Specifying Agent Behaviors with UML Statecharts and StatEdit

Jan Murray*

Institut für Informatik
Universität Koblenz-Landau, Campus Koblenz
Universitätsstraße 1, D-56070 Koblenz, Germany
murray@uni-koblenz.de

Abstract. The use of agents and multiagent systems is widespread in computer science nowadays. Thus the need for methods to specify agents in a clear and simple manner arises. One way of achieving this is by means of a graphical formalism. For using such a formalism the availability of tools, that support a developer, is of great importance. In this paper we present an approach to specifying agent behaviors on different levels of abstraction with the help of UML statecharts. Cooperation between different agents can explicitly be modeled. To help a developer with applying this formalism to the specification of agent behaviors the statechart editor StatEdit is presented. This development tool supports not only the modelling of an agent but a simple form of code generation as well.

1 Introduction

The use of agent technologies and multiagent systems has gained entrance into almost all branches of computer science. Thus the need for standards and design techniques has arisen. In order to gain wide acceptance an agent specification and design procedure must fulfill several constraints. It has to be as precise as possible to avoid ambiguities in the design of an agent. Nevertheless, the formalism must be easy to understand and use. It would also be desirable, that the application of formal methods is supported by the specification mechanism. This calls for a formal semantics.

All of these requirements can be met by a graphical formalism. Such a formalism is usually easy to understand. The definition of the individual graphical elements that make up the formalism can be done in an unambiguous way. Defining a formal semantics is also possible. Another thing that is always desirable for any formalism, especially graphical ones, is the availability of development tools which help a developer model and implement a system. In this paper we present both, a graphical way of modelling agent behaviors and an editor for working with this formalism.

Our approach is based on UML statecharts [11]. With this approach it is possible to specify not only behaviors for single agents on different levels of abstraction, but multiagent plans as well. Statecharts are a means for describing the behavior of a system in response to external events. Their graphical notation is intuitive and easy to understand. In addition to that the use of UML as a specification and modelling language is already

* Supported by the grant *263/8-1* from the German research foundation *DFG*.

D. Polani et al. (Eds.): RoboCup 2003, LNAI 3020, pp. 145–156, 2004.

widely accepted. Although the specification of UML statecharts does not provide a formal semantics, work on this has already been done, mostly with the aim of verifying properties of UML models, e.g. in [8, 14]. A tool for editing statecharts is also presented. With the StatEdit editor an agent designer can easily create statecharts describing agent behaviors. The resulting statecharts can also be exported to a number of other formats, which helps a developer with creating running code from the specification.

The rest of the paper is organized as follows. Section 2 briefly summarizes the relevant parts of the UML statechart formalism. Section 3 introduces our approach to agent specification with UML. The specification is described on different levels of abstraction. Special attention is laid on the specification of multiagent plans (Section 3.3). In Section 4 the statechart editor StatEdit, which is being developed at the University of Koblenz, is presented. This editor allows for the graphical design of agent behaviors and exporting the resulting statechart to different formats, e.g. Prolog or XML. Section 5 finally concludes the paper with an overview over related work, some final remarks and an outlook to future work.

2 UML Statecharts

The behavior of a system – like a program or an agent – can be described as a sequence of states the system is in. Depending on (external) events the system changes from one state to another. Such a change may be accompanied by the execution of an action.

In the Unified Modelling Language (UML) [11] *statecharts* are used to model behavioral aspects of systems. Statecharts are basically directed graphs, where different kinds of nodes represent states and pseudostates and edges stand for transitions. Labels on the edges describe properties of the transitions. Statecharts are *hierarchical* state transition diagrams, i.e. states in a statechart can contain other states or even whole state machines. In the following we will briefly summarize those parts of the UML statechart formalism that are employed by us for the design of agents.

In UML a *state* (represented as a box with rounded corners) is considered a period in the life of a system/agent during which a certain condition holds or an activity is performed. An agent may for example remain in a state while it waits for some external event to occur. In UML three different types of states are distinguished. *Simple states* are atomic in the sense that they do not possess any internal structure. *Composite states* can be further decomposed. They contain submachines which describe the activity associated with the composite state. *Concurrent states* are special types of composite states. A concurrent state contains two or more composite substates, which are called *regions*. If an agent is in a concurrent state, it is in *all* regions simultaneously. Thus concurrent states are used to model concurrent activities in a system or an agent.

