Agent UML Class Diagrams Revisited

Marc-Philippe Huget

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

Abstract. Multiagent system designers already use Agent UML in order to represent the interaction protocols [8] [2]. Agent UML is a graphical modeling language based on UML. As UML, Agent UML provides several types of representation covering the description of the system, of the components, the dynamics of the system and the deployment. Since agents and objects are not exactly the same, one can guess that the UML class diagram has to be changed for describing agents. The aim of this paper is to present how to extend Agent UML class diagrams in order to represent agents. We then compare our approach to Bauer's approach [1].

This paper is published as the technical report ULCS-02-013 from the department of computer science, University of Liverpool. Version 2002/04/16

1 Introduction

Since several years, designers have a modeling language so as to design multiagent systems. This modeling language is called Agent UML [8] [2] and is derived from the modeling language UML which is well-known in software engineering. As Odell and Bauer quoted it (ibid.), it is important that designers have a modeling language which is not far from their usual modeling language. Actually, most of designers have first designed software or they are aware of UML. Moreover, it is not possible to use directly UML since objects and agents are not the same [7]. Particularly, agents encompass the notion of autonomy and the ability to cooperate.

The modeling language Agent UML is more and more an appealing approach if we consider the number of papers written on it (see Agent UML web site[1]). Designing multiagent systems with Agent UML needs to represent elements through diagrams and representations. In UML, objects which are the main element in the software are represented by class diagrams. Since objects and agents are not exactly the same, one guesses these class diagrams have to be modified in order to consider special agent characteristics. The aim of this paper is to present our

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

approach for the design of agents by class diagrams. This name *class diagram* is now not really correct how much we modify this diagram. It might be better to call it *agent diagram* like Bauer proposed it [1]. Actually, an object is represented by attributes and operations where agents encompass behaviors, knowledge and plans as well.

The remainder of this paper is as follows. Section 2 presents our approach to class diagrams. Another work exists in the domain of Agent UML class diagram. It is the work from Bauer [1]. Section 3 gives a comparison between our approach and Bauer's approach. Section 4 concludes the paper and gives some information about future work. We do not give a long example in this paper but several examples in the section 2, especially in the domain of supply chain management.

2 Our Proposal of Agent UML Class Diagrams

Class diagrams in UML are used in order to represent the relationships between classes and to define attributes and operations for all the classes. It is possible, in UML, to consider different levels of abstraction for the description of class diagrams. Actually, at some levels, it is not interesting to have an accurate view of all classes but just the name of classes. These different levels of abstraction help designers to tackle the complexity of the system. For instance, in software engineering, a special view can only consider the connection to a data server. UML class diagrams are described in [3]. We do not go further in this presentation since most of the elements are explained below when we present our approach.

Unlike systems described through class diagrams in UML, Agent UML class diagrams must contain both agents and classes. Actually, agents could be defined with classes or could use classes. For instance, when one considers ants and pheromones, ants are defined as agents and pheromones as objects. Thus, we have to distinguish agents and classes on diagrams. We propose to write first `«agent»` then the name of the agent `ant` as shown on Figure 4.

In order to ease the reading of our proposal, we explain the different elements used in Agent UML class diagrams in Section 2.1. Then, we describe the two different levels of abstraction that we consider. In Section 2.2, the conceptual view of the system. It is a really abstract view of the system where only agents' name, classes' name and relationships are defined. In Section 2.3, we deal with an accurate view of the system where agents are defined in depth.

2.1 Class Diagram Elements

Two elements are considered in this section: classes and relationships between these classes.

Classes A class is represented by a rectangular icon which is divided in compartments. In UML, there exists usually three compartments: (1) the class name, (2) the class attributes and (3) the class operations (see Figure 1).

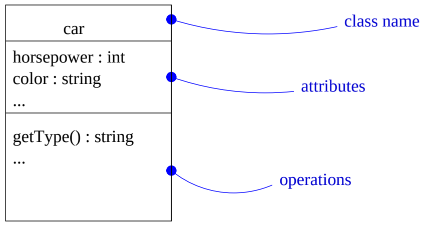

Fig. 1. Class Representation in UML Class Diagrams

The class name can be prefixed by a path if this class belongs to a package[2]. There exist three methods for representing attributes: (1) the attribute name, (2) the name and the type of the attribute and (3) the name and the type of the attribute and its initial value. The schema for building attributes is: `name : type = value`. The description of operations is as follows: `name(parameters) : type` where parameters contain the name and the type of the parameters and, if needed, its initial value. The operation type is only used if this operation is a function. In this case, the type corresponds to the type of the returned value.

