Specifying Agent Interaction Protocols with Standard UML

Jürgen Lind

iteratec GmbH
Inselkammerstr. 4
D-82008 Unterhaching, Germany
jli@agentlab.de

Abstract. In this paper, I will demonstrate how the Unified Modeling Language (UML) can be used to describe agent interaction protocols. The approach that is presented in this paper does not propose enhancements or completely new diagrams but instead relies on existing UML elements and the UML extension mechanism that is part of the standard. This conformity with the base UML is a major advantage of the idea as it prevents a diversification of the UML into different potentially incompatible dialects. The practical use of the method is demonstrated with an example on how to specify a realistic agent interaction protocol.

1 Introduction

One of the currently most popular graphical design languages is the Unified Modeling Language (UML) [3], [17] that aims at a global standard for the description of software systems. Such standardized blueprint languages already exist for electrical, mechanical or civil engineering for several years. The advantage of a blueprint language for software systems is that it provides a set of symbols and mechanisms together with well defined semantics that enables software designers from all over the world to express, exchange and work on their ideas without complicated and error-prone translation processes. Furthermore, a unified language increases the inter-operability among software design tools and allows software developers to become more independent of particular development environments and to assemble customized environments out of different tool suites. The UML combines original ideas with established features of other graphical design languages into a coherent framework that allows for the specification of a broad range of design aspects of a software system.

Due to the strong focus on object-oriented software design, however, the UML is not right away suitable for agent-based systems. In order to make it fit some special requirement of agent-oriented software, there are two possible ways to be taken. One way is to extend the UML by providing new structural elements and diagrams that enhance the expressive power of the base language. This way is favored by the developers of AUML [2], [1], [14] which proposes an extension of the UML with respect to agent-oriented concepts. This approach, however, has the major drawback that it violates the idea of the UML as a general design language. To quote from [17], p. 103: “Many modelers wish to tailor a modeling language for a particular application domain. This

carries some risk, because the tailored language will not be universally understandable, but people nevertheless attempt to do it." Thus, if each group within the computer science community added their own UML extension according to their particular needs, the base language is likely to be split up in several increasingly unrelated dialects. The result, as it can be observed with programming languages such as Basic, is a collection of inconsistent language fragments. Besides this not being the idea of a standard language, it introduces the additional difficulty of having to learn a new dialect when switching between two specialized application fields. Furthermore, tool support is usually not available for special purpose diagrams.

As a consequence from the above considerations, I suggest to take another approach to the use of the UML for describing agent-specific aspects of a software system. A major goal is to remain within the boundaries of the original language and to use only those extension mechanisms that were explicitely admitted by the language designers [3]. Thus, I will not introduce completely new diagram types or the like but instead rely on the provided structural elements and use them to model the system of agent-based applications.

In this paper, I will demonstrate how the UML can be used to capture one of the core concepts of multiagent systems – *interaction*. Interaction is the foundation for cooperative or competitive behavior among several autonomous agents and thus encapsulates the most fundamental design decisions within the development of multiagent systems. Before interaction can take place, however, some technical and conceptual difficulties must be solved. First of all, the agents must be able to understand each other. Mutual understanding is achieved by relying of accepted formal or informal standards where the de-facto standard of todays agent applications seems to be KQML [7], others can be found in [5] or [8]. Although agent communication languages are an important aspect of multiagent systems design, these aspects are not covered by this investigation of the UML as interaction description language. Instead, this paper will focus on the second important aspect of agent interaction which is that the agents must know which messages they can expect in a particular situation and what the are supposed to do (e.g. sending a reply message) when a certain message arrives (or does not arrive for a given period of time). This part of the interaction process is controlled by *interaction protocols* (or simply protocols).

For an example of an interaction protocol, consider an English auction. There, an auctioneer offers a product at a particular price to a group of bidders. Each of the bidders individually decides to accept that price or to decline the offer. If one of the bidders accepts the current price, the auctioneer raises the price by a fixed rate and asks the group of bidders again if any of them accepts the new price. If this is the case, the price is raised again and the cycle repeats until none of the bidders is willing to pay the current price. Then, the last bidder who accepted the price is given the product.