The state of a system can be changed in reaction to external events. Such a *transition* is shown as a directed edge from state s_1 to state s_2, which is labeled with $e[c]/a$, where e is an event, c a boolean expression, and a an action. The (informal) semantics of t is "if the system is in state s_1 and event e occurs and the condition c holds, then the system executes action a and changes to state s_2".

Pseudostates may be seen as transient simple states, i.e. a system cannot remain in a pseudostate – it is left without delay. In other respects a pseudostate behaves just

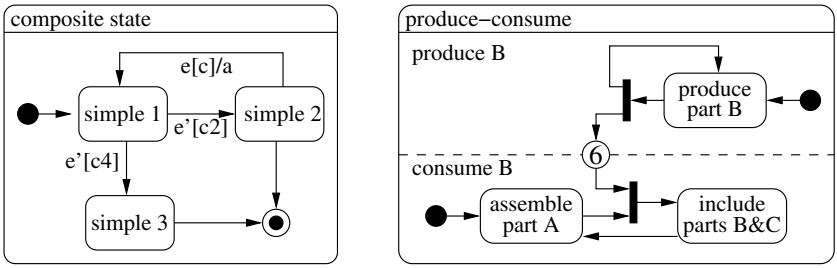

Fig. 1. Statechart examples. On the left a composite state with simple substates is shown. The right statechart shows a concurrent state modelling a producer consumer scenario.

like a simple state. With the help of *fork* and *join pseudostates* concurrent states may be entered and left. The *initial pseudostate* points to a designated start state of a state machine. The *final state* of a state machine is entered, if the activity modeled by this machine is finished. In this case a *completion event* is generated. Figure 1 shows two examples of statecharts.

Sometimes it is necessary to synchronize the concurrent activities in a statechart. This can be done with a *synch state*, which is shown as a circle residing on the dashed line separating different regions. The transition leaving a synch state may only be enabled if the transition entering the synch state has fired at least once. A synch state is labeled either with a positive integer giving an upper bound on the number of times the incoming and outgoing transitions have fired or with an asterisk, if there is no such upper bound.

In a typical producer-consumer scenario, for example, goods are produced by an agent and consumed by another. But the latter can only consume something that has been produced beforehand, so the need for synchronization is obvious. Figure 1 shows such a scenario on the right. Parts of type B are produced by one agent and used by another to assemble components ABC. The upper bound on the synch state is 6, so only six parts of type B may be produced, before one has to be consumed.

3 Designing Agents with UML

The behaviors of an agent can be seen as a sequence of states the agent is in. Each state corresponds to an activity or indicates that he agent is idly waiting for something to happen in its environment. Furthermore the behaviors of an agent can be specified on different levels of abstraction. This makes it possible, for example, to design an agent in a top-down manner by first specifying its tasks and behaviors on a very abstract level and then refining the abstract behaviors more and more.

One important feature of multiagent systems is the (explicit) cooperation of several agents to achieve a common goal or solve a problem. The agents play different *roles* in a shared (or multiagent) plan, i.e. they execute behaviors that solve parts of the problem or support other agents. Thus a role in a multiagent plan can once again be modeled as a sequence of states an agent passes through, partially in response to (external) events.

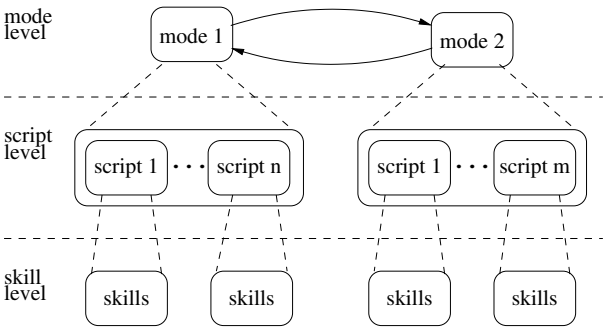

Fig. 2. Layered Architecture of an Agent.

But there is an important difference between the design of a single agent behavior and the role of an agent in a multiagent plan. When different agents work together to achieve a goal, their behaviors are not completely independent of each other, although most of the time they can be executed concurrently. But at several points of the multiagent plan *synchronization* of the individual behaviors of the agents is necessary, because sometimes agents have to work together on a subtask, or one agent has to wait for another agent to finish a subtask.

We present a layered approach to designing agents for the RoboCup Simulation League. For the specification and implementation of an agent three levels are distinguished, each of which is more global than the layer below (cf. Fig. 2).

