger than typical academic research efforts, involving many more people across a longer period of time, and artifacts facilitate communication. The skills of developers are focused more on development methodology than on tracking the latest agent techniques, and artifacts can help codify best practice. The success criteria for industrial projects require traceability between initial requirements and the final deliverable—a task that artifacts directly support.

The Unified Modeling Language (UML) is gaining wide acceptance for the representation of engineering artifacts in object-oriented software. Our view of agents as the next step beyond objects leads us to explore extensions to UML and idioms within UML to accommodate the distinctive requirements of agents. The result is an Agent UML (AUML).[1]

Section 2 provides background information on agent design methods in general, on UML, and on the need for AUML. Section 3 introduces a layered approach to representing agent protocols in AUML. Templates and packages provide a high-level summary (Section 4), sequence diagrams and collaboration diagrams furnish alternative views of the interactions among agents (Section 5), and state diagrams and activity diagrams detail the internal behavior of individual agents in executing protocols (Section 6). Section 7 proposes some other UML extensions that represent several commonly used notions employed by the agent community.

---

[1] Note: Many of the features required by agents and agent-based systems are also—independently—cited as useful by objects and object-oriented systems. As such, many of the extensions suggested in this paper would be in harmony with many of the new features suggested for OO development.

## 2. BACKGROUND

Agent UML (AUML) synthesizes a growing concern for agent-based software methodologies with the increasing acceptance of UML for object-oriented software development.

### 2.1 Agent Software Methodologies

The agent R&D community is increasingly interested in design methods and representational tools to support the associated artifacts (see [12] for a helpful survey). Multi-Agent System Engineering was the focus of a session at ATAL'97 [5, 10, 13, 17, 19, 23, 25, 26] and the entire MAAMAW'99 [9].

A number of groups have reported on methodologies for agent design, touching on representational mechanisms as they support the methodology. Our own report [23] emphasizes methodology, as does the work by Kinny and colleagues [15, 16] on modeling techniques for BDI agents. The close parallel that we observe between design mechanisms for agents and for objects is shared by a number of authors, for example [5, 6].

The GAIA methodology [28] includes specific recommendations for notation that supports the high-level summary of a protocol as an atomic unit, a notation that is reflected in our recommendations. The extensive program underway at the Free University of Amsterdam on compositional methodologies for requirements [11], design [4], and verification [14] uses graphical representations with strong links to UML's collaboration diagrams, as well as linear (formulaic) notations better suited to alignment with UML's metamodel than with the graphical mechanisms that are our focus. Our discussion of the compositionality of protocols is anticipated in the work of Burmeister et al. [7], though our notation differs widely from hers. Dooley graphs facilitate the identification of the "character" that results from an agent playing a specific role (as distinct from the same agent playing a different role) [21, 27]. We capture this distinction by leveraging UML's existing name/role:class syntax in conjunction with collaboration diagrams.

This wide-ranging activity is a healthy sign that agent-based systems are having an increasing impact, since the demand for methodologies and artifacts reflects the growing commercial importance of our technology. Our objective is not to compete with any of these efforts, but rather to extend and apply a widely accepted modeling and representational formalism (UML)—one that harnesses insights and makes them useful for communicating across a wide range of research groups and development methodologies.

### 2.2. UML

During the seventies, structured programming was the dominant approach to software development. Along with it, software engineering technologies were developed in order to ease and formalize the system development lifecycle: from planning, through analysis and design, and finally to system construction, transition, and maintenance. In the eighties, object-oriented (OO) languages experienced a rise in popularity, bringing with it new concepts such as data encapsulation, inheritance, messaging, and polymorphism. By the end of the eighties and beginning of the nineties, a jungle of modeling approaches grew to support the OO marketplace. To make sense of and unify these various approaches, an Analysis and Design Task Force was established on 29 June 1995 within the OMG. By November 1997, a *de jure* standard was adopted by the OMG members called the Unified Modeling Language (UML).

The UML unifies and formalizes the methods of many approaches to the object-oriented software lifecycle and supports the following kinds of models:

- **static models**- such as class and package diagrams describe the static semantics of data and messages. Within system development, class diagrams are used in two different ways, for two different purposes. First, they can model a problem domain conceptually. Since they are conceptual in nature, they can be presented to the customers. Second, class diagrams can model the implementation of classes—guiding the developers. At a general level, the term class refers to the encapsulated unit. The conceptual level models types and their associations; the implementation level models implementation classes. While both can be more generally thought of as classes, their usage as concepts and implementation notions is important both in purpose and semantics. Package diagrams group classes in conceptual packages for presentation and consideration. (Physical aggregations of classes are called components which are in the implementation model family, mentioned below.)
- **dynamic models**- including interaction diagrams (i.e., sequence and collaboration diagrams), state charts, and activity diagrams.
- **use cases-** the specification of actions that a system or class can perform by interacting with outside actors.
- **implementation models-** such as component models and deployment diagrams describing the component distribution on different platforms.
- **object constraint language (OCL)-** is a simple formal language to express more semantics within an UML specification. It can be used to define constraints on the model, invariant, pre- and post-conditions of operations and navigation paths within an object net.

In this response, we are suggesting agent-based extensions to the following UML representations: packages, templates, sequence diagrams, collaboration diagrams, activity diagrams, class diagrams, deployment diagrams, and statecharts. The UML model semantics are represented by a metamodel whose structure is also formally defined by OCL syntax. OCL and the metamodel offer resources to capture the kinds of logical specifications anticipated in (for example) [4, 11, 14, 15, 16, 28], but space does not permit exploring this use of UML in this response.

## 2.3. AUML

The current UML is sometimes insufficient for modeling agents and agent-based systems. However, no formalism yet exists to sufficiently specify agent-based system development. To employ agent-based programming, a specification technique must support the whole software engineering process—from planning, through analysis and design, and finally to system construction, transition, and maintenance.

A proposal for a full life-cycle specification of agent-based system development is beyond the scope of this response. Both FIPA and the OMG Agent Work Group are exploring and recommending extensions to UML [1, 20]. In this paper, we will present a subset of an agent-based extension to the standard UML (AUML) for the specification of *agent interaction protocols* (AIP) and other commonly used agent-based notions.

Interaction protocols were chose because they are complex enough to illustrate the nontrivial use of AUML and are used commonly enough to make this subset of AUML useful to other researchers. Agent interaction protocols are a good example of software patterns which are ideas found useful in one practical context and probably useful in others. A specification of an AIP provides an example or analogy that we might use to solve problems in system analysis and design.

We want to suggest a specification technique for AIPs with both formal and intuitive semantics and a user-friendly graphical notation. The semantics allows a precise definition that is also usable in the software-engineering process. The graphical notation provides a common language for communicating AIPs—particularly with people not familiar with the agent-based approach.

Before proceeding, we need to establish a working definition:

> *An agent interaction protocol (AIP) describes a communication pattern as an allowed sequence of messages between agents and the constraints on the content of those messages.*

# 3. A LAYERED APPROACH TO PROTOCOLS

Figure 1 depicts a protocol expressed as a UML *sequence diagram* for the contract net protocol. When invoked, an Initiator agent sends a call for proposal to an agent that is willing to participate in providing a proposal. (The stick-headed arrow indicates an asynchronous communication.) The Participant agent can then choose to respond to the Initiator before a given deadline by: refusing to provide a proposal, submitting a proposal, or indicating that it did not understand. (The diamond symbol indicates a decision that can result in zero or more communications being sent—depending on the conditions it contains; the “x” in the decision diamond indicates an *exclusive or* decision.) If a proposal is offered, the initiator has a choice of either accepting or rejecting the proposal. When the participant receives a proposal acceptance, it will inform the initiator about the proposal’s execution. Additionally, the Initiator can cancel the execution of the proposal at any time. (Note: the stick arrows indicate unnested, asynchronous communications (messages). The interpretation of the term "unnested" in UML 1.3—as communicated via email from Gunnar Overgaard—is not yet well defined in a UML document. This notations can be changed, once a firm standard exists for unnested, asynchronous communications.)

