Specifying Agents with UML in Robotic Soccer

Jan Murray

10/2001

Specifying Agents with UML in Robotic Soccer[*]

Jan Murray
Institut für Informatik
Universität Koblenz-Landau
Rheinau 1
D-56075 Koblenz
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.

In this paper we propose an approach to specifying agents with the help of UML statecharts. Agents are specified on different levels of abstraction. In addition a method for specifying multiagent plans with explicit cooperation is shown.

As an example domain we chose robotic soccer, which lays the basis of the annual RoboCup competitions. Robotic soccer is an ideal testbed for research in the fields of robotics and multiagent systems. In the RoboCup Simulation League the research focus is laid on agents and multiagent systems, and we will demonstrate our approach by using examples from this domain.

Keywords: Multiagent Systems, Unified Modeling Language (UML), Specification, RoboCup, Robotic Soccer

1 Introduction

The use of agent technologies and multiagent systems has gained entrance into almost all branches of computer science. With the propagation of this technology the need for standards and design techniques has arisen.

[*]This research is supported by the grant *Fu 263/8-1* from the German research foundation *DFG*.

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 has to be easy to understand and use. In addition to that it would be desirable, if the application of formal methods is supported by the specification mechanism. This calls for a formal semantics of the specification formalism.

In this paper we present an approach to agent design that is based on UML statecharts [8]. With this approach it is possible to specify not only behaviors for single agents on different levels of abstraction, but multiagent plans as well.

The use of UML statecharts for this purpose meets the requirements stated above. 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 modeling language is already widely accepted. Although the specification of UML statecharts [8] 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 [4, 12].

Last but not least, the transformation of the agent specification into running code is very straightforward. As the limited space in this paper does not allow for a discussion of implementation issues, the interested reader is referred to [5].

The choice of soccer as an example domain comes quite naturally. As soccer is a team sport, it provides a test bed for multiagent systems. The high dynamics of the game make fast reasoning and quick reactions necessary. The RoboCup initiative, finally, provides a simulation environment and the annual benchmarks at the RoboCup competitions.

1.1 Overview of the Paper

The paper is organized as follows. Section 2 provides a short introduction to the RoboCup initiative. The Simulation League, which we chose as example domain, is presented and the simulator used in this league – the soccer server – is described in more detail to show the environment the agents act in (Section 2.1).

In Section 3 we shortly summarize the UML statechart formalism with the focus laid on those parts of the formalism that will be employed for the specification of agents.

Section 4 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 4.3).

Section 5 finally concludes the paper with some experimental results

(Section 5.1), an overview over related work (Section 5.2) and some final remarks (Section 5.3).

2 RoboCup

RoboCup is an ongoing initiative to bring together researchers from the fields of artificial intelligence, robotics and multiagent systems. The domain that was chosen as a testbed and for motivation is *soccer*. In several leagues real and simulated robots play each other at annual competitions. In addition to the competitions RoboCup includes demonstrations, technical sessions and teaching.

2.1 Simulation League

In the RoboCup Simulation League the research focus is laid on topics related to agents and multiagent systems. Two teams – each consisting of 11 autonomous agents – connect to a simulator, the so called *soccer server*[7], which simulates a soccer match.

The soccer server uses a *discrete* time model. Every 100 ms time is advanced by one *simulation step* (or *simstep* for short). In each step the client can execute actions by sending the corresponding commands to the server. There are two classes of commands. The *major commands* are used to accelerate, turn or move the client and to kick the ball. Only one of those commands may be sent in each cycle. In addition there are several *minor commands*, which are, for example, responsible for controlling the vision sensors or communication. Several minor commands may be sent to the server in one simulation step.

Simulated sensor data is sent to a client at intervals that depend on the current settings of the sensors (view range, quality). So sensing and acting is asynchronous in this setting. *Communication* between agents is only possible via the soccer server. An agent may send a message up to a fixed length to the server, which is broadcast to all agents within a certain distance. Thus the available communication channels are insecure and suffer from a low bandwidth.

The discrete time model and the fact that commands are executed by the soccer server at the end of a simulation step suggest the execution cycle that is shown in Figure 1 for an agent control program.