We let readers read [3] for further details about access modifiers (private, protected, public), adornments and the possible other compartments like responsibilities and exceptions.

Relationships There exist four different relationships in UML: association, generalization, aggregation and dependency.

The *association* is used for linking different classes. Usually, an association is a binary link between two classes. With association links, it is possible to go from one class to another one and vice-versa. For instance, if one has an association between one class *company* and one class *employee*, it is possible to know the company where one considers one employee and it is possible to know all the employees if one considers the company. An association is rendered as a solid line connecting the same or different classes (see Figure 2). It is also possible to name this association as well as the multiplicity of the association: how many objects are considered in this association.

The *generalization* allows to represent a connection between a general class and a more specific class. The general class is called the *parent* and the more specific class, the *child*. The *child* inherits the properties of its parents: attributes and operations. Graphically, a generalization is rendered as a solid directed line with a large open arrowhead pointing to its parent as shown in Figure 2.

The *aggregation* defines a class as a composition of smaller parts. It is for instance, the case for the class *car* which is composed of wheels, bodywork, doors, etc. Graphically, an aggregation is rendered a solid directed line with an open

[2] A package is a mechanism for organizing elements into groups.

diamond as shown in Figure 2. The diamond points to the class which merges parts together.

The *dependency* represents the relationship between an independent class and a dependent class. As soon as the independent class is modified, the dependent class has to be modified. It is often the case when the independent class is used in signatures or in the code of the dependent class. Graphically, a dependency is rendered as a dashed directed line, directed to the class being depended on as shown in Figure 2.

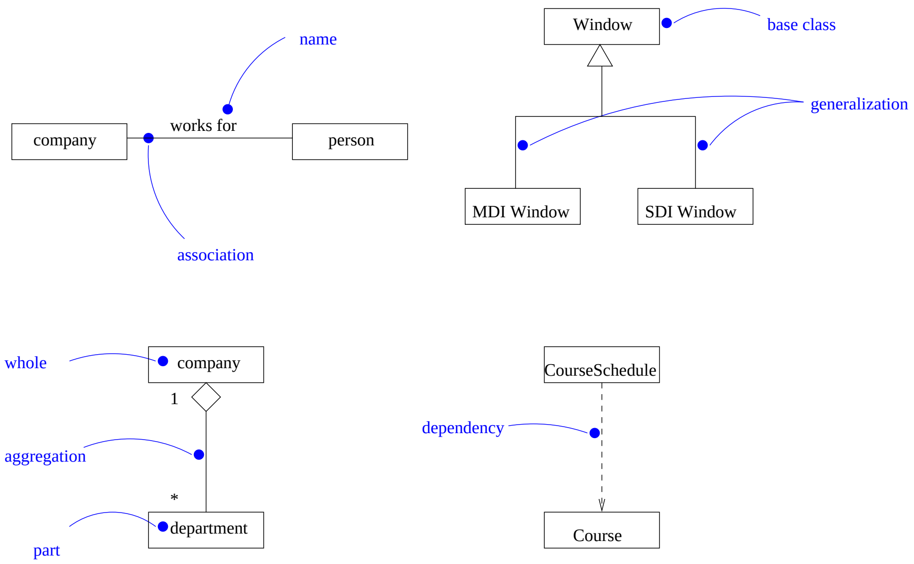

Fig. 2. Relationships in UML

2.2 Conceptual Level

The conceptual level allows designers to have an overall view of the classes in UML. It is the same principle in Agent UML except that we consider here both agents and classes. Since we have two different kind of elements in our class diagrams, we need to rethink the relationships.

The allowed relationships between agents are association and generalization. The association means that several agents can exchange messages between them. The generalization has the same meaning as for classes in UML. We do not use the dependency between two agents since the meaning is different between two objects and between two agents.

