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. Agent UML is a graphical modeling language based on UML. Like UML, Agent UML provides several types of representation covering the description of the system, the components, the dynamics of the system and the deployment. Multiagent system designers already use Agent UML to represent interaction protocols [12] [2]. Since agents and objects differ on several points, UML class diagram has to be modified for describing agents. The aim of this paper is to present how to extend UML class diagrams in order to represent agents. We then compare our approach to Bauer's approach [1].

1 Introduction

Multiagent system designers like object-oriented system designers need methodologies and tools to design their systems. Several methodologies are available for multiagent system design (see [9] for a helpful survey). We are particularly interested in the graphical modeling language Agent UML [12] [2]. This language is used for the analysis and design of multiagent systems and is an extension of the well known UML [3]. As Odell and Bauer noted it [12] [2], it is important to make profit of previous work (here UML) since it steeps the learning curve for software designers and several industrial-strength tools are available for UML. UML is not directly used since agents differ from objects. They present some features that are not available in objects such as autonomy, enriched communication or rationality [11] [15]. When agents match objects, multiagent system designers use UML diagrams and when it is not the case, specific Agent UML diagrams are provided. Sequence diagrams and class diagrams are two examples of specific diagrams tailored to agents.

A first extension of Agent UML class diagrams was proposed by Bauer in [1]. This paper presents a prolongation of this work.

The remainder of this paper is as follows. Section 2 presents our proposal of class diagrams. Section 3 gives a comparison between our approach and Bauer's approach [1]. Section 4 concludes the paper and gives some information about future work. We do not give a long example but several small examples are in Section 2.

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 these ones. 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 particular view can only consider the connection to a data server. Description of UML class diagrams is in [3]. Most of these elements are present in our proposal.

Agent UML class diagrams may contain both agents and classes. Actually, agents could be defined as a set of classes or could use classes. For instance, when one considers ants and pheromones, ants are defined as agents and pheromones as instances of classes. Thus, we have to distinguish agents from classes on diagrams. We propose to prepend the agent name by the stereotype `«agent»` as shown on Figure 3.

UML considers several levels of abstraction: from low level of abstraction to high level of abstraction [3]. This range of levels of abstraction allows designers to have different views of the system. They are particularly interesting when the system is complex. Two levels of abstraction are usually in use in UML: conceptual level and implementation level. The conceptual level corresponds to high level of abstraction. The implementation level corresponds to low level of abstraction. Section 2.1 presents the conceptual view of the system. Section 2.2 describes the implementation level.

2.1 Conceptual Level

The conceptual level allows designers to have a bird's eye view of both the agents and the classes. The conceptual level is especially used to represent the different agents and classes used in the system and the relationships between these elements. Since agents and classes are mixed on the same diagrams, we have to consider what the four relationships in UML (association, generalization, aggregation and dependency) mean when they are applied between agents or between an agent and objects.

In UML, an *association* is a structural relationship, specifying that objects of one thing are connected to objects of another. Given an association connecting two classes, it is possible to navigate from an object of one class to an object of the other class, and vice-versa. An association that connects exactly two classes is called a binary association. For instance, if an association is defined between one class *company* and one class *employee*, it is then possible to know the company while one employee is considered and it is possible to know the employees while the company is selected. An association is rendered as a solid line connecting the same or different classes (see Figure 1). There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association and aggregation.

In UML, a *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called "is-a-kind-of" relationship: a thing is-a-kind-of a more general thing. 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 on Figure 1.

In UML, an *aggregation* is a plain association between two classes that represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. An aggregation is a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "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 diamond as shown in Figure 1. The diamond points to the class which merges parts together.

In UML, a *dependency* is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the class being depended on as shown in Figure 1.