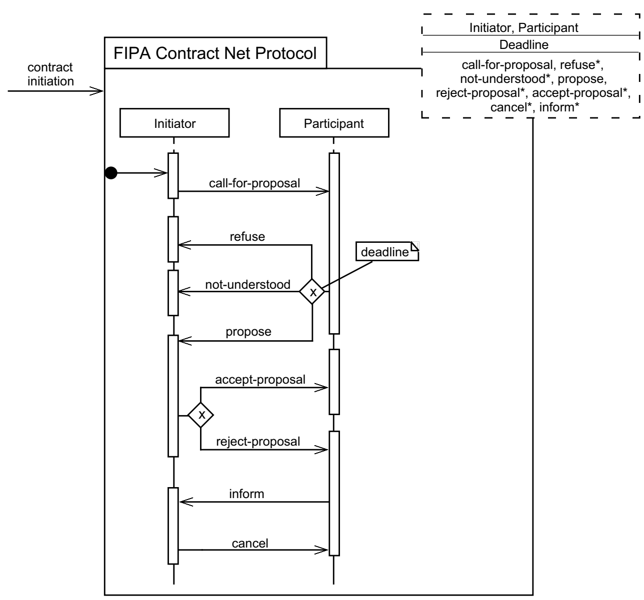



**Figure 1. A generic AIP expressed as a template package.**

This figure also expresses two more concepts represented at the top of the sequence chart. First, the protocol as a whole is treated as an entity in its own right. The tabbed folder notation at the upper left indicates that the protocol is a *package*, a conceptual aggregation of interaction sequences. Second, the packaged protocol can be treated as a pattern that can be customized for analogous problem domains. The dashed box at the upper right-hand corner expresses this pattern as a *template* specification that identifies unbound entities within the package which need to be bound when the package template is being instantiated.

The original sequence diagram in Fig. 1 provides a basic specification for a contract net protocol. More processing detail is often required. For example, an Initiator agent requests a call for proposal (CFP) from a Participant agent. However, the diagram stipulates neither the procedure used by the Initiator to produce the CFP request, nor the procedure employed by the participant to respond to the CFP. Yet, such details are important for developing detailed agent-based system specifications.

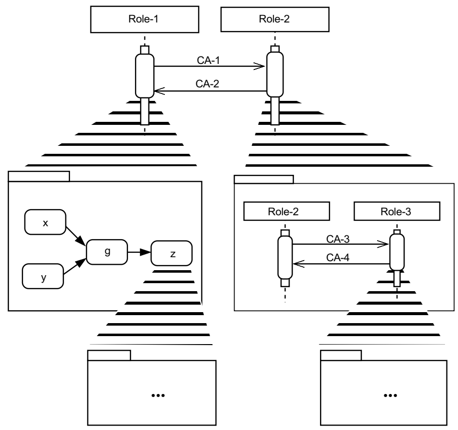



**Figure 2. Interaction protocols can be specified in more detail (i.e., leveled or nested) using a combination of diagrams.**

Figure 2 illustrates how *leveling* can express more detail for any interaction process. For example, the process that generated the communication act CA-1 could be complex enough to specify its processing in more detail using an activity diagram. The agent receiving CA-1 has a process that prepares a response. In this example, the process being specified is depicted using a sequence diagram, though any modeling language could be chosen to further specify an agent's underlying process. In UML, the choice is an interaction diagram, an activity diagram, or a statechart.

Finally, leveling can continue "down" until the problem has been specified adequately to develop or generate code. So in Fig. 2, the interaction protocol at the top of the diagram has a level of detail below, which in turn has another level of detail. Each level can express *intra-agent* or *inter-agent* activity.

In summary, these two examples illustrate several features of our approach:

- The protocol as a whole is an entity. This top level is discussed further in Section 4.
- The sequence diagram itself describes the inter-agent transactions needed to implement the protocol. Section 5 further discusses this notation and an alternative (the collaboration diagram).

In addition to inter-agent transactions, complete specification of a protocol requires discussion of intra-agent activity and is supported by UML's activity diagrams and statecharts (discussed in Section 6).

## 4. LEVEL 1: REPRESENTING THE OVERALL PROTOCOL

Patterns are ideas that have been found useful in one practical context and can probably be useful in others. As such, they give us examples or analogies that we might use as solutions to problems in system analysis and design. Agent interaction protocols, then, provide us with reusable solutions that can be applied to various kinds of message sequencing we encounter between agents. There are two UML techniques that best express protocol solutions for reuse: *packages* and *templates*.

### 4.1. Packages

Since interaction protocols are patterns, they can be treated as reusable aggregates of processing. UML describes two ways of expressing aggregation for OO structure and behavior: *components* and *packages*. Components are physical aggregations that compose classes for implementation purposes. Packages aggregate modeling elements into conceptual wholes. Here, classes can be conceptually grouped for any arbitrary purpose, such as a subsystem grouping of classes. Since AIPs can be viewed in conceptual terms, the package notation of a tabbed folder was employed in Fig. 1.[2]

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, Fig. 3 depicts two packages. The Purchasing package expresses a simple protocol between a Broker and a Retailer. Here, the Broker sends a call for proposal to a Retailer and the Retailer responds with a proposal. For certain products, the Retailer might also place a request with a Wholesaler regarding availability and cost. Based on the return information, the Retailer can provide a more accurate proposal. All of this could have been put into a single Purchasing Protocol package. However, many

[2] In UML 1.3, packages only group class diagrams as a unit. However, we recommend that behavior-related diagrams should also be grouped using the same mechanism—instead of defining yet-another symbol. And more generally, packages should also be used to group any arbitrary model grouping.

businesses or departments may not need the additional protocol involving the Wholesaler. Therefore, two packages can be defined: one for Purchasing and one for Supplying. When a particular scenario requires the wholesaler protocol, it can be nested as a separate and distinct package. However, when a purchasing scenario does not require it, the package is more parsimonious.

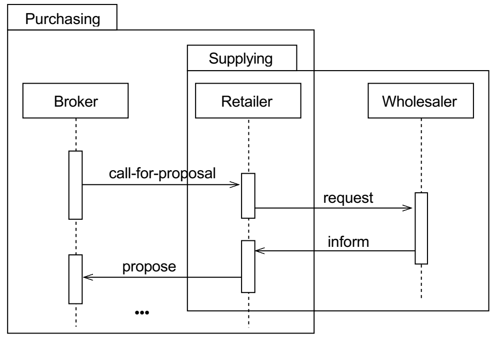



**Figure 3. Using packages to express nested protocols.**

Burmeister et al. suggest a similar construct when they describe their *complex cooperation protocols* [7]. Their three primitive protocols—offering, requesting, and proposing—"are general enough to be used in a large number of interaction situations." Their approach "allows for the construction of (more complex) application or task protocols." In addition to their three primitive protocols, we advocate a pragmatic approach where the analyst may extend Burmeister's general set to include any protocols that might be reused for a nested specification—using AUML.