If the relationships are used on objects, they have the same meaning that the one defined in UML.

The dependency is the only one relationship which is allowed between agents and classes. In this case, the dependency means that agents use this class.

The table 1 give the allowed relationships between agents and objects.

	object	agent
object	association aggregation generalization dependency	dependency
agent	dependency	association generalization

Table 1. Agents-Objects Relationships

Such an example of conceptual level is given in Figure 4. This is the example of supply chain management [10] where it is represented clients, the company and the providers (see Figure 3). Briefly, we consider nine types of agents: one type for the client, one type for the provider and seven types for the company. The agent *Order Acquisition* who receives the orders and negotiates the delays and the prices. Orders are negotiated with the agent *Logistics*. As soon as orders are accepted, these orders are sent to the agent *Logistics*. It asks the agent *Scheduler* to generate a plan for this order in order to allocate materials, people and transportation. As soon as the plan is available, it is sent to the agent *Transporter*, *Dispatcher* and *Resource*. The agent *Transporter* is responsible for the transportation of the materials and the order to the client. The agent *Dispatcher* is responsible for finding agents *Worker*s who have to do the allocated tasks. The agent *Resource* has to provided materials when needed. If the materials are not available, it has to order these materials to providers.

This example points out that no attributes nor operations are defined in this diagram. It allows designers not to be drowned by numerous information which are not important at this level of abstraction. The figure 4 gives one package called *Company* containing all the agents used in the company. It is interesting to merge together agents working in the same entity. A package is mechanism in UML which organizes objects into groups [3].

It is clear in this example that several levels of abstraction are possible. Actually, in the Figure 4, no classes are given even if these classes are used. One of the great advantage of UML class diagrams is that designers are not required to represent all the information. They can decide what kind of data is important on such a diagram.

Generalization is not considered in this example. However, it would be used if we consider different kind of orders. In this case, we have one agent broker who receives the orders and dispatches them to the corresponding *Order Acquisition* agent. These agents have the same behavior but different properties according to the different lines of products of this company.

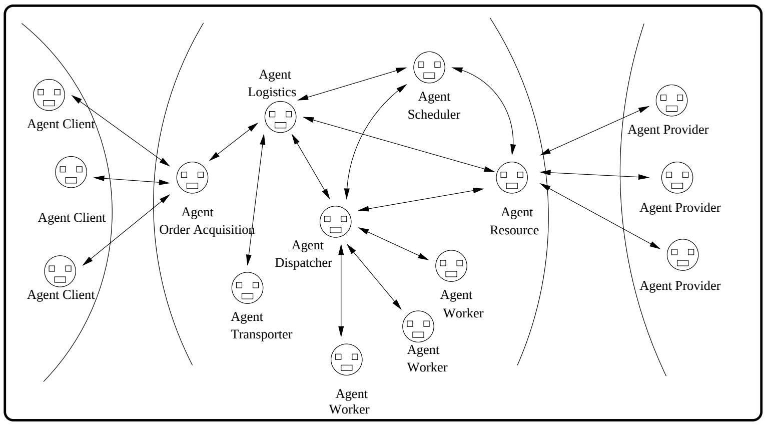

Fig. 3. A Multiagent Approach of the Supply Chain Management

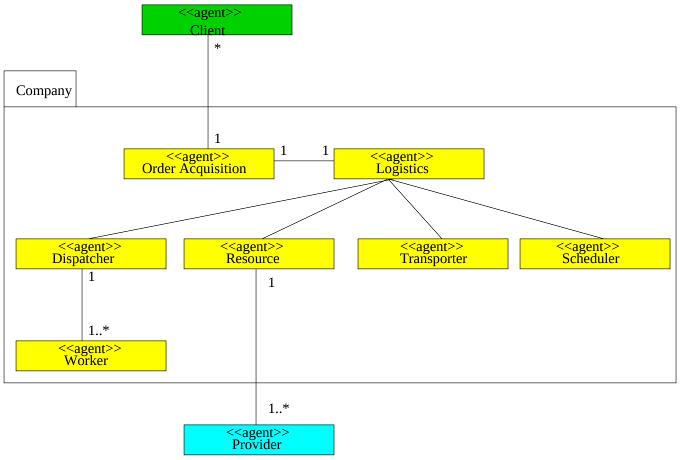

Fig. 4. Conceptual Level of the Supply Chain Management

2.3 Implementation Level

The conceptual level defined in the previous section might seem to be superficial. However, it is useful to have a general view of the system. As soon as designers want to implement their systems, they need to have a detailed view of their systems. The implementation level is given for this purpose. Obviously, designers can define as much diagrams as they want in order to point out different views of the system.