- On the highest level – the *mode level* – global patterns of behavior are specified. They can be thought of as the most abstract desires an agent has, e.g. attacking or handling standard situations like corner kicks. These abstract desires correspond to different states an agent can be in. We will refer to them as *modes*.
- For each mode an agent has a repertoire of skeleton plans that it can use as long as it does not change its mode. The specification of these plans or *scripts* and their assignment to the global states constitute the second level of the agent design. On this level *explicit* specification of cooperation and multi agent behaviors can be realized.
- On the third and lowest level of the hierarchy the simple and complex actions the agents can execute are described. These actions, the *skills* of an agent, are used in the scripts.

So each level shows details of a higher level. The result of the design process is a layered specification of an agent. The connections between the three levels are not predefined in a rigid manner. Thus a developer can adapt the modelling technique to the kind of agent that ist to be designed. If, for example the statecharts created on all levels are merged into *one* chart, the resulting statechart models a hysteretic agent as described in [13]. If an agent for a special architecture (e.g. SOAR) is modeled, the statecharts are mapped to components of the target architecture.

Throughout the rest of the section we will describe, how UML statecharts are employed in the specification of agents. Section 3.1 explains the high level specification of

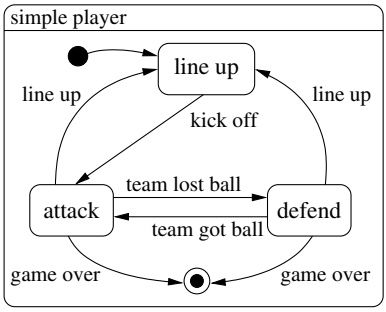

Fig. 3. Modechart for a simple player. The line up event is generated after a goal and at the beginning of each half time.

an agent, while the subsequent sections deal with the design of single agent behaviors (Section 3.2) and multiagent plans (Section 3.3). The skill level will only be addressed very briefly in Section 3.4.

3.1 Mode Level

In robotic soccer an agent frequently switches its behaviors on a very abstract level. For example, an agent may either be *defending* or *attacking*, depending on which team is controlling the ball. All changes of such global behaviors happen in response to one or more external events. If a state is associated with each of the behaviors and the events and conditions that lead to a change from one state to another are determined, the agent can already be modeled by a statechart, called *modechart*, on this very abstract level.

Consider a very simple soccer playing agent with only three such modes. Whenever the agent's team controls the ball the agent is *attacking*. If the opponent team gains control of the ball, the agent switches to a *defensive* behavior. Before each half of the game, as well as after a goal, there is a period in which the teams *line up*. Figure 3 shows the resulting modechart. As the soccer teams line up on the field before the game is started, the line-up state was chosen as the initial state. When the game is over, the agent enters its final state and ceases its activities.

The transitions between different modes are usually triggered by easily observable events and guided by guards, whose truth values can easily be determined. As a lot of behaviors an agent performs only make sense in certain situations, its action selection mechanism is guided on an abstract level by the current mode, and so the search space for selecting an action is pruned. For example, in the line-up mode the agent only has to consider a single action, namely moving to its home positon.

3.2 Script Level

Up to now only very abstract desires of an agent have been specified with the help of modecharts. But nothing has been said about how to achieve those abstract desires.

Each agent is equipped with a repertoire of *scripts*, which are short local skeleton plans for handling particular situations. In each mode an agent can access a subset of

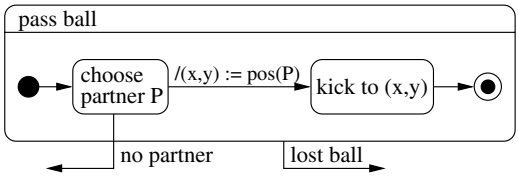

Fig. 4. A script for passing.

all scripts, depending on the situations that can arise in the particular mode. If no script is applicable, the agent has to fall back to a (possibly purely reactive) default behavior. One of the main problems of a past approach [15] to specifying such scripts lay in the rigidity of the plans. Once a script was selected for execution, it was hard to interrupt or abort it. UML statecharts, however, are very well suited for this kind of specifications.

For an agent, the execution of a script means executing a sequence of activities, some of which depend on the outcome of previous activities. As they can be associated with (simple) states in UML statecharts, the whole script can easily be represented by a composite state. Such a *script state* is entered at a designated state representing the beginning of the activity and can be left at a variety of points according to the outcome of the script. Interruption of a script in response to changes in the world are modeled by transitions originating from the edge of the composite state.