In this example, we can identify the major elements of interaction protocols. First, we can separate the participating agents into different groups. In this case, we have two groups: the auctioneer and the bidders. Each group has a set of associated incoming and outgoing messages an internal functions that decide about their next action. I will refer to the set of messages and behaviors that are associated with a group of agents as a *role* that can be played by an agent. Please note that agents are not limited to a single

role, e.g. the auctioneer in the previous example can be a bidder in another auction at the same time. The second important aspect of an interaction protocol besides the participating roles is the temporal ordering of function evaluation and the messages that are exchanged. For example, it would not make sense or would be impossible for the bidder to decide on an offer and to decline it before it has even received the offer. Therefore, the interaction protocol determines the flow of control within each role as well as between different roles.

It is precisely the dualism mentioned in the previous paragraph that makes protocol design a difficult task. There are not only intra-role aspects to consider during the design process, but also inter-role dependencies induced by the other roles. Even worse, there is currently only little software engineering support for the design of interaction protocols. A number of protocol specification languages have been proposed ranging from specification languages for low level communication protocols [18], [9] up to high level specification languages for multiagent applications [4], [10]. Up to now, however, none – perhaps except for Estelle – of these languages has gained wide-spread acceptance. Estelle [18], is a specification language for service description and system behavior in telecommunications that uses extended finite automata to describe the intended behavior. Extended finite state machines are normal finite state machines plus (typed) variables. The state in the finite state machine has a set of associated variables that can be queried and/or manipulated in the transition specifications. In Estelle, a protocol is a collection of several distinct automata where each automaton can have an arbitrary number of interaction points with other automata. These interaction points are called *channels* and they control the message exchange between different automata. Estelle is a very powerful language that was mainly developed for the specification of low level protocols. It is therefore not directly suitable for the use in multiagent applications.

One reason for the lack of acceptance mentioned above is probably the fact that protocol description languages usually provide only a text-based representations for the interaction protocols. This makes it hard, especially for complex protocols, to understand the flow of control within the protocol. An alternative for these text-based languages are therefore graphical languages that make the described protocols more accessible for the reader. As mentioned above, I argue that the UML allows the software engineer to specify the interaction schemes that can be found within a multiagent system. In an earlier approach described in [12], I have proposed a method to describe interaction diagrams using a standard diagram type provided by the UML with minor modifications of the proposed standard elements. The modification that I found necessary in my earlier work, however, have shown to be unnecessary now that I have gained greater knowledge of the UML meta-model that allows for defining new UML elements within well-defined bounds. Basically, this paper is a revised version of the Section in [12] that corrects the errors that have been made there.

2 Related Work

In the previous section, I have already mentioned some protocol specification languages that have been proposed to describe interaction protocols within agent-based systems.

One of the most recent approaches for modeling agent-specific aspects of a software system is the AUML approach mentioned above. As part of AUML, the authors suggest an extension of the UML by introducing a completely new diagram type called *protocol diagrams*. These diagrams combine elements of UML interaction diagrams and state diagrams to model the roles that can be played by an agent in the course of interacting with other agents. The new diagram type allows for the specification of multiple threads within an interaction protocol and supports protocol nesting and protocol templates based on generic protocol descriptions. In [19], an extension of this UML extension is proposed; a comparison of the two approaches can be found in [11]. As I have argued earlier, however, I see a major problem in this approach as it supports a diversification within the UML community that may not be in the sense of the original inventors.

The Protoz [15] protocol specification environment features a specification language that is related to Estelle [18] and that is based on a similar computational concept. However, due to the focus on multiagent specific aspects, Protoz provides a more accessible interface to protocol design. The main tool of the protocol environment is a compiler that generates Oz code [16] from a given protocol specification, a graphical notation is currently not available. In the Protoz environment, a protocol is defined by a collection of roles where each of these roles is specified as an extended finite state machine. The state machine transitions fire upon incoming messages; messages can stem from other agents or from internal procedures. These internal procedures implement the connection to the application and allow for a uniform modeling of internal and external communication.