## 4.2. Templates

Figure 1 illustrates a common kind of behavior that can serve as a solution in analogous problem domains. In Fig. 3, the supplying behavior is reused exactly as defined by the Supplying package. However, to be truly a pattern—instead of just a reusable component—package customization must be supported. For example, Fig. 4 applies the FIPA Contract Net Protocol to a particular scenario involving buyers and sellers. Notice that the Initiator and Participant agents have become Buyer and Seller agents, and the call-for-proposal has become the seller-rfp. Also in this scenario are two forms of refusal by the seller: Refuse-1 and Refuse-2. Lastly, an *actual* deadline has been supplied for a response by the seller.

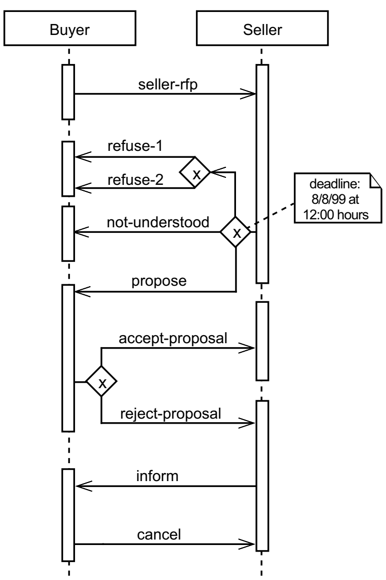



**Figure 4. Applying the template in Fig. 1 to a particular scenario involving buyers and sellers.**

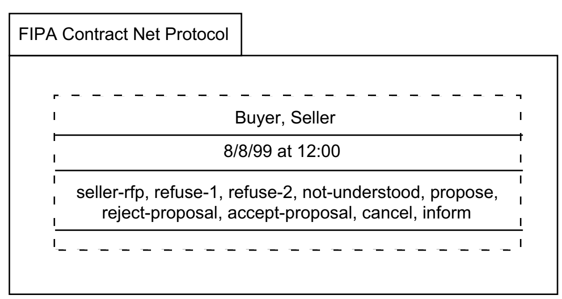



**Figure 5. Producing a new package using the Fig. 1 template; Fig. 4 is the resulting model.**

In UML argot, the AIP package serves as a *template.* A template is a parameterized model element whose parameters are bound at model time (i.e., when the new customized model is produced). In Fig. 1, the dotted box in the upper right indicates that the package is a template. The unbound parameters in the box are divided by horizontal lines into three categories: role parameters, constraints, and communication acts. Figure 5 illustrates how the new package in Fig. 4 is produced using the template definition in Fig. 1.[3] Wooldridge et al. suggest a

[3] This template format is not currently UML compliant but is recommended for future UML extensions.

similar form of definition with their *protocol definitions* [28]. In their packaged templates "a pattern of interaction . . . has been formally defined and abstracted away from any particular sequence of execution steps. Viewing interactions in this way means that attention is focussed on the essential nature and purpose of interaction rather than the precise ordering of particular message exchanges." Instead of the notation illustrated by Wooldridge et al., our graphical approach more closely resembles UML, while expressing the same semantics.

# 5. LEVEL 2: REPRESENTING INTERACTIONS AMONG AGENTS

UML's *dynamic models* are useful for expressing interactions among agents. *Interaction diagrams* capture the structural patterns of interactions among objects. Sequence diagrams are one member of this family; collaboration diagrams are another. The two diagrams contain the same information. The graphical layout of the sequence diagram emphasizes the chronological sequence of communications, while that of the collaboration diagram emphasizes the associations among agents. *Activity diagrams* and *statecharts* capture the flow of processing in the agent community.

## 5.1. Sequence Diagrams

A brief description of sequence diagrams using the example in Fig. 1 appeared above. (For a more detailed discussion of sequence diagrams, see Rumbaugh [24] and Booch [3].) In this section, we discuss some possible extensions to UML that can also model agent-based interaction protocols.

Figure 6 depicts some basic elements for agent communication. The rectangle can express individual agents or sets (i.e., roles or classes) of agents. For example, an individual agent could be labeled Bob/Customer. Here Bob is an instance of agent playing the role of customer. Bob could also play the role of Supplier, Employee, and Pet Owner. To indicate that Bob is a Person—independent of any role he plays—Bob could be expressed as Bob:Person. The basic format for the box label is agent-name/role:class. Therefore, we could express all the various situations for Bob, such as Bob/Customer:Person and Bob/Employee:Person.

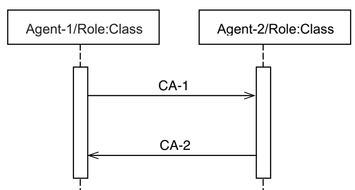
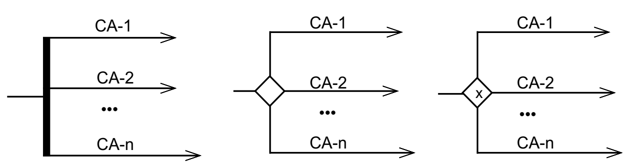



**Figure 6 Basic format for agent communication.**

The rectangular box can also indicate a general set of agents playing a specific role. Here, just the word Customer or Supplier would appear. To specify that the role is to be played by a specific class of agent, the class name would be appended (e.g., Employee:Person, Supplier:Party). In other words, the agent-name/role:class syntax is used without specifying an individual agent-name.

The agent-name/role:class syntax is already part of UML (except that the UML syntax indicates an object name instead of an agent name). Figure 6 extends UML by labeling the arrowed line with an agent *communication act* (CA), instead of an OO-style *message*.




(a) (b) (c)

**Figure 7. Some recommended extensions that support concurrent threads of interaction.**

Another recommended extension to UML supports concurrent threads of interaction. While concurrent threads are not forbidden in OO, they are not commonly employed.[4] Figure 7 depicts three ways of expressing multiple threads. Figure 7(a) indicates that all threads CA-1 to CA-n are sent concurrently. Figure 7(b) includes a decision box indicating that a decision box will decide which CAs (zero or more) will be sent. If more than one CA is sent, the communication is concurrent. In short, it indicates an *inclusive or*. Fig. 7(c) indicates an *exclusive or*, so that exactly one CA will be sent. Figure 7(a) indicates an *and* communication.

Figure 8 illustrates one way of using the concurrent threads of interaction depicted in Fig. 7. Figures 8(a) and (b) portray two ways of expressing concurrent threads sent from agent-1 to agent-2. The multiple vertical, or *activation*, bars indicate that the receiving agent is processing the various communication threads concurrently. Figure 8(a) displays parallel activation bars and Fig. 8(b) activation bars that appear on top of each other. A few things should be noted about these two variations:

- The semantic meaning is equivalent; the choice is based on ease and clarity of visual appearance.
- Each activation bar can indicate either that the agent is using a different role or that it is merely employing a different processing thread to support the communication act. If the agent is using a different role, the activation bar can be annotated appropriately. For example in Figs. 8(a) and (b), CA-n is handled by the agent under its role-1 processing.

[4] As OO implementations become more advanced, such an extension would be considered useful in any case.

- These figures indicate that a single agent is concurrently processing the multiple CAs. However, the concurrent CAs could *each* have been sent to a *different* agent, e.g., CA-1 to agent-2, CA-2 to agent-3, and so on. Such protocol behavior is already supported by UML; the notation in Fig. 7, on the other hand, is a recommended extension to UML.

(For more detailed treatment of these extensions to the UML sequence diagram for protocols, see [2].)