This level is the most difficult. Actually, it is needed to give all the elements which define the agents and the classes. The description of agents might be really difficult if the application domain is complex. In order to help designers, we propose to follow the Vowels approach [4] [5] for the definition of agents. The Vowels model considers four elements in multiagent systems: agent, environment, interaction and organization. These four elements represent the four different views that designers have on agents: (1) an *agency* view where it is given its knowledge, its beliefs, its intentions, its plans and its behaviors, (2) an *environmental* view where it is given how agents react when the environment is modified, (3) an *interactional* view where it is given how agents interact with other agents and (4) an *organizational* view where it is given the organizations in which the agents evolve and what is their role in these organizations.

The Figure 5 summarizes all the information that we define below. We use new compartments since it is possible to store information in new compartments in UML.

Agency View Agents in Agent UML class diagrams are represented by their role in the multiagent system. It is possible that agents have several roles. For instance, it is the case in auctions where agents can be both seller and buyer. Roles are used when designers describe interaction protocols or organizations. The notion of roles reminds us the notion of services provided by agents. Services could be multiagent system administration, directory service, yellow page service, etc. Graphically, a service is rendered as a solid line ended by a circle. The name of the service is given close to the circle as shown in Figure 5.

The execution of agents is done by behaviors which give what actions have to be performed for the current situation. We propose to use statecharts [6] [3] so as to represent the behavioral part of the agents. Since statecharts are another representation, we do not consider statecharts within class diagrams but we just give the name of the statecharts which are defined elsewhere. We will see further that these statecharts are also used for representing the actions that have to be performed when the environment changes. Statecharts have to be seen as finite state machines. Jumping from one state to another one corresponds to an action which occurs either in this agent or in the environment.

Obviously, we need to insert attributes and operations as well. However, the compartment attributes is richer than the one defined in UML class diagrams. Actually, we insert the knowledge of the agent and especially, the BDI (Belief-Desire-Intention) [11] knowledge. Designers will store information as *believe self (b == 10)* denoting that this agent believes that the value of b is 10. Designers

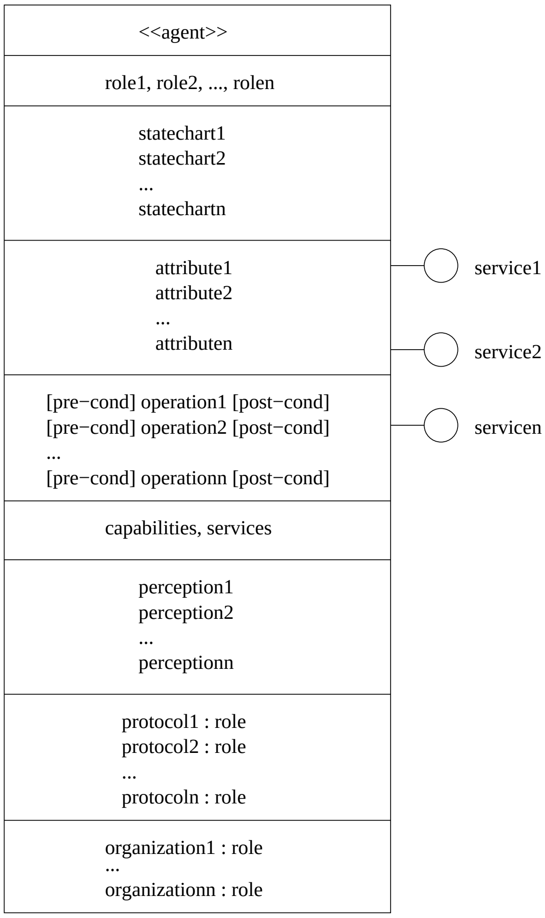

Fig. 5. Agent Class Diagram

could store intentions and plans as well. All knowledge (save BDI knowledge) are adorned with the letter K in order to distinguish classic attributes to knowledge. In this case, knowledge is flat knowledge, that is to say, we do not consider hierarchies of knowledge and composite of knowledge. If designers want to do that, they have to create one attribute which is an instance of one class representing the knowledge. Intentions and plans cannot be flattened so, it is needed to create one attribute which is an instance of one class. A plan would be represented as a tree.