The ZEUS development environment [13] from BT is a a design method and tool collection for the engineering of distributed multiagent applications. The ZEUS tools all encompass the direct-manipulation metaphor and allow the designer to use drag-and-drop technology to assemble the application from pre-defined components. The tool-kit allows the designer to specify models for different types of agents, for the organizational structure of agent societies and for negotiation models. The negotiation models are either pre-defined or the can be build by the designer if no appropriate pre-defined model is available for a particular task. In [6] a notation for role models is presented that originates from UML class diagram notation and that contains also elements from UML interaction diagrams (e.g. message sequencing). The ZEUS role models capture structural (static) relationships between roles as well as communicative acts that describe the dynamic aspects of inter-agent communication. The pre-defined role models that are provided by the ZEUS environment include various protocols from the trading domain as well as business processes such as supply chain management.

3 UML Activity Diagrams

Activity diagrams in UML models provide a number of structural elements as shown in in Figure 1 to describe algorithms in a flowchart like manner. To this end, each computation is expressed in terms of *states* and the progression through these states. In order to allow for a hierarchical modeling, the UML distinguishes between two classes of states. *Action states* are atomic entities that cannot be decomposed and that relate to

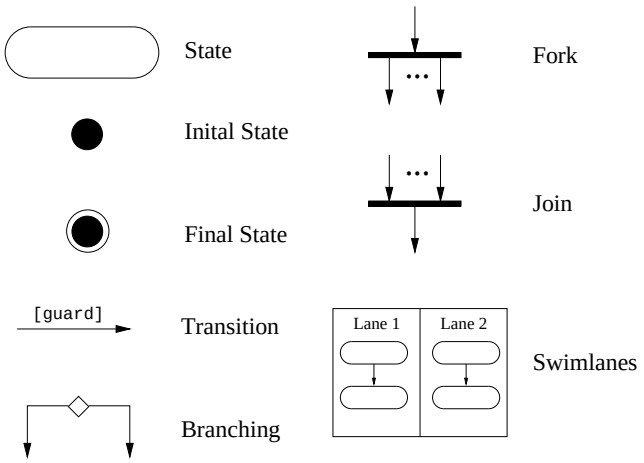

Fig. 1. Structural Elements of UML Activity Diagrams

atomic statements in a programming language, eg. variable assignment. *Activity states*, on the other hand, represent a collection of atomic states and can thus be decomposed into these atomic states. Furthermore, the execution of an activity can be interrupted between any two subsequent states. In terms of programming languages, actions relate to statements and activities relate to subroutines.

The states of an activity diagram are linked with each other through *transitions* that indicate the control flow within the activity diagram. Each transition can have a *guard* condition that controls the flow of control in that it only allows a transition to fire if the guard condition is true. Because of the basic requirement that each transition must have at least one start and one end point, special states are introduced that represent the beginning and the end of an activity diagram, respectively.

The control flow within an activity diagram is not necessarily linear, otherwise it would be impossible to express anything other then trivial algorithms. Therefore, *branching* elements that represent the decision points within a diagram are provided. Each branching points stands for a boolean decision, i.e. the flow of control can proceed along two different paths.

Many modern programming languages provide some notion for pseudo-parallel program execution within a single operating system process. These light-weight processes – usually referred to as “threads” – can be modeled in UML activity diagrams by using two structural elements. A *fork* operation splits a single thread of execution into two or more threads that are subsequently executed in parallel. Thus, a fork bar has one incoming transition and several outgoing transitions. In order to merge several of these parallel threads into a single thread again, UML activity diagrams provide the *join* element. Thus, a join barrier has several incoming transitions and only a single outgoing transition, it can therefore be used to synchronize several parallel threads of execution. Note that a join barrier waits until *all* incoming threads have arrived at the barrier before proceeding with the single master thread.

Because of the fact that activity diagrams tend to become somewhat confusion with growing in size, UML activity diagrams can contain so-called *swimlanes* that are used to partition an activity diagram into several conceptually related parts. Within an activity diagram, each swimlane must have a unique name and each activity must belong to exactly one swimlane.