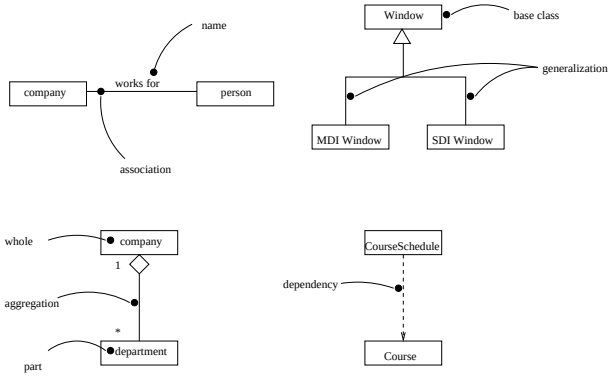

Fig. 1. Relationships in UML

The relationships as defined above correspond to the relationships between classes. Some modifications have to be done when these relationships deal with agents and classes.

The relationships have the following meaning while they are used between agents:

association: the *association* means that there is a relation of acquaintance between the linked agents. An acquaintance corresponds to a relationship

where two agents know each other. This relationship is used when agents are not in a context of cooperation or coordination. As a consequence, agents may exchange messages.

generalization: the *generalization* has the same meaning as for UML classes. The definition of an agent can be derived from another one.

aggregation: the *aggregation* seems to be only possible in one situation: recursion [6]. For instance, if the agent has a recursive architecture, i.e. an agent is an aggregation of several agents.

dependency: the *dependency* relationship in UML corresponds to a unilateral dependency in multiagent theory. We propose to extend this relationship to mutual dependency. If A has a dependency over B, B has also a dependency over A. It is the case if A needs a result that B can provide to it but B needs information that A has.

A new relationship between agents has to be considered: the relationship of *order*. This relationship is particularly used in hierarchies to exhibit that an agent higher in the hierarchy commands an agent lower in the hierarchy to do something.

The relationships have the following meaning while they are used between agents and classes:

association: the *association* relationship connects an agents and several classes. It means that this agent use the connected classes for their execution. It might be a specific class for messages, for goals, for plans. We see further the example of beliefs which are defined as an associated class to agents. This relationship is unidirectional from agents to classes.

generalization: since agents and objects present many differences, it is then impossible that the agents are derived from objects.

aggregation: the *aggregation* between agents and classes implies that agents are defined as an aggregation of several classes. It is frequently the case when designing agent architectures: an agent architecture contains a reasoning part, an interaction part and a perception part such as InteRRaP [10]. All these parts are implemented as objects.

dependency: the *dependency* between agents and classes is possible and means that one agent needs this class either in its code or during its execution since an instance of this class corresponds to an object that the agent can use.

Such an example of conceptual level is given in Figure 3. This is the example of supply chain management [13] where it is represented clients, the company and providers as shown on Figure 2.

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 to allocate materials, workers and transportation. As soon as the

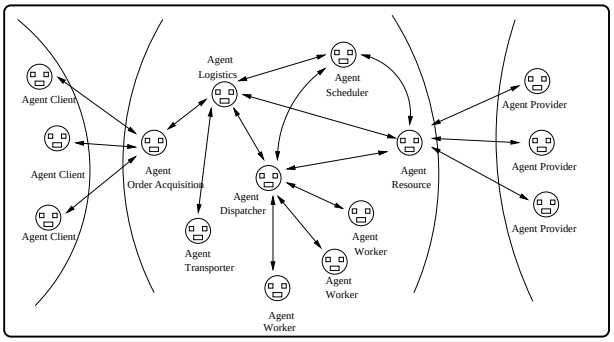

Fig. 2. A Multiagent Approach of the Supply Chain Management

plan is available, it is sent to the agents *Transporter*, *Dispatcher* and *Resource*. The agent *Transporter* is responsible for the transportation of the materials and the final product 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.

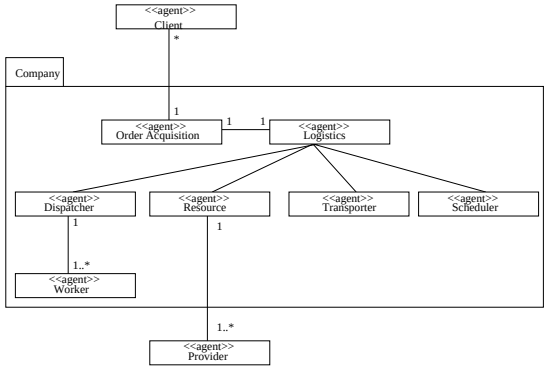

Fig. 3. Conceptual Level of the Supply Chain Management

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. Figure 3 gives one package called *Company* containing all the agents working 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 Figure 3, 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 a particular diagram.