At the beginning of a simulation step the agent updates its world model, either by integrating new sensor information into its current belief of the world, or by estimating the effects of its last action if no new sensor data is available. Based on the updated world model the agent then selects a major action to execute and perhaps one or more of the minor actions. At the end

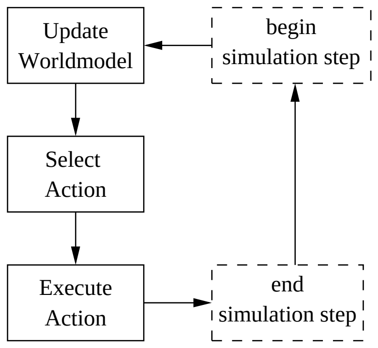

Figure 1: Execution cycle of the agent control program.

of the cycle the selected actions are sent to the soccer server for execution. In the next simstep the whole execution cycle of the agent control program starts anew. In Figure 1 this cycle is shown. The dashed boxes indicate the beginning and end of a simulation cycle and are not a part of the agent control program itself.

3 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 Modeling Language (UML) [8] *state machines* are used to model behavioral aspects of systems. State machines are normally described in a graphical notation like state transition diagrams or, as is the case with UML, *statecharts*.

Statecharts are an extension of classical state transition diagrams (STD's). Statecharts, like STD's, 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. In contrast to classical STD's statecharts possess depth, i.e. states in a statechart can contain other states or even whole state machines. So a statechart is a *hierarchical* STD.

In the following we will briefly summarize those parts of the UML statechart formalism that are employed for the design of agents.

States and Transitions

In UML a *state* 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 a statechart a state is represented as a box with rounded corners.

The state a system is in can be changed in reaction to external events. Such a change of state is called a *transition*. A *transition string* is used to specify the behavior of a transition. A transition string is a tuple $\mathcal{T} = (e, c, a) \in (E \times C \times A)$, where E is the set of (external) events, C is the set of boolean expressions over a domain and A is the set of possible actions that can be taken. A transition then can be defined as a tuple $t = (s_1, \mathcal{T}, s_2)$, where s_1, s_2 denote arbitrary states and $\mathcal{T}$ is a transition string of the form (e, c, a). 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". In a statechart diagram a transition is shown as a directed edge from s_1 to s_2, which is labeled with $\mathcal{T}$ in the form $e[c]/a$.

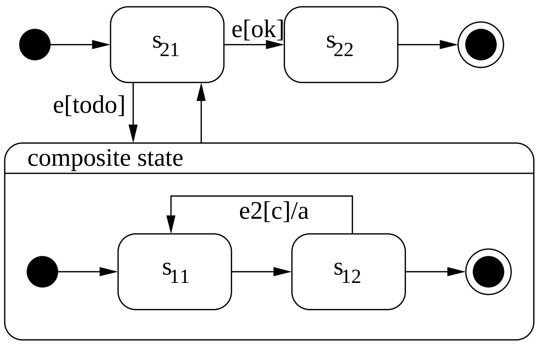

Figure 2: A statechart.

More about States

In the UML statechart formalism different types of states exist. As a statechart is hierarchical in nature, the UML distinguishes between three major classes of states two of which are shown in Figure 2.

Simple states are atomic in the sense that they do not possess any internal structure. Nevertheless it is possible to assign some kind of behavior to a simple state by defining internal transitions.

Composite states are states that can be further decomposed. They contain internal submachines which describe the activity associated with the composite state.

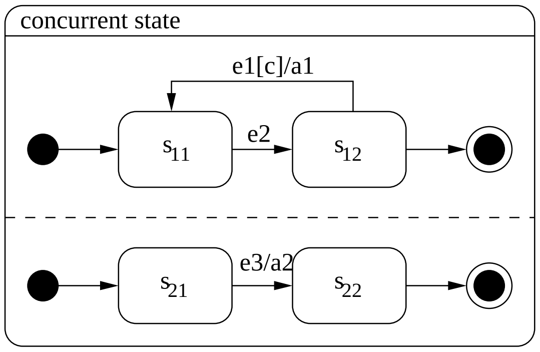

Figure 3: A concurrent state with two regions.

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. An example is shown in Figure 3.

Apart from those states a variety of other types of states for different purposes exist, although many of them are just syntactic sugar which help to keep statecharts readable.

Synchronization

Sometimes it is necessary to synchronize the concurrent activities in a statechart. 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 in this scenario. The statechart in Figure 4 shows a scenario in which parts of type **B** are produced by one agent and used by another agent to assemble components **ABC**.

The synchronization in the concurrent state is handled by 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 by a positive integer giving an upper bound of the number of times the incoming and outgoing transitions have fired or with an asterisk, if there is no such upper bound. In our example the upper bound is 6, so only six parts of type **B** may be produced, before one has to be consumed.