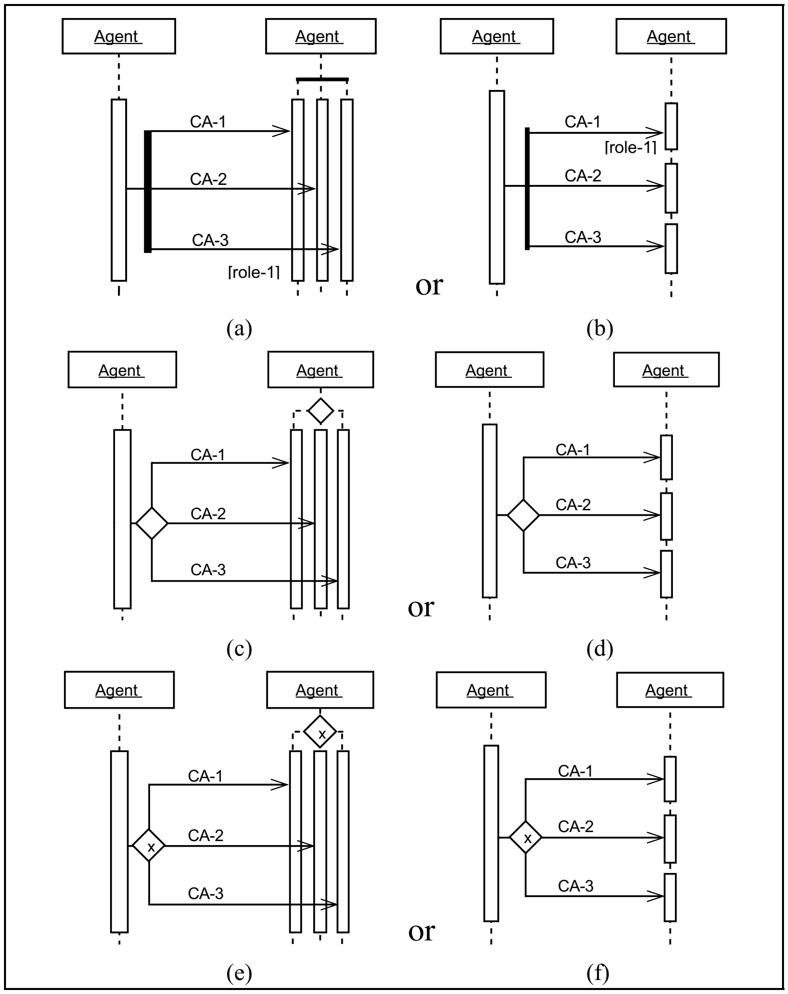



**Figure 8. Multiple techniques to express concurrent communication with an agent playing multiple roles or responding to different CAs.**

## 5.2. Collaboration Diagrams

Figure 9 is an example of a collaboration diagram and depicts a pattern of interaction among agents. One of the primary distinctions of the collaboration diagram is that the agents (the rectangles) can be placed anywhere on the diagram; whereas in a sequence diagram, the agents are placed in a horizontal row at the diagram's top. The sequence of interactions are numbered on the collaboration diagram; whereas the interaction diagram is basically read from the top down. If the two interaction diagrams are so similar, why have both? The answer lies primarily in how clear and understandable the presentation is. Depending on the person and interaction protocol being described, one diagram type might provide a clearer, more understandable representation over another. Semantically, they are equivalent; graphically they are similar. For example, Fig. 10 expresses the same underlying meaning as Fig. 8 using the sequence diagram. Experience has demonstrated that agent-based modelers can find both types of diagrams useful.

Dooley Graphs [21] are isomorphic to collaboration diagrams. The critical distinction is that a single agent can appear as multiple nodes in a Dooley Graph. The ICMAS paper calls these nodes *characters*. The intuition in the terminology is that a character is a specific agent playing a specific role. The role is an abstraction over several characters with similar patterns of interaction. Inversely, each node is an agent in a specific role, where "role" is here defined fairly narrowly (not just purchaser, for example, but purchaser under a renegotiated contract in contrast with the same purchaser's role in the original contract).

Given our notation for an agent playing a role and having a precise enough definition of roles, we could construct a collaboration diagram that has the same semantic content as a Dooley Graph.

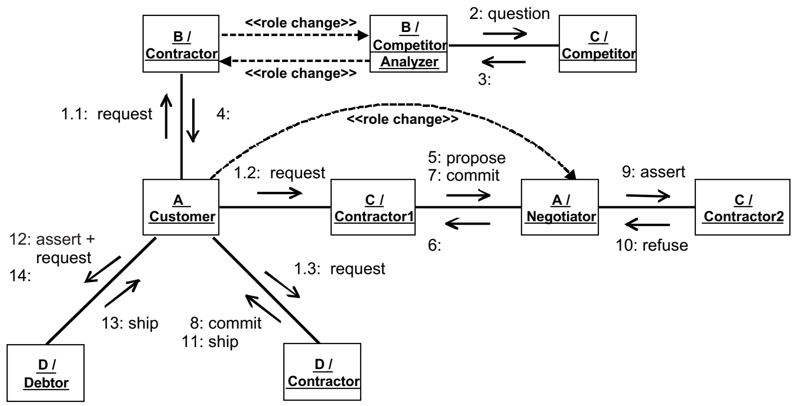



**Figure 9. An example of a collaboration diagram depicting an interaction among agents playing multiple roles.**

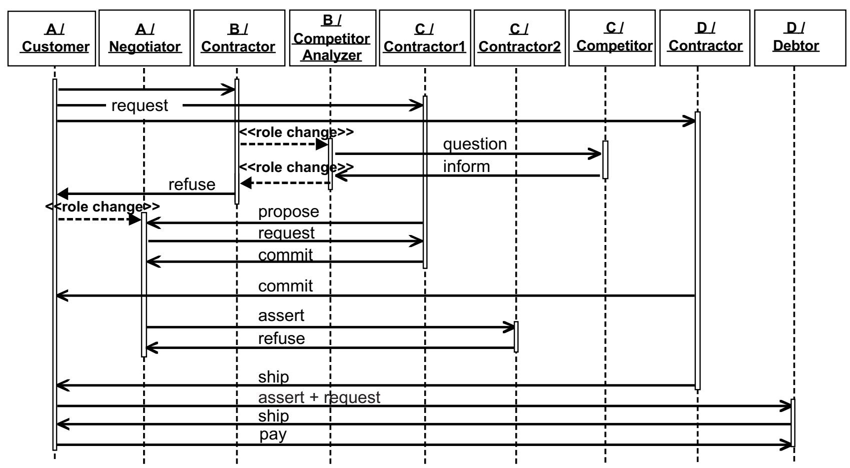



**Figure 10. A sequence diagram version of Fig. 9.**

## 5.3. Activity diagrams

Agent interaction protocols can sometimes require specifications with very clear processing-thread semantics. The *activity diagram* expresses operations and the events that trigger them. (For a more detailed treatment, see Odell's description of activity diagrams in [18].) The example in Fig. 11 depicts an order processing protocol among several agents. Here, a Customer agent places an order. This process results in an Order placed event that triggers the broker to place the order, which is then accepted by an Electronic Commerce Network (ECN) agent. The ECN can only associate an order with a quote when both the order and the market maker's quote has been accepted. Once this occurs, the Market Maker and the Broker are concurrently notified that the trade has been competed. The activity diagram differs from interaction diagrams because it provides an explicit thread of control. This is particularly useful for complex interaction protocols that involve concurrent processing.