Consider the example of a *passing script* in Fig. 4. The agent selects a teammate to kick the ball to, gets its position on the field and finally kicks the ball to those coordinates. If the agent cannot find a suitable teammate or loses the ball, the script is aborted. The agent can lose the ball during either activity, so the transition handling this event originates from the state representing the whole script. But only the *choose partner* activity may fail because the agent cannot find a partner, so the corresponding transition starts from the substate modelling this activity.

3.3 Multiagent Plans

Up to now we have shown how to model scripts or behaviors of a single agent with the help of UML statecharts. But what about multiagent plans? In a multiagent plan or script several agents act *simultaneously* in order to achieve a common goal. At certain points their activities have to be synchronized. Those two additional requirements – concurrency and synchronization – are modeled with the help of concurrent states. A multiagent script is specified as a concurrent state with a region for each role hat has to be played by an agent. If the activities carried out by different agents have to be synchronized, this is modeled with the help of a synch state.

An example may help to clarify this. Consider a typical double passing situation. An agent *A* controlling the ball wants to get past an opponent *O* by playing a double pass. The agent *passes* the ball to a teammate *B* and *runs* past the opponent. The teammate *B dribbles* a little with the ball and *passes* it back to *A* as soon as possible. To handle this situation the agents can be equipped with a multiagent script with two roles that correspond to the behaviors of the agents *A* and *B*. The concurrent state modelling this script is shown by Figure 5. The simple states in the script correspond to the activities

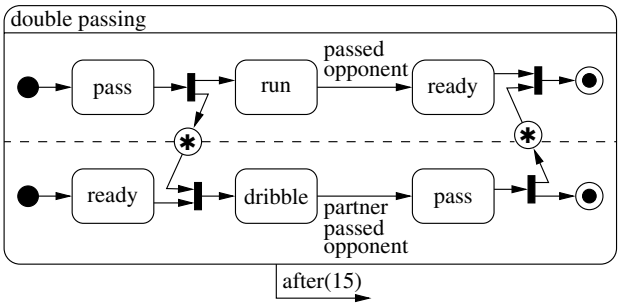

Fig. 5. A multiagent script with two roles for double passing.

of the agents. Synchronization is necessary twice in this script, namely for passing the ball. As an object (the ball) is passed between the agents and both agents can only continue their role at the respective positions if they are in possession of the ball, the need for synchronization is evident. Finally, a timeout for the execution is modeled by the transition labeled *after(15)*, which means that the execution of the script is terminated after 15 simulation steps. In this case the script has failed. Additional error transitions, e.g. modelling loss of the ball, may be added.

So far we have modeled the script as a whole. But some aspects have still been omitted. First of all, we only specified *where* synchronization has to take place, but we did not clarify *how* the activities are synchronized. As the different roles are played by different agents and their internal states are usually not known to teammates, means have to be provided that enable an agent to determine whether its partner has already reached a synchronization point or not. In addition to that an agent can only play one of the roles in a script at a time. Therefore the specification should model not only the script on the whole, but also the individual roles.

So we add a second step to the specification of a multiagent script, in which the behaviors corresponding to each role are derived from the script state to yield an *agent state*. This is done by "cutting along the dashed lines". As each role in the script is modeled by a region in the concurrent state, it is easy to see that the specification of an agent's behavior must be based upon the corresponding region. So the substates of each region are simply copied to the corresponding agent state. This process is quite straightforward, since most of the time the agents' behaviors are independent of each other.

The only spots that require special attention are the synch states. As we said before, a synch state only models the need for synchronization but says nothing about how this synchronization is realized. Unfortunately there is no unique way of handling synchronization in the derivation of an agent state, so the designer has to tackle this issue as the case arises. The required synchronization may, for example, be indicated by the change of a guard condition or the occurrence of an event. It may, however, be necessary for one of the agents to explicitly generate an event, for example by communicating its internal state.