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

2.2 Implementation Level

The conceptual level defined in the previous section is too abstract to be considered for an implementation. As soon as designers want to implement their systems, they need to have an accurate 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 one. Actually, it is required 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 could have when designing agents: (1) an *agency* view where it is given agents' knowledge, beliefs, intentions, plans and behaviors, (2) an *environmental* view where it is given how agents react to the events coming from the environment and other agents, (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 are their roles in these organizations.

We add several new compartments to UML class diagrams as shown on Figure 4. UML class diagrams only consider attributes and operations. These two compartments are insufficient to represent the agent complexity. Figure 4 summarizes the information defined below.

Agency View Several information have to be supplied according to the agency view. An agent is represented by a name which could be a generic name if this agent class is a pattern for agents. An agent class is denoted by the stereotype `«agent»` and a unique agent name. Agent name is insufficient to represent the role of the agent in the multiagent system. We add the compartment *Role* for this purpose. This compartment describes what roles are played by this agent. This information is then used for interaction protocols and organizations.

Like UML classes, agents are characterized by several attributes. These attributes correspond to data used during the agent execution. Common attributes

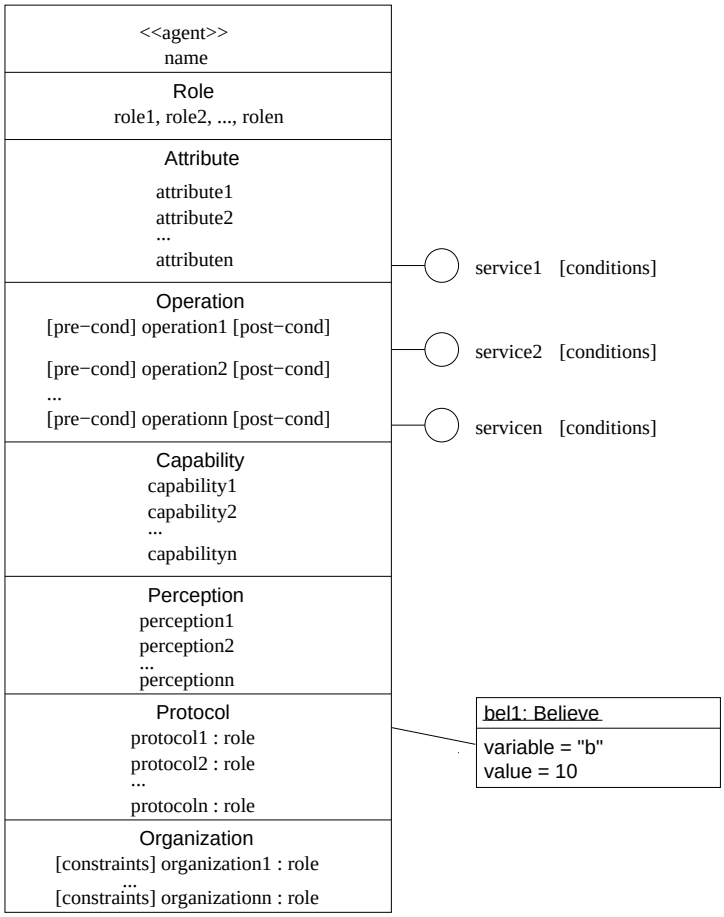

Fig. 4. Agent Class Diagram

are for instance, the agent identifier, its physical address, its profile if it acts on behalf of users, etc. These attributes are stored in the compartment *Attribute*.