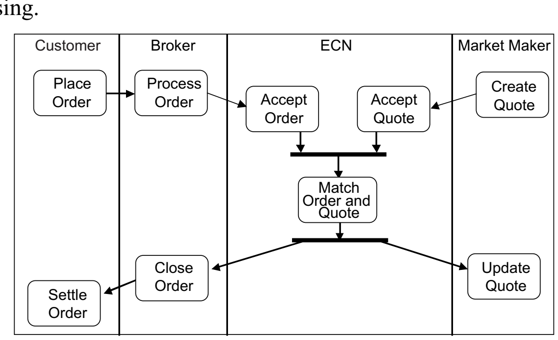



**Figure 11. An activity diagram that depicts a stock sale protocol among several agents.**

Activity diagrams are similar in nature to colored Petri nets in several ways. First, activity diagrams provide a graphical representation that makes it possible to visualize processes simply, thereby facilitating the design and communication of behavioral models. Second, activity diagrams can represent concurrent, asynchronous processing. Lastly, they can express simultaneous communications with several correspondents. The primary difference between the two approaches is that activity diagrams are formally based on the extended state-machine model defined by UML [24]. Ferber's BRIC formalism [8] extends Petri nets for agents-based systems; this response extends UML activity diagrams for the same purpose.

## 5.4. Statecharts

Another process-related UML diagram is the *statechart*. A statechart is a graph that represents a state machine. States are represented as round-cornered rectangles, while transitions are generally rendered by directed arcs that interconnect the states. Figure 12 depicts an example of a statechart that governs an Order protocol. Here, if a given order is in a Requested state, a supplier agent may commit to the requested negotiation—resulting in a transition to a Committed negotiation state. Furthermore, this diagram indicates that an agent's commit action may occur only if the order is in a Requested state. The Requested state has two other possible actions besides the commit: the supplier may refuse and the consumer may back out. Notice that the supplier may refuse with the order in either the Proposed or the Requested states.

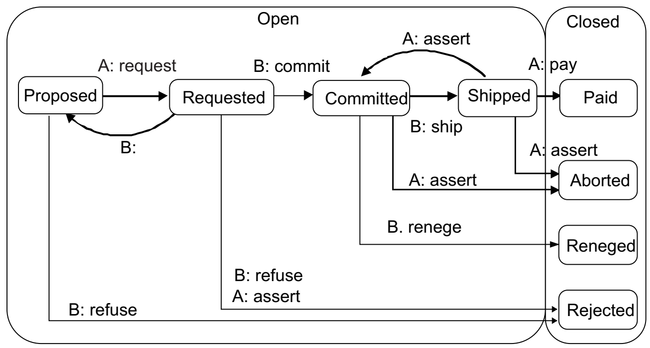



**Figure 12. A statechart indicating the valid states and transitions governing an Order protocol.**

The statechart is not commonly used to express interaction protocol because it is a state-centric view, rather than an agent- or process-centered view. The agent-centric view portrayed by interaction diagrams emphasizes the agent first and the interaction second. The process-centric view emphasizes the process flow (by agent) first and the resulting state change (i.e., event) second. The state-centric view emphasizes the permissible states more prominently than the transition agent processing. The primary strength of the statechart in agent interaction protocols is as a constraint mechanism for the protocol. The statechart and its states are typically not implemented directly as agents. However, an Order agent could embody the state-transition constraints, thereby ensuring that the overall interaction protocol contraints are met. Alternatively, the constraints could be embodied in the supplier and customer roles played by the agents involved in the order process.

# 6. LEVEL 3: REPRESENTING INTERNAL AGENT PROCESSING

At the lowest level, specification of an agent protocol requires spelling out the detailed processing that takes place within an agent in order to implement the protocol. In a holarchic model, higher-level agents (holons) consist of aggregations of lower-level agents. The internal behavior of a holon can thus be described using any of the Level 2 representations recursively. In addition, state charts and activity diagrams can also specify the internal processing of agents that are not aggregates, as illustrated in this section.

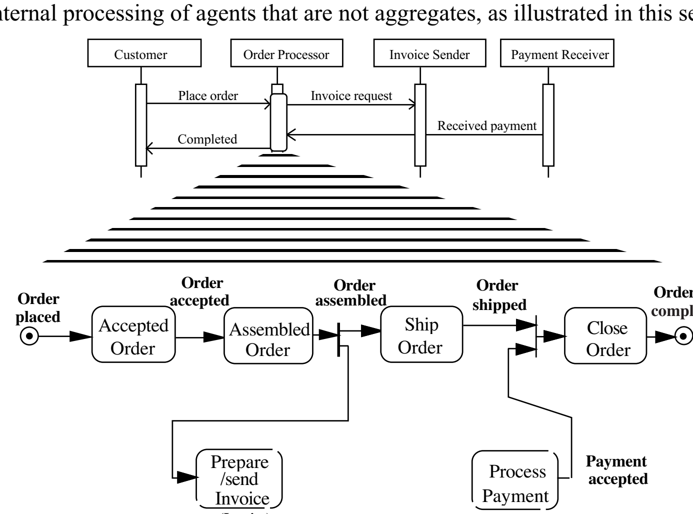



**Figure 13. An activity diagram that specifies order processing behavior for an Order agent.**

## 6.1. Activity Diagrams

Figure 13 depicts the detailed processing that takes place within an Order Processor agent. Here, a sequence diagram indicated that the agent's process is triggered by a Place Order CA and ends with the order completed. The internal processing by the Order Processor is expressed as an activity diagram, where the Order Processor accepts, assembles, ships, and closes the order. The dotted operation boxes represent interfaces to processes carried out by external agents—as also illustrated in the sequence diagram. For example, the diagram indicates that when the order has been assembled, both Assemble Order and Prepare/send Invoice actions are triggered concurrently. Furthermore, when both the payment has been accepted and the order has been shipped, the Close Order process can only then be invoked.

## 6.2. Statecharts

The internal processing of a single agent can also be expressed as statecharts. Figure 14 depicts the internal states and transitions for Order Processor, Invoice Sender, and Payment Receiver agents. As with the activity diagram above, these agents interface with each other—as indicated by the dashed lines. This intra-agent use of UML statecharts supports Singh's notion of agent skeletons [27].