Transitions modelling interruptions or errors are just copied from the script state. If such a transition starts on the edge of the composite state, it has to be copied to the edges

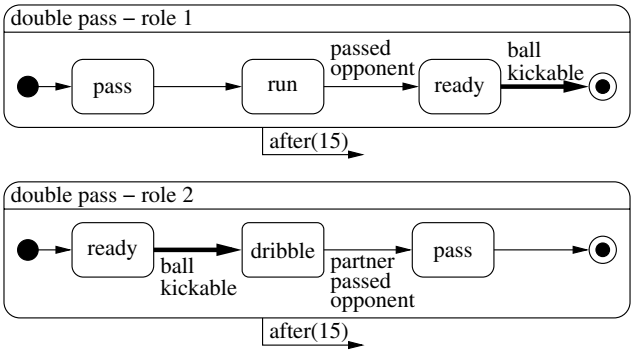

Fig. 6. The derived agent states for double passing. Bold transitions indicate synchronization.

of *all* agent states representing a role in the script. Finally, some events and guards have to be chosen, which enable the agent to notice that a situation has arisen in which the execution of a certain script is appropriate, and to determine its role in the script.

Let us now continue our example from above. The double passing script consists of two roles, so two agent states have to be generated, which are shown in Figure 6. In this example synchronization is needed, because the ball has to be transported (kicked) from one player to another.

As the ball entering the kick range of a player is an observable event and does not involve the knowledge of internal states of the teammate, synchronization can easily be handled by using a change event or by putting a guard on the respective transition edges, which prevents the transition from firing unless the ball has become kickable for the recipient of the pass. Last but not least, the transition modelling the timeout of the script has been copied to the edges of both agent states, indicating that both agents terminate the double pass after 15 cycles as a failure. The determination of the roles the agents play in the script is handled by the possession of the ball.

3.4 Skill Level

With the use of the server commands (*dash, turn, kick, turn_neck,...*) alone, the ability of an agent to interact with its environment is very limited. Therefore there are procedures or functions that provide more sophisticated *skills* for an agent at the bottom level of almost all RoboCup simulation teams.

Dribbling, for example, is modeled as a sequence of controlled kicks and dashes. The *dribbling skill* of an agent is responsible for generating the sequence of *dash*, *turn* and *kick* commands needed to keep the ball under control while running to a particular position. Figure 7 shows a statechart describing a simplified dribbling skill.

From the viewpoint of agent modelling the skill level is very similar to the script level. Skills are modeled like single agent behaviors. There are, however, no analogues to multiagent scripts, as skills are abilities of *one* agent only.

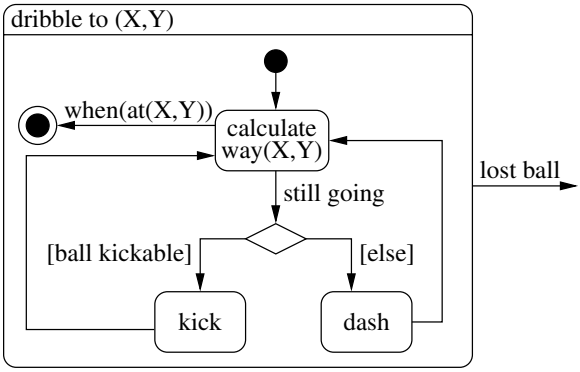

Fig. 7. A statechart modelling a simple dribbling skill.

4 StatEdit – A Statechart Editor

In order to assist an agent designer in modelling the behaviors of an agent with the help of the presented formalism the statechart editor StatEdit (Fig. 8) has been developed at the University of Koblenz [4]. With this editor the designer can easily create statecharts that specify the desired agent behaviors. Apart from the usual functionality provided by an editor for graphical elements, such as drawing, moving, erasing, or grouping objects, StatEdit offers a number of functions to support the modelling tasks of an agent developer. Two of them will be presented in greater detail, namely splitting (Sec. 4.1) and exporting (Sec. 4.2) of statecharts.

States and transitions are created and edited via pop up dialogues. The syntax of a statechart or a selection can automatically be checked, and a selected statechart can mechanically be beautified. In order to enhance the readability of a diagram, not all labels are shown at the respective elements. Transitions are only labeled with a unique identifier. The corresponding transition string is shown in a separate window when the pointer is over the transition (cf. Fig. 8). The same holds for the entry and exit actions of composite states.

4.1 Splitting Concurrent States

As we explained in Sec. 3.3 the specification of a multiagent script is done in two steps. First a concurrent script state is created. In a second step the script state is split into agent states and means of synchronization are specified.