The compartment of operations does not only contain the operations but also some rules. These rules are used to constrain the execution of agents in order to avoid some unexpected situations. One application of rules is for knowledge consistency in order to avoid that two conflicting information are stored in the knowledge base. Operations are adorned with pre-conditions and post-conditions. If pre-conditions hold, agent can fire this operation and post-conditions must hold at the end of the execution of this operation. These conditions can be written informally as text or formally with OCL [9].

Environmental View The environmental view deals with what agents have to do according to the modifications of the environment. In order to represent the modifications and actions, we use statecharts [6]. These statecharts allow designers to represent the agent states and actions needed for going from one state to another one. Statechart diagram is the solution that we want to follow even if statecharts are not the only possibility. We can also use activity diagrams and state machines.

Here follows a small example of an agent monitoring the temperature room. The resulting statechart is given in Figure 6.

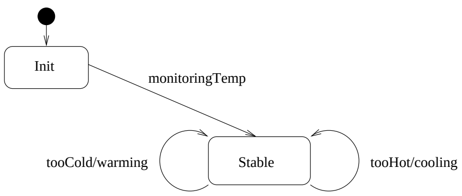

Fig. 6. Statechart Diagram for Agent Monitoring Room Temperature

Statecharts are not defined within class diagrams but outside, only the name of statecharts is given.

Interactional View The interactional view gives all the interaction protocols used by this agent. Interaction protocols are represented by protocol diagrams. Protocol diagram is a proposal of Odell and Bauer [8] [2] and extends UML sequence diagrams. Protocol diagrams allow designers to represent agents by their role in this interaction and messages sent between roles. The example of the English auction protocol is given in Figure 7.

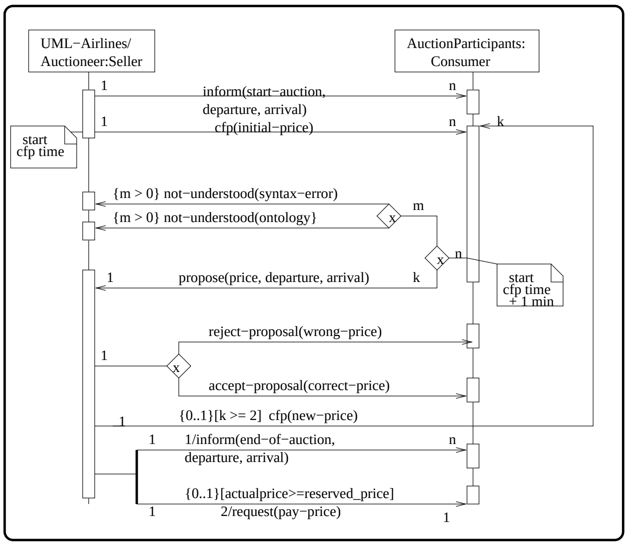

Fig. 7. English Auction Protocol in Agent UML

Protocol diagrams are not defined within class diagrams but outside, only the name of the protocol diagrams are given in class diagrams.

Organizational View Two data are stored in the organizational view. Firstly, designers can insert the different organizations that an agent belongs to in the compartment organization. The roles played by agents in this organization are also given. Secondly, an agent is represented by its services to the system. These services are also stored in this compartment.

3 Comparison with other Approaches

Only one work (except ours) exists in the domain of Agent UML class diagrams [1]. In this section, we compare our approach to Bauer's approach.

Several information are considered in Bauer's approach:

1. Agent class description and roles
2. State description
3. Actions
4. Methods
5. Capabilities

6. Sending and receiving of communicative acts
7. Matching of communicative acts

Agent class description and roles merges two information that we have in our proposal: the agent class name and the roles played by this agent.