We propose to store beliefs, desires, intentions and goals not as attributes but as objects. An object in UML is an instance of a class with well defined boundary. This approach allows designers to represent concrete data within agents. For instance, it is difficult to represent a belief such as *believe(Melbourne sunny)* directly within agent class particularly if we need to modify it after. As a consequence, it is easier to handle these information. When considering a goal, it is then possible to retrieve a part of the goal or to modify it. As far as we are concerned this approach brings flexibility to the management of BDI information. Moreover, it helps designers in the context of mutual beliefs and joint intentions [14] since it is easier to share beliefs or intentions if they are defined outside agents.

Operations are stored in the compartment *Operation*. Operations correspond to actions performed by agents. We divide operations in three categories: internal operations, pro-active operations and reactive operations. Internal operations correspond to operations that are not visible by other agents. For instance, if agents have a specific interaction module for communication [8], such operations could be used to retrieve messages. Pro-active operations correspond to operations that are fired given some conditions such as timer, exceptions or conditions. An usual example is a server which informs agents of a specific value every t units of time. Reactive operations are actions linked to modifications of the environment or in reaction of other agents' actions. These operations are associated to the statecharts corresponding to agent behaviors.

These three kinds of operations are distinguished on class diagrams by a keyword: `int` for internal operations, `pro` for pro-active operations and `reac` for reactive operations.

Environmental View The environmental view deals with what agents do as soon as the environment is modified. In order to represent the modifications and actions, we use statecharts [7]. These statecharts allow designers to represent the agent states and actions needed for going from state to state. Statechart diagram is the solution that we follow even if statecharts are not the only possibility. We can also use activity diagrams or state machines.

Here follows a small example of an agent monitoring the temperature room. The resulting statechart is given in Figure 5.

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

Interactional View The interactional view gives the interaction protocols used by this agent. Interaction protocols are represented by sequence diagrams [12] [2] and extends UML sequence diagrams. Sequence diagrams allow designers to represent agents by their role in this interaction. The messages are sent between roles. The example of the English auction protocol is given in Figure 6.

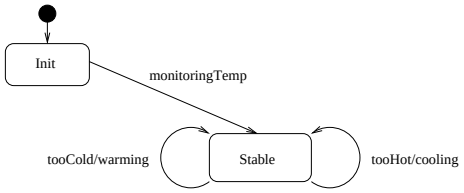

Fig. 5. Statechart Diagram for Agent Monitoring Room Temperature

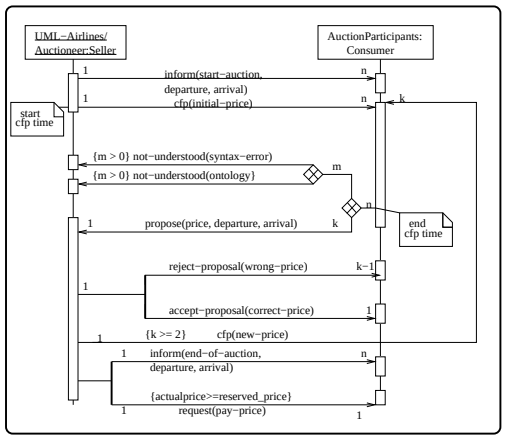

Fig. 6. English Auction Protocol in Agent UML

Sequence diagrams are not defined within class diagrams but outside, only the name of the protocol diagrams are given in class diagrams. The roles played by the agent are given for each sequence diagram.

Organizational View Agents are represented in multiagent systems by their roles and their actions. Several information are provided according to the organizational view: capabilities, services and organizations in which agents involve.

Capabilities describe what agents are able to do. Capabilities are written as a free-format text. Capabilities are stored in the compartment *Capability*. Capabilities are derived as services to agents. These services are not rendered within agent class diagrams but as lollipops linked to agent class diagrams. We add the name of the service and the conditions under which the service is available.

The last information in the organizational view is the list of organizations, the agents belong to. This list of organizations is stored in the compartment *Organization*. Agents play roles in organizations. Designers insert the roles played for each organization. We also add the constraints imposed for entering into, belonging to and leaving this organization.