StatEdit offers the functionality to split a concurrent state c between two regions, simply by clicking on the border between them. The result of this process are two (possibly concurrent) composite states c_1 and c_2, each corresponding to one (group) of the regions comprising the original state c. Transitions originating on the border of c are copied to the borders of both c_1 and c_2, while transitions from a substate are only moved with the corresponding state.

If a concurrent state is split between synchronized regions, synchronization is resolved with the help of events. Whenever this has to be done a unique signal event e^{ID}_{sync}

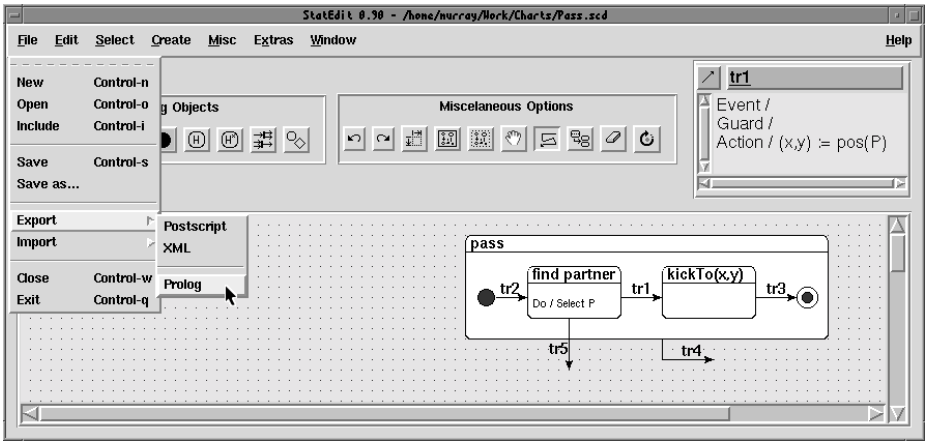

Fig. 8. The StatEdit main window with opened export menu. Prolog is selected as export format. The showinfo window on the right shows the transition string of tr_1.

is generated by StatEdit. The appropriate transitions in the resulting statecharts are labeled with transition strings, that either create e^{ID}_{sync} or are triggered by this event. Bounded sync states are resolved by putting additional guards on the transitions and incrementing resp. decrementing a global variable. Of course the user has the possibility to avoid the use of these generic events and resolve synchronization interactively.

4.2 Exporting a Statechart

Implementing a system that fulfills a given specification is a difficult and error prone process, especially for agent systems, which interact with their environment and have a certain degree of autonomy. In order to simplify this task StatEdit offers functions for exporting the created statechart to several other formats. This includes graphical formats for easy integration of figures into documentation, XML, and functions or function skeletons in different programming languages for (semi-)automatic code generation. Currently StatEdit supports conversion to EPS, XML, and a subset of Prolog, which can be interpreted by a statemachine built into our RoboCup team *RoboLog* [9, 10].

Exporting a statechart is done in two steps. First the internal representation of a statechart is converted to an XML structure which is then translated to the desired target format. This two step translation makes it easy to add further export modules, as there are already lots of XML based development tools available, which can easily be adapted for working as export functions.

5 Conclusion

This section closes the paper with some final remarks. First an overview over related work is briefly discussed. Then a short summary and an outlook to some future work will be presented.

Bergenti and Poggi [3] state that agent oriented software engineering adds another level of abstraction to the process of modelling software systems. This level, the *agent level*, treats agents as atomic units and models multiagent systems as interactions between agents. They present four agent oriented diagrams using standard UML notation. With these diagrams ontologies, agent classes and protocols are described. With this approach only aspects of the interactions among agents can be modeled. The agents themselves are treated as atomic entities. In contrast to this, our approach allows for the modelling of interactions between agents as well as describing the behavior of a single agent, i.e. the internals of an agent with only one formalism.

In [12] Odell et. al propose a number of extensions to UML for modelling multiagent systems or interactions between agents under the name of *AUML (Agent UML)*. A layered approach to specifying interaction protocols for agents is presented. At the top agent interaction is specified in different levels of detail with (extended) *sequence diagrams* and *collaboration diagrams*. *Activity diagrams* and *statecharts* are also used on this levels to emphasize certain aspects of the specification. On the lowest level intra-agent processes are specified with extensions of *activity diagrams* and *statecharts*. This approach is continued in [2] with the introduction of *protocol diagrams* into AUML, which extend the semantics of agent messages and improve inter-agent protocols. In contrast to the proposed Agent UML, our approach needs only one formalism to describe both the inter-agent and intra-agent behaviors in a multiagent system. In addition no extensions to the existing formalisms of UML have to be made.