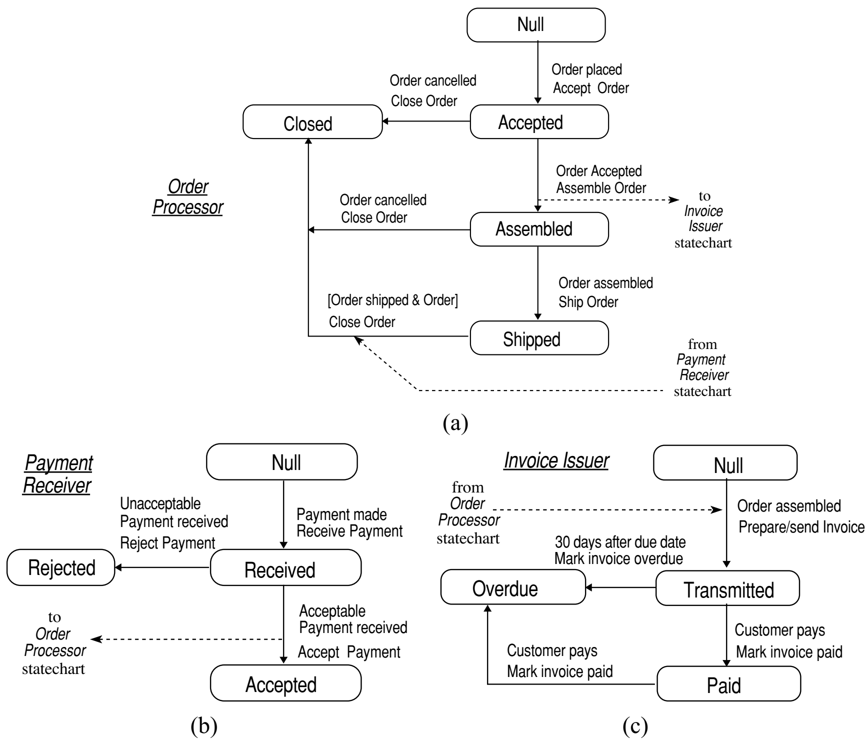



**Figure 14. Statechart that specifies order processing behavior for the three agents.**

# 7. OTHER AUMLCONSIDERATIONS

The previous sections were constrained to examine AUML extensions for agent interaction protocols. This section presents some other agent and agent-based notions that are also recommended for inclusion into UML 2.0.

## 7.1 Richer role specification

Expressing the roles (*ClassifierRole*s) an agent may play in the course of its interaction with other agents is an vital technique for modelers of agent-based systems. UML 1.3 already provides some facility for expressing roles. For example in Fig. 15, agent (object) A is depicted as playing two roles in the interaction protocol: Customer and Negotiator. The slash symbol indicates that the string that follows is a role name.

Figure 15 contains only four agents, each playing just a few roles. Visually, however, it is at the threshold of human readability. Such an approach could quickly become graphically too complex to be comprehensible when even a few more agents are added that play only a few roles. Figures 16 and 17, then, illustrate two new techniques that reduce the visual complexity, yet preserve the same underlying semantics. Figure 16 represents each role with its own *lifeline*; Figure 17 depicts each agent with a single lifeline, where that each *activation* (the tall thin rectangle) is labeled with the appropriate role name. (Note: the usage of the guillemot in Figs. 16 and 17 are not orthodox accoding to UML 1.3; however, no existing technique yet exists to label lifelines with role indication.)

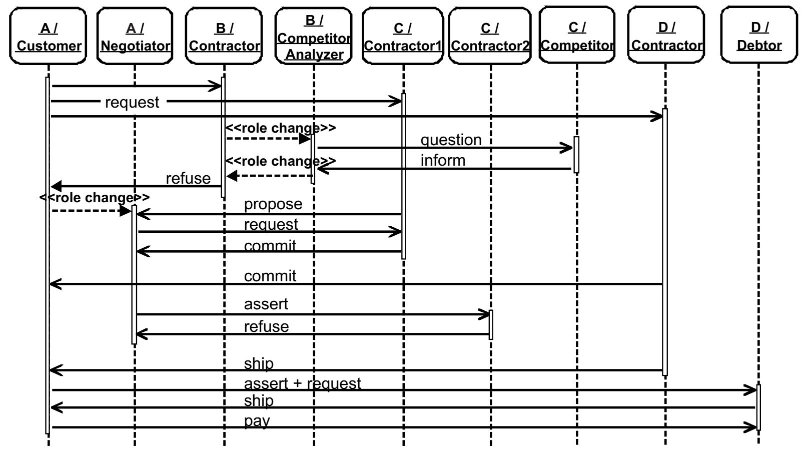



**Figure 15. Role modeling using the existing UML 1.3 "slash" notation.**

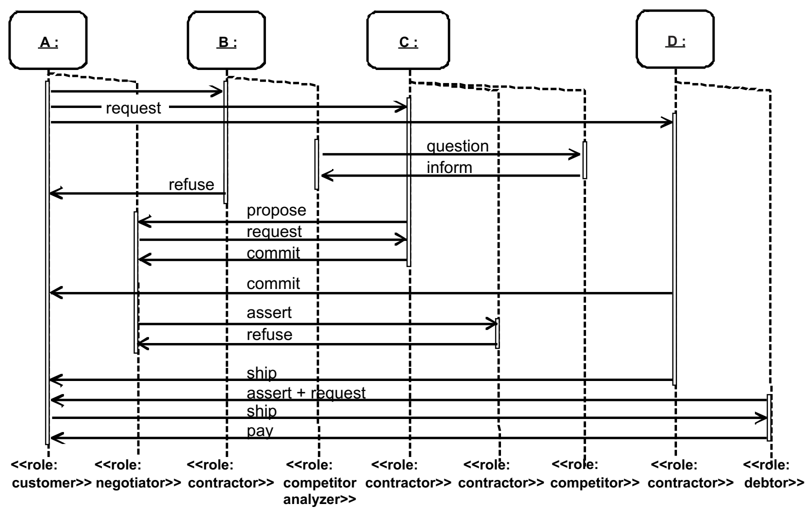



**Figure 16. Role modeling representing each role with its own lifeline.**

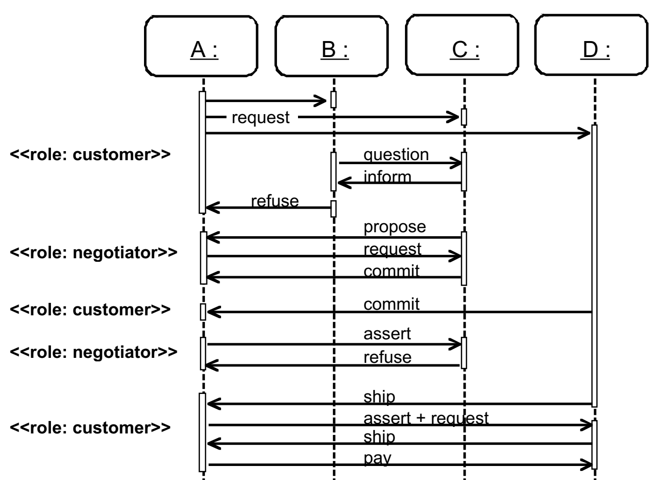



**Figure 17. Role modeling where activation bars are labeled with the appropriate role.**

Most modelers use two approaches to modeling agent roles. A modeler could either start with some conception of roles required (Fig. 18) and the identify the agents involved (Figs. 15 through 17). Or, the modeler could use a diagram like Fig. 18 to summarize the role involvement expressed in Figs. 15 through 17. Either way (bottom up or top down) the technique employed in Fig. 18 is useful—and it is currently supported by UML 1.3; it is included here to ensure an understanding on the overall requirement for roles by agent-based modelers.

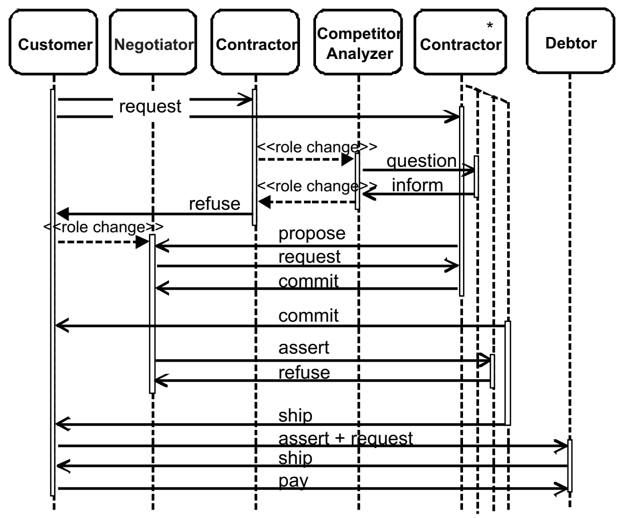



**Figure 18. Role modeling where each "object" node is a class or role.**
**(Role nodes would of course require the slash preceeding the node label.)**

Collaboration diagrams currently have no facility to represent agent roles on interaction lines within UML 1.3. However, each communication act (or message) could be labeled with the role that is responsible for issuing the request. (Note: the usage of the guillemot in Fig. 19 is not orthodox according to UML 1.3; however, no existing technique yet exists to label interaction lines with the sender's role indication.) This technique is illustrated in Fig. 19. For example, communication act 1.1 request is sent by agent A playing the role of Customer.