4 Tailoring UML

The UML has built-in extension mechanisms based on *constraints*, *tagged values*, and *stereotypes* that makes it possible to create UML profiles for particular application domains. A UML profile is a collection of modeling elements together with well defined semantics of these elements and the possible relations between them. For the purpose of this paper, stereotypes are sufficient; for a general UML profile for agent-based applications, all three extension mechanisms are likely to be necessary.

Stereotypes are new model elements that are declared within the model itself, i.e. stereotypes extend the modeling capabilities by introducing new classifiers that may extend the semantics but not the structure of existing meta-model classes. As an example for a stereotype, consider a business application where we want to deal with business processes explicitly. We can then introduce the `business process` stereotype as a means to describe a special kind of classes with attributes and methods but with additional constraints on usage and allowed structural relationships within the design model. Each stereotype must be based on an existing modeling element, this enables tools to deal with arbitrary stereotypes in the same way as with the respective base elements. To visually distinguish stereotypes and standard UML modeling elements, each stereotype can have its own icon. Furthermore, it is possible to define hierarchies of stereotypes with inheritance between them and using meta-model class diagrams to visualize the relationships between stereotypes. To store additional information about an instance of a stereotype, the creator of a stereotype can define a list of required tags that must be set whenever a stereotype instance is created. The information kept in the tagged values can, for example, be used by automatic code generators.

In the following section, I will define a couple of stereotypes that are necessary to model agent interaction protocols. For other aspects of agent-based systems, additional stereotypes will be needed.

5 Protocol Specification with Activity Diagrams

In this paper, I propose a notation for interaction protocols that is based on the basic elements of UML activity diagrams. In order to make them more usable to describe agent interaction protocols, I will introduce several stereotypes that relate the basic elements to the specific application area. First of all, I will extend the idea of swimlanes as a means to describe the *roles* (stereotype `<<role>>`) that occur within the application. In my view, these swimlanes are interpreted as physically – as opposed to conceptually – separated flows of control. I will sometimes refer to these independent flows of control as *control flow spaces* in the rest of this paper. The roles within the diagrams are linked with each other via explicit communication *channels* (`<<channel>>`) that

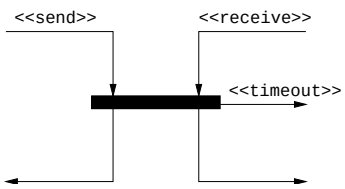

Fig. 2. Synchronization Point

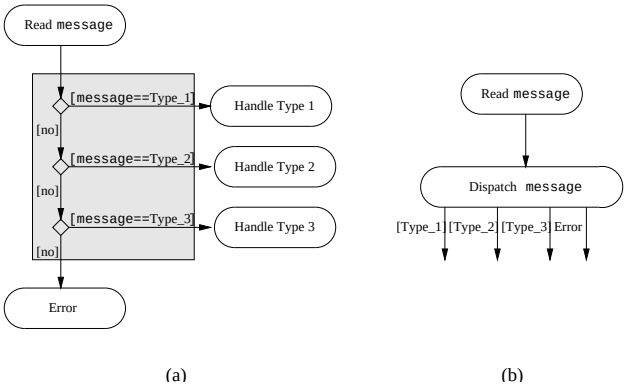

Fig. 3. Defining Macros

manage the message exchange between two roles. The message exchange itself is modeled in *synchronization points* (`<<synchronization point>>`) that denote the sending and the reception of messages, respectively. The graphical representation of a synchronization point is shown in Figure 2 where the arrows on either side denote the the control flow of the sender and the control flow of the receiver, respectively.

Each synchronization point has several incoming transitions out of which exactly one must be a `<<send>>` operation. The other transitions are the receivers of the respective message. Whenever the control flow of a receiver enters a synchronization point, the receiver suspends until a message has been delivered. This happens whenever the control flow of the sender reaches the synchronization point. After the massage has been delivered, the control flow of the sender and the control flow of the receivers resumes after the synchronization point. In order to prevent the receivers from infinite blocking while waiting for a message that never arrives, an additional `<<timeout>>` transition for each receiver can be attached to the synchronization. Whenever the timeout is reached and no message has been delivered, the control flow of the respective receiver resumes at the state pointed to by the timeout transition.