In [7] an approach to agent specification and modelling of multiagent-systems based on object oriented formalisms is presented. Kinny and Georgeff use several object oriented modelling techniques, e.g. *class diagrams* and *statecharts*, for specifying agents. The plans that an agent may apply to reach a certain goal are described by *plan diagrams* which are based on statecharts. In contrast to the method presented in this paper plan diagrams are not used to express multiagent plans or explicit cooperation among agents. Multiagent systems are rather modeled by a class diagram, which describes the different classes of agents and the relationships among them. Object oriented mechanisms like inheritance are then used to distribute attributes between agent classes.

5.1 Summary and Outlook

We presented a graphical formalism for the layered specification of agent behaviors and multiagent systems. The formalism is based on UML statechart diagrams, which are used for modelling the behavior of systems in the object oriented software development paradigm. The approach allows for the explicit modelling of cooperation between two or more agents to achieve a common goal. From the specification of such a behavior the individual roles of the participating agents can be derived in a straightforward manner. To support a developer using our modelling technique the statechart editor StatEdit has been developed. With this editor statecharts can be created and exported to a variety of formats for further processing.

Future work includes applying the presented methods to other applications as well. First steps in this direction have already been taken [1]. The statechart editor StatEdit will be extended by further export modules. In the near future modules for integrating StatEdit with Golog [6] and the double pass architecture [5] will be implemented.

References

1. Toshiaki Arai and Frieder Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In Cristiano Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the first international joint conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*. ACM Press, 2002.
2. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In Paolo Cinacarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, 2001.
3. Frederico Bergenti and Agostino Poggi. Exploiting UML in the Design of Multi-Agent Systems. In *Proceedings of Engineering Societies in the Agents' World*, pages 96–103, 2000.
4. Michael C. Bruhn. StatEdit – a state chart editor. Diplomarbeit, Universität Koblenz-Landau, Campus Koblenz, Germany, 2003. to appear (German only).
5. Hans-Dieter Burkhard. Mental models for robot control. In *Advances in Plan-Based Control of Robotic Agents*, volume 2466 of *Lecture Notes in Artificial Intelligence*. Springer, Heidelberg, 2002. Postproceedings of Dagstuhl Seminar 01431 (Oct 21-26, 2001).
6. Frank Dylla, Alexander Ferrein, and Gerhard Lakemeyer. Acting and deliberating using GOLOG in robotic soccer — a hybrid architecture. In *Proceedings of the 2002 Workshop on "Cognitive Agents"*, September 2002. (Workshop during KI 2002, Germany).
7. David Kinny and Michael Georgeff. Modelling and design of multi-agent systems. In J. P. Müller, Michael Wooldridge, and Nicholas R. Jennings, editors, *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, volume 1193 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag: Heidelberg, Germany, 1997.
8. Johan Lilius and Iván Porres Paltor. The semantics of UML state machines. Technical Report 273, TUCS - Turku Centre for Computer Science, 1999.
9. Jan Murray, Oliver Obst, and Frieder Stolzenburg. RoboLog Koblenz 2001. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Artificial Intelligence*. Springer, Berlin, Heidelberg, New York, 2002. Team description.
10. Jan Murray, Oliver Obst, and Frieder Stolzenburg. RoboLog Koblenz 2002 – short team description. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup 2002: Robot Soccer World Cup VI*, Fukuoka, Japan, 2002. Pre-Proceedings.
11. Object Management Group, Inc. *OMG Unified Modeling Language Specification*, September 2001. Version 1.4.
12. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for Agents. In *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence (AOIS Worshop at AAAI 2000)*, 2000.
13. Mikhail Prokopenko. Situated reasoning in multi-agent systems. Working Notes of the AAAI-99 Spring Symposium on Hybrid Systems and AI., 1999.
14. Frieder Stolzenburg. Reasoning about cognitive robotics systems. In Reinhard Moratz and Bernhard Nebel, editors, *Themenkolloquium Kognitive Robotik und Raumrepräsentation des DFG-Schwerpunktprogramms Raumkognition*, Hamburg, 2001.
15. Frieder Stolzenburg, Oliver Obst, Jan Murray, and Björn Bremer. Spatial agents implemented in a logical expressible language. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, volume 1856 of *Lecture Notes in Artificial Intelligence*. Springer, 2000.