State description corresponds to our field attribute. However, we add knowledge in this compartment.

Actions corresponds to two kind of actions in Bauer's approach. We have actions that the agent fires itself and we have actions of this agent that another agents can fire. The first kind of actions reminds us the statecharts in order to represent the behavior of the agents. The compartment *perception* is also considered by this kind of actions. The second kind of actions has to be linked to our compartments *statechart*, *perception* and *protocol*. We cannot go further in this discussion about actions since Bauer is not clear about this kind of actions. Normally, it is impossible to fire actions in another agents since it is opposite to the notion of autonomy in agency.

Methods corresponds to our compartment *operation*.

Bauer does not explain sufficiently what the fields *Capabilities* means. It seems to be that it corresponds to our capabilities, services and interfaces that we propose.

The main difference between our approach and Bauer's approach concerns how the interaction is managed. Actually, Bauer proposes to render received and sent messages on the diagram. This approach seems to be unrealistic as soon as interaction protocols begin complex and several messages are considered. Indeed, in this kind of diagrams, designers have to represent incoming messages, resulting actions and outgoing messages. In our approach, we propose to only insert the name of the protocol diagrams within the class diagram.

For summing up

Several points of resemblance exist between Bauer's approach and ours since we consider attributes, knowledge, operations, roles or capabilities. The main difference is on the management of interaction. Bauer proposes to render the protocols on the class diagrams. This approach seems to be really complex when the number of protocols increase and if the number of messages is important. In our opinion, Bauer does not enough link his work to UML representations. For instance, we insert a link to statecharts and protocol diagrams.

4 Conclusion

Following the example of software engineering where designers have modeling languages for describing elements used in their software, it is important that multiagent system designers have modeling languages too. The modeling language Agent UML [8] seems to be an answer to this need. This modeling language extends UML in order to consider special features encompassed in agents such as autonomy, cooperation and richer interactions.

Since objects and agents are not the same, some UML representations have to be modified. The aim of this paper is to rethink the class diagrams which allow designers to define the structure of the system and the elements. In our proposal, class diagrams contain agents as well as objects. Several compartments are inserted in these diagrams in order to represent agent features such as behaviors, capabilities, services or protocols.

Bauer has also considered how to augment class diagrams in order to represent agents [1]. His approach presents several similarities with ours but interaction is managed according to a different approach. Bauer proposes to render incoming messages, resulting actions and outgoing messages on class diagrams. This approach tends to reduce the readability of the class diagrams if designers have to represent numerous messages and actions.

Several different ways are possible for future work: (1) until now, designers use UML tools in order to represent their Agent UML diagrams. Since we have modified deeply the class diagrams, we have to design specific tools for Agent UML, (2) the design of class diagrams is really difficult since designers have to consider a really accurate view of the agents and multiagent systems. It seems to be useful to propose a methodology or recipes in order to help designers in managing this task. Finally, (3) it might be interesting to consider how to generate code for class diagrams giving the skeleton of a multiagent systems.

References

1. B. Bauer. UML class diagrams revisited in the context of agent-based systems. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Proceedings of Agent-Oriented Software Engineering (AOSE 01)*, number 2222 in LNCS, pages 1–8, Montreal, Canada, May 2001. Springer-Verlag.
2. B. Bauer, J. P. Muller, and J. Odell. An extension of UML by protocols for multiagent interaction. In *International Conference on MultiAgent Systems (ICMAS'00)*, pages 207–214, Boston, Massachussetts, july, 10-12 2000.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.
4. Y. Demazeau. Steps towards multi-agent oriented programming. slides Workshop, 1st International Workshop on Multi-Agent Systems, IWMAS '97, October 1997.
5. Y. Demazeau. *VOYELLES*. Habilitation à diriger les recherches, Institut National Polytechnique de Grenoble, Grenoble, avril 2001.
6. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
7. N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In J. Bradshaw, editor, *Handbook in Agent Technology*. MIT Press, 2000.
8. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, Texas, july, 30 2000. ICue Publishing.
9. OMG. UML 1.4. Technical report, OMG, 2001.
10. J. Swaminathan, S. Smith, and N. Sadeh-Koniecpol. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences*, April 1997.
11. M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.