Note that the `<<synchronization point>>` stereotype includes a *semantic* extension of the UML because no such thing as a “timeout” is defined for join elements of standard activity diagrams. As explained in the previous paragraph, however, this semantic extension (which is still covered by the bounds of the extension mechanisms

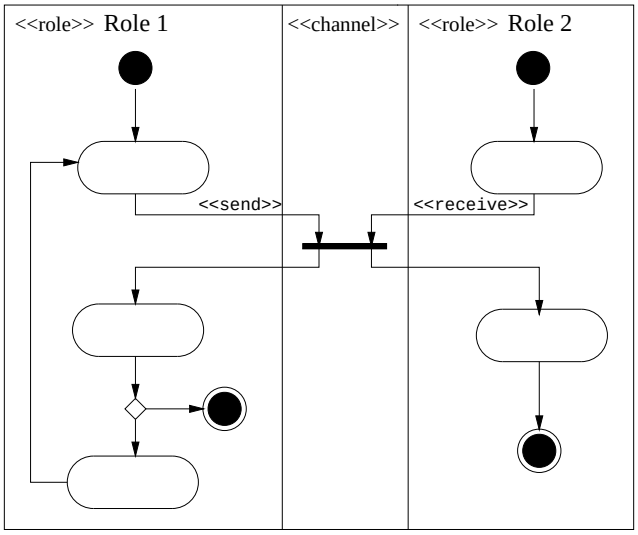

Fig. 4. Augmented Activity Diagram

described above) is necessary to prevent either sender or receiver from infinite blocking. Note also, that it is not possible to express *asynchronous* message exchange with the protocol description mechanisms presented in this paper. For such diagrams, another modeling element would have to be defined either as stereotype or as new UML elements. Thus, here we might have the case that the UML is not sufficient and would need some extension. I will return to this later in the conclusion.

A very important feature of UML diagrams is that they provide a powerful structuring mechanism that can be used to make protocol mode readable. Since activity states can represent complete automata, it is straightforward to use them for macro definitions that can be used in interaction protocols. Figure 3 illustrates the idea. Figure 3 (a) shows an activity diagram for dispatching an incoming message according to the message type. Using the UML rule that a state can have several outgoing transitions that are labeled with conditional statements, we can rewrite the shaded part of the original automaton that contains three branching points into a single state as shown in 3 (b)[1]. Collapsing several states into a single macro state has not only the advantage to make a diagram more readable, it is also important that the macro state can be given a speaking name that highlight its purpose. Although the overall gain seems to be pretty small in the above example, the gain soon becomes apparent in more complex protocols where each decision point or loop construction that is hidden improves the readability of the protocol. Furthermore, this mechanism can be used to embed protocols into others, allowing for a hierarchical structuring, flexible combination and re-use of protocols.

The use of the various modeling elements for specifying agent interaction protocols is shown in Figure 4. The swimlanes indicate the control flow spaces that are associated

[1] Note that conditions on the outgoing transitions are abbreviated in the example.