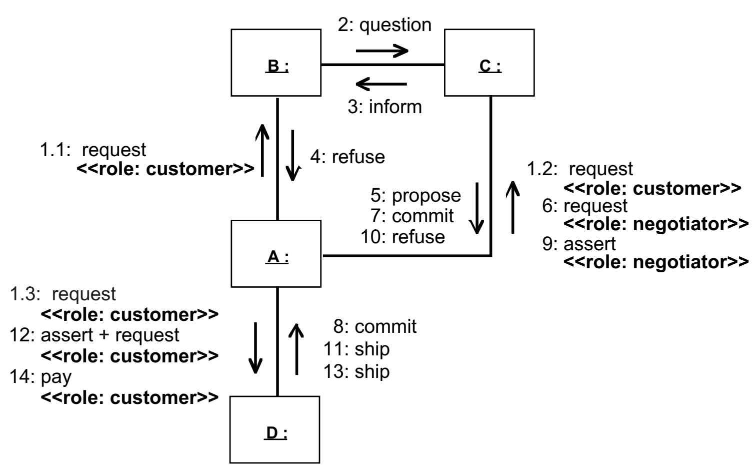



**Figure 19. Role modeling where communication acts (messages) indicate the requesting role.**

Activity diagrams can also be modified to represent agent roles by associating each activity with the name of the appropriate ClassifierRole. For example in Fig. 20, an Order agent accepts, assembles, ships, and closes orders.

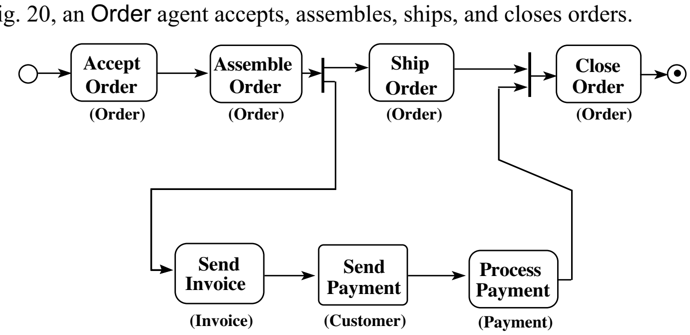



**Figure 20. An activity diagram with activities labeled with the appropriate agent role.**

And lastly, changes in roles can be represented on activity diagrams using notes. For example in Fig. 21, the Hire Employee activity changes the role of an agent from being a Person to also being an Employee. Such a technique also currently supported by UML; it is included here to ensure an understanding on the overall requirement for roles by agent-based modelers.

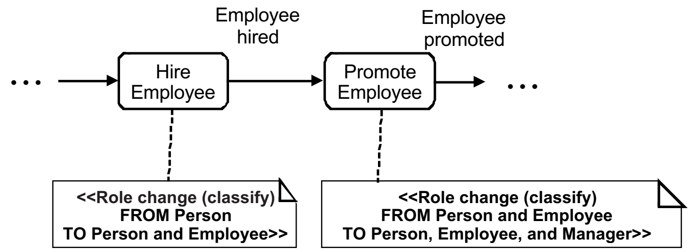



**Figure 21. Dynamic classification (role changes) expressed as notes.**

## 7.2. Package extension

UML 1.3 employees "lollypops" to indicated interfaces. While this will suffice for some agent implementations, it will not handle those situations where the agent—itself—is the interface. For example, a cell membrane receives various "communications" and processes them by rejecting them or accepting them. There is not interface set that a cell publishes to the outside world. A similar analogy is applied in many agent-based implementations. For example in Fig. 22, a Negotiator agent acts as the interface between the outside world and the Manufacturing Cell. This would require a small extension to UML.

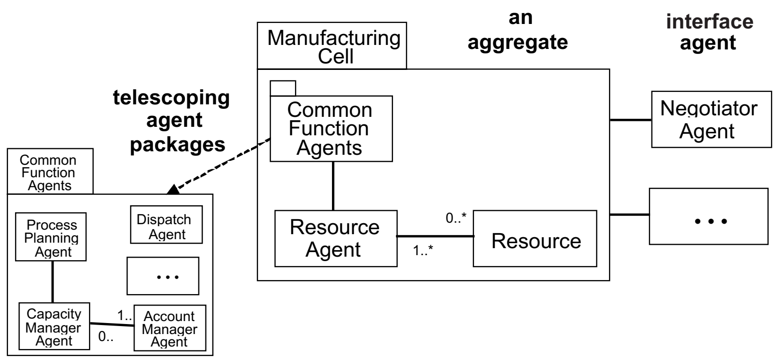



**Figure 22. Package specifying agents instead of operations as interface points.**

## 7.3. Deployment diagram extensions

Mobility is an important agent property—as it is for objects , as well. A way of indicating mobility paths and at-home declarations would be a useful extension to UML.

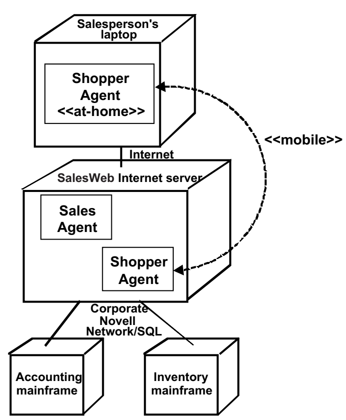



**Figure 23. Adding mobility to deployment diagrams.**

## 7.4. Other useful agent-based notions

Agents borrow many analogies from nature. Agent-based researchers reason that if it took nature 3 billion years to develop what we now see in the modern world, perhaps we could learn some lessons about building our own systems.

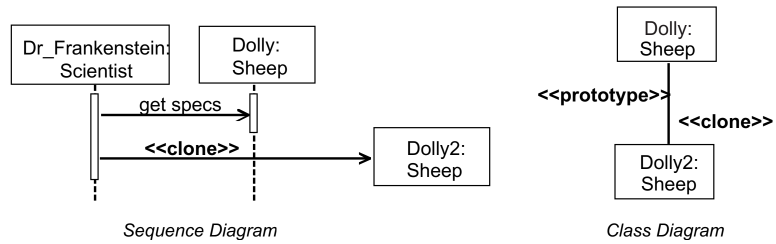



**Figure 24. Representing cloning using sequence and class diagrams.**

For example, agent cloning is a common agent-based technique. Figure 24 illustrates a behavioral and a structural view of agent cloning. Mitosis and reproduction are also common techniques for agent-based societies. Figure 25 illustrates possible ways of expressing these. Extending the UML to include these notion permanently instead of by stereotype is a recommendation as these are common techniques. If OO approaches determine that such notions are not useful, perhaps they belong in a an Agent Profile.