3 Comparison with Bauer's approach

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, services and supported protocols
6. Organization
7. Agent head automata

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 compartment *Attribute*. The difference between our compartment *Attribute* and this compartment *State description* is for the computation of beliefs, desires, intentions and goals. Bauer considers these information as well formed formulas. As far as we are concerned, we think that it is better to represent beliefs, desires and intentions as objects linked to the agent class. As a consequence, it is easier to modifiy the belief or to handle information. For instance, if the agent learnt that the value of b is no longer 5 but 10. It is easier to modify the associated belief since the agent has just to modify the attribute value of the belief. This approach is also more efficient for retrieving parts of goals or to merge different sub-goals. Last advantage of such

a method is in the context of mutual beliefs, joint intentions and shared plans. It is easier to share these elements since they are defined outside agents.

Actions corresponds to two kinds of actions in Bauer's approach: pro-active actions and reactive actions. Pro-active actions correspond to actions fired by the agent itself such as exceptions or triggered actions. Reactive actions are linked to the modification of the environment. These two kinds of actions are equivalent to ours. We also propose internal actions.

Methods corresponds to our compartment *Operation* when operations are internal ones.

Capabilities corresponds to our compartment *Capabilities*. *Services* are described within agent class diagrams. It is not explained how these services are described. In our proposal, we follow the approach used in UML for interfaces for rendering services.

Supported protocols corresponds to our compartment *Protocol*. However, we add the roles played by the agents. Actually, it is possible that agents play several roles in the same protocol.

Organization corresponds to our compartment *Organization*. One more time, we add the notion of roles played by the agents. Moreover, it is not clear what the piece of information *constraint* means. In our case, we have the information about the conditions to enter the organization, the conditions to stay in it and the conditions to leave it.

The main difference between Bauer's approach and ours remain in the *Agent head automata*. Bauer proposes to render on agent class diagrams, the matching of communicative acts. This approach seems to be unrealistic as soon as agents encompass several complex protocols. This agent head automata contains communicative acts as well as actions triggered by the receipt of messages. In our opinion, this rendering must not be done within agent class diagrams but outside. It is better to use sequence diagrams if designers want to represent protocols or activity diagrams if they want to represent the actions triggered by the messages. Moreover, if designers insert an agent head automata within agent class diagrams, they break one principle of UML saying that one diagram is only performing one kind of view of the system: agent class diagrams are used to represent the content of agents and not how they perform actions linked to messages.

To sum up

Several elements are present both in Bauer's approach and in our approach even if sometimes, the term used is not the same: roles, attributes, operations, actions, capabilities, protocols or organizations. The main differences between our two approaches are located in the management of beliefs, desires, intentions and goals; and in the agent head automata. Beliefs, desires, intentions and goals are stored as objects. This approach makes the handle ease. The second main difference is the agent head automata. In our opinion, this agent head automata is inefficient as soon as agents encompass complex protocols and that several actions have to be done between the receipt of messages and the sending of

messages. Moreover, this approach is contrary to one principle in UML saying that one diagram is designed to perform only one view of the system.

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 [12] 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 classes. 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 future directions 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. Müller, 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. K. Fernandes and M. Occello. A recursive approach to build hybrid multi-agent systems. In *III Iberoamerican Workshop on Distributed Artificial Intelligence and Multi-Agent Systems, SBIA/IBERAMIA*, Sao Paulo, Brazil, November 2000.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
8. M.-P. Huget. Design agent interaction as a service to agents. In M.-P. Huget, F. Dignum, and J.-L. Koning, editors, *AAMAS Workshop on Agent Communication Languages and Conversation Policies (ACL2002)*, Bologna, Italy, July 2002.
9. C. A. Iglesias, M. Garijo, and J. C. Gonzalez. A survey of agent-oriented methodologies. 1999.
10. J. Müller. *The Design of Intelligent Agents - a layered approach.* Number LNAI 1177 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1996.
11. J. Odell. Objects and agents compared. *Journal of Object Computing*, 1(1), May 2002.
12. 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.
13. J. Swaminathan, S. Smith, and N. Sadeh-Koniecpol. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences*, April 1997.
14. M. Wooldridge. *Reasoning about Rational Agents.* MIT Press, 2000.
15. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.