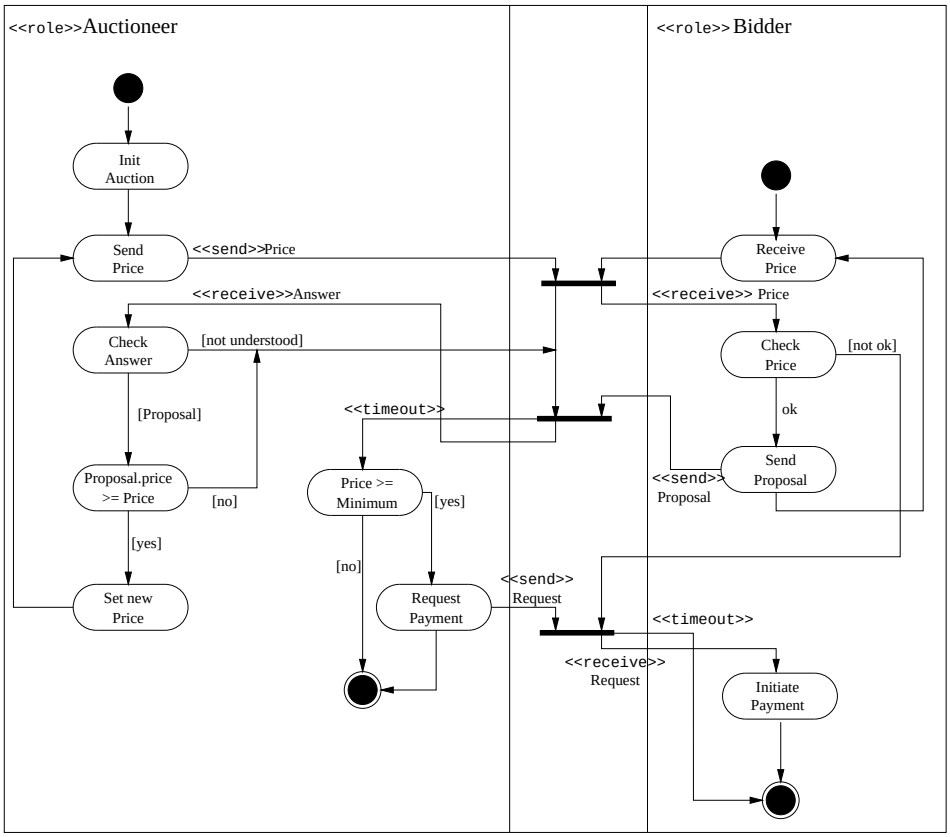

Fig. 5. English Auction

with each role within the agent interaction protocol. The control flow of each of these roles is modeled using the structural elements that are provided by standard UML activity diagrams. The self-contained control flow spaces are linked via a communication channel that holds one synchronization point that links the activity diagrams of the interacting roles. In the following section, we will see, how the modeling elements and the new stereotypes can be used to specify real agent interaction protocols.

6 Example

In order to illustrate the use of UML activity diagrams for interaction protocol specification on a realistic example, recall the English Auction that was mentioned in the introductory section. In Figure 5, I have depicted an interaction protocol that describes the course of actions and message exchanges within the auction more formally.

The first step in the interaction design process is to identify the roles that interact with each other. In the example, we have already identified the *auctioneer* and the

bidder as the participating roles. Now, we create a control flow space that will later hold the finite automaton that describes the behavior of the agent playing a particular role. It is usually a good idea to develop an initial version of each automaton without considering the other automata, i.e. without switching back and forth between different automata. Thus, for the auctioneer, the auction starts with an initialization of its internal data, e.g. with determining the initial price of the product. Then, the auctioneer sends out a proposal to the bidders and waits for the incoming replies. In order to make the example more realistic, we shall assume that a bidder can indicate that the proposal was not understood, e.g. because the bidder is not familiar with the ontology used. In that case, the auctioneer simply ignores the message and continuous to wait for further messages. If, on the other hand, the price is accepted by the bidder, the auctioneer raises the price according to a fixed rate and the cycle starts from the beginning. In the offer is not accepted by the bidder, the auctioneer continuous to wait for incoming replies until a fixed timeout. When the timeout has expired and no bidder has accepted the offer, the product is given to the last bidder that has accepted the price (if that price exceeds a previously defined minimal acceptable price). Please note, that the *CheckAnswer* state uses the macro mechanism explained earlier to dispatch the incoming messages.

Now that the behavior of the auctioneer has been fully specified, we can turn to the bidder role. In the example, the bidder goes into a waiting loop as soon as the protocol execution is started. It leaves this loop when it receives an offer proposed by the auctioneer and checks whether to offered price is acceptable according to its individual goals. If this is the case, the bidder sends out a positive reply and re-iterates the waiting process. If the actual price is not acceptable, the bidder waits for a message from the auctioneer that indicates if the bidder is given the product or nor. Obviously, this can only happen when the bidder has issued a positive reply during the auction. To avoid an infinite blocking of the bidder, a timeout is applied to terminate the waiting process after a finite time. The bidder that receives the positive acknowledgment from the auctioneer will immediately initiate the payment process to finally receive the product.