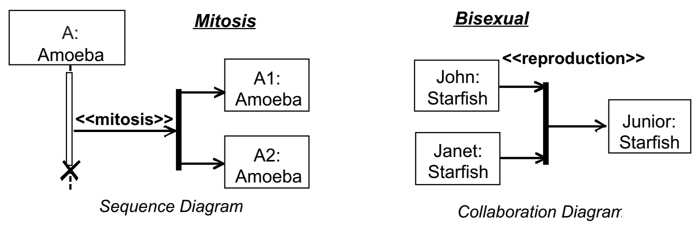



**Figure 25. Representing mitosis and reproduction using sequence and activity diagrams.**

Other notions that are commonly employed for agents are parasitic and symbiotic relationships. Figure 26 illustrates some examples of these.

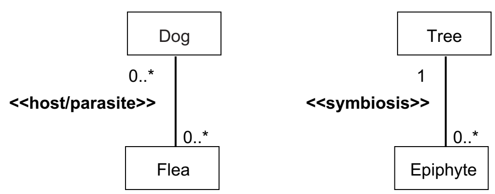



**Figure 26. Representing parasitic and symbiotic relationships using class diagrams.**

The interaction of many individual agents can give rise to secondary effects where groups of agents behave as a single entity, or aggregate agent. This phenomenon is know as *emergence*. While such occurrences are often unplanned and unexpected, they should be anticipated. In multiagent systems, emergence is a core concept. As such, a way of representing it is important.

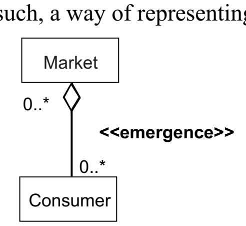



**Figure 27. Representing emergence possibilities using a class diagram.**

## 8. CONCLUSION

UML provides tools for :

- specifying agent interaction protocols as a whole, as in [28];
- expressing the interaction pattern among agents within a protocol, as in [1, 8, 21]; and
- representing the internal behavior of an agent, as in [27].
- representing other agent-related UML extensions that are already commonly used, such as richer role specification, packages with agent interfaces, deployment diagrams indicating mobility, and other techniques.

Some of these tools can be applied directly to agent-based systems by adopting simple idioms and conventions. In other cases, we suggest several straightforward UML extensions that support the additional functionality that agents offer over the current UML version 1.3. Many of these proposed extensions are already being considered by the OO community as useful extensions to OO development on UML version 2.0. Furthermore, many of the AUML notions presented here were developed and applied within the MoTiV-PTA projects [http://www.motiv.de/], an agent-based realization of a personal travel assistant, supported by the German Ministry of Technology.

Agent researchers can be gratified at the increasing attention that industrial and business users are paying to their results. The transfer of these results to practical application will be more rapid and accurate if the research community can communicate its insights in forms consistent with modern industrial software practice. AUML builds on the acknowledged success of UML in supporting industrial-strength software engineering. The idioms and extensions proposed here for AIP's—as well as others that we are developing—are a contribution to this objective.

## 9. REFERENCES


1. Bauer, B., *Extending UML for the Specification of Interaction Protocols*, submitted for the 6th Call for Proposal of FIPA, 1999.
2. Bauer, B., *Extending UML for the Specification of Interaction Protocols*, submitted to ICMAS 2000, 2000.
3. Booch, Grady, James Rumbaugh, and Ivar Jacobson, *The Unified Language User Guide*, Addison-Wesley, Reading, MA, 1999.
4. Brazier, Frances M.T., Catholijn M. Jonkers, and Jan Treur, ed., *Principles of Compositional Multi-Agent System Development* Chapman and Hall, 1998.
5. Bryson, Joanna, and Brendan McGonigle, "Agent Architecture as Object Oriented Design," *Intelligent Agents IV: Agent Theories, Architectures, and Languages. Proceedings of ATAL'97.,* ed., Springer, Berlin, 1998.
6. Burmeister, B., ed., *Models and Methodology for Agent-Oriented Analysis and Design* 1996.
7. Burmeister, Birgit, Afsaneh Haddadi, and Kurt Sundermeyer, ed., *Generic, Configurable, Cooperation Protocols for Multi-Agent Systems* Springer, Neuchâtel, Switzerland, 1993. (Programmable model of interaction)
8. Ferber, Jacques, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison Wesley Longman, Harlow, UK, 1999.
9. Garijo, Francisco J., and Magnus Boman ed., *Multi-Agent System Engineering: Proceedings of MAAMAW'99*, Springer, Berlin, Germany, 1999.
10. Gustavsson, Rune E., "Multi Agent Systems as Open Societies," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* ed., Springer, Berlin, 1998.
11. Herlea, Daniela E., Catholijun M. Jonker, Jan Treur, and Niek J.E. Wijngaards, ed., *Specification of Behavioural Requirements within Compositional Multi-Agent System Design* Springer, Valencia, Spain, 1999.
12. Iglesias, Carlos A., Mercedes Garijo, and José C. González, ed., *A Survey of Agent-Oriented Methodologies* University Pierre et Marie Curie, Paris, FR, 1998.
13. Iglesias, Carlos A., Mercedes Garijo, José C. González, and Juan R. Velasco, "Analysis and Design of Multiagent Systems using MAS-CommonKADS," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 313-328.
14. Jonker, Catholijn M., and Jan Treur, ed., *Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness* Springer, 1997.
15. Kinny, David, and Michael Georgeff, "Modelling and Design of Multi-Agent Systems," Intelligent Agents III: Proceedings of the Third International workshop on Agent Theories, Architectures, and Languages (ATAL'96), ed., Springer, Heidelberg, 1996.
16. Kinny, David, Michael Georgeff, and Anand Rao, "A Methodology and Modelling Technique for Systems of BDI Agents," *Agents Breaking Away. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96).,* Walter VandeVelde and John W. Perram ed., Springer, Berlin, 1996, pp. 56-71. .
17. Lee, Jaeho, and Edmund H. Durfee, "On Explicit Plan Languages for Coordinating Multiagent Plan Execution," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* ed., Springer, Berlin, 1998, pp. 113-126.
18. Martin, James, and James J. Odell, *Object-Oriented Methods: A Foundation*, (UML edition), Prentice Hall, Englewood Cliffs, NJ, 1998.
19. Nodine, Marian H., and Amy Unruh, "Facilitating Open Communication in Agent Systems: the InfoSleuth Infrastructure," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 281-296.
20. Odell, James ed., *Agent Technology*, OMG, green paper produced by the OMG Agent Working Group, 2000.
21. Parunak, H. Van Dyke, ed., *Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis* 1996.
22. Parunak, H. Van Dyke, and James Odell, *Engineering Artifacts for Multi-Agent Systems*, ERIM CEC, 1999.
23. Parunak, H. Van Dyke, John Sauter, and Steven J. Clark, "Toward the Specification and Design of Industrial Synthetic Ecosystems," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 45-59.
24. Rumbaugh, James, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.
25. Schoppers, Marcel, and Daniel Shapiro, "Designing Embedded Agents to Optimize End-User Objectives," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 3-14.
26. Singh, Munindar P., "A Customizable Coordination Service for Autonomous Agents," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 93-106.
27. Singh, Munindar P., ed., *Developing Formal Specifications to Coordinate Heterogeneous Autonomous Agents* IEEE Computer Society, Paris, FR, 1998.
28. Wooldridge, Michael, Nicholas R. Jennings, and David Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," International Journal of Autonomous Agents and Multi-Agent Systems*,* 3:Forthcoming, 2000.