This small example should be sufficient to provide the reader with an impression on how to apply the suggested method to arbitrary agent interaction protocols. The best way to see how the method works in practice is to pick an (preferably easy) protocol from the application domain of interest and then to simply start right away with an iterative modeling process. The value of the diagrams will then quickly become apparent.

7 Conclusion

In this paper, I have demonstrated how UML activity diagrams can be used for the specification of agent interaction protocols. The suggested method uses existing UML concepts and requires no additional elements, therewith making it easy for UML users to understand the interaction protocols without having to learn a completely new type of diagram. The method that was explained in this paper has been used in practical situations and has shown to be a valuable tool for modeling, understanding and communicating agent interaction protocols.

At the beginning of the paper, I have argued that, in my view, a special UML version only for agent-based systems is not desirable because of the potential emergence

of mutually incompatible UML dialects. This does not mean, however, that the UML in its current version is perfect. Certainly there are aspects that need further elaboration and improvements and the requirements in the design of agent-based applications that provide input on which features should be added or improved are absolutely necessary. Approaches such as AUML can help to identify the potentially problematic or insufficient parts of the UML. The resulting extensions and improvements of the UML, however, should be chosen such that they are useful not only for agent-based systems, and they should be part of the general standard and not just a dialect thereof. In this sense, we shall continue to identify agent-specific requirements for the UML, try to solve the upcoming problems within the standard and suggest extensions only in those cases where solutions within the standard are not possible.

Acknowledgments

I would specifically like to thank Jeremy Pitt for the discussion on the weak points of an earlier version of this paper. His comments caused me to investigate the UML extension mechanism in greater depth in order to get them fixed.

References

1. Bernhard Bauer. UML Class Diagrams Revisited in the Context of Agent-Based Systems. In *Proceeedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, Canada, 2001.
2. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2001. Springer Verlag.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
4. B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic configurable cooperation protocols for multi-agent systems. In C. Castelfranchi and J.-P. Müller, editors, *From Reaction to Cognition — 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'93)*, volume 957 of *LNAI*, pages 157–171. Springer-Verlag, 1995.
5. S. Bussmann and H. J. Müller. A Communication Structure for Cooperating Agents. *Computers and AI*, I, 1993.
6. J. Collins and D. Ndumu. The ZEUS Role Modelling Guide. Technical report, BT, Adastral Park, Martlesham Heath, 1998.
7. T. Finin and R. Fritzson. KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the 13th International Distributed Artificial Intelligence Workshop*, pages 127–136, Seattle, WA, USA, 1994.
8. FIPA. *AgenTalk Reference Manual*. NTT Communication Science Laboratories and Ishida Laboratory, Department of Information Science, Kyoto University., 1996.
9. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
10. M. Kolb. A cooperation language. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 233–238, June 1995.
11. Jean-Luc Koning, Marc-Philippe Huget, Jun Wei, and Xu Wang. Extended Modeling Languages for Interaction Protocol Design. In *Proceeedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, Canada, 2001.

12. Jürgen Lind. *Iterative Software Engineering for Multiagent Systems - The* MASSIVE *Method*, volume 1994 of *Lecture Notes in Computer Science*. Springer, May 2001.
13. Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collins. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.
14. H. V. D. Parunak and James Odell. Representing Social Structures in UML. In *Proceeedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, Canada, 2001.
15. Stefan Philipps and Jürgen Lind. Ein System zur Definition und Ausführung von Protokollen für Multi-Agentensystemen. Technical Report RR-99-01, DFKI, 1999.
16. Programming Systems Lab. The mozart programming system. University of the Saarland, 1999. http://www.mozart-oz.org.
17. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addision-Wesley, 1999.
18. The International Organization for Standardization. IS-9074 (Information processing systems/Open systems interconnection): Estelle — a formal description technique based on an extended state transition model, 1997.
19. J. Wei, S.-C. Cheung, and X. Wang. Towards a Methodology for Formal Design and Analysis of Agent Interaction Protocols. In *Proceeedings of the International Software Engineering Symposium*, Wuhan, Hubei, China, 2001.