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 like a simple state.

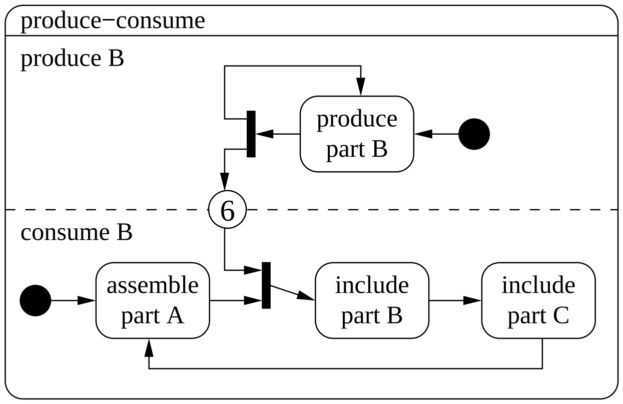

Figure 4: A producer-consumer scenario modeled by a statechart.

History States

One special type of pseudostate is the *history state*, which is drawn as a circle labeled with “H*”. With the help of such a state a composite state can be made to remember its last active substate.

The purpose of a history state is to model some kind of subroutine calls or interrupt. If a statechart, that has previously been interrupted by a transition from its enclosing composite state, is re-entered through a history state, not its initial state becomes active but the state that has been active just before the statechart has been left the last time. In the context of agent design this corresponds nicely to the continuation of a previously interrupted behavior by the agent.

Last but not least, a statechart may contain two special (pseudo-)states, namely an *initial* and a *final* state. The initial state points to a designated start state of a state machine. Any state machine is entered at its initial state unless it is entered by a transition explicitly leading to another state of the machine. When a composite state is entered, the initial states of all directly contained state machines are entered, as well. The final state of a state machine is entered, if the activity modeled by this machine is finished. In this case as special event, a so called *completion event*.

4 Designing Agents with UML

The behaviors of an agent can be seen as a sequence of states the agent is in. Each state may correspond to an activity the agent executes or indicate that the 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 topdown 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.

But there is an important difference between the design of a behavior of a single agent 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 propose 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.

- 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 expands a higher level. The result of the design process is a layered specification of an agent. This is shown by Figure 5.

Throughout the rest of the section we will describe, how UML statecharts are employed in the specification of agents with the above approach. Section 4.1 explains the high level specification of an agent, while the subsequent sections deal with the design of single agent behaviors (Section 4.2) and multiagent plans (Section 4.3). The skill level will only be addressed very

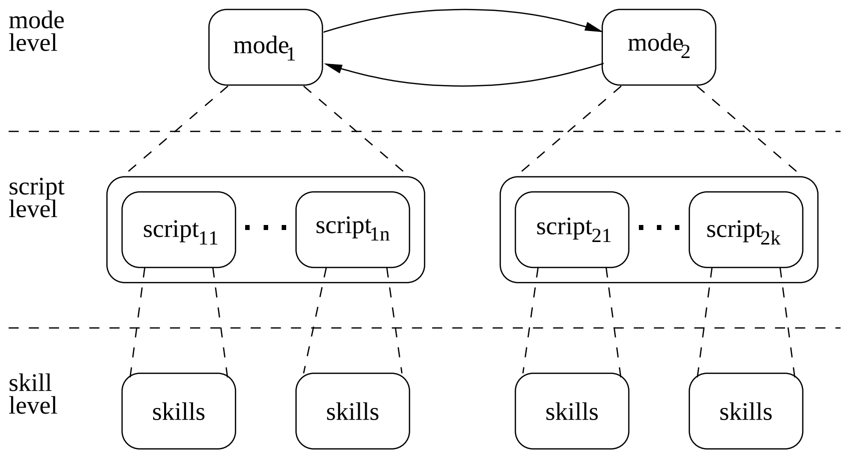

Figure 5: Layered Architecture of an Agent.

briefly in Section 4.4, because this level does not raise any additional interesting topics in the design of an agent. The last part of this section deals with some additional uses of UML statecharts for agent specification (Section 4.5).

4.1 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 agents 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 behaviors. Whenever the agent's team controls the ball or is assigned a *free kick, kick in,* etc. 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, usually by positioning the players at their initial positions. This is a recurring situation with a fixed activity which can be modeled by a mode, too.

Figure 6 shows the resulting modechart. As the soccer teams line up on the field before the game is started, the setup state was chosen as the initial state. When the game is over, the agent enters its final state and ceases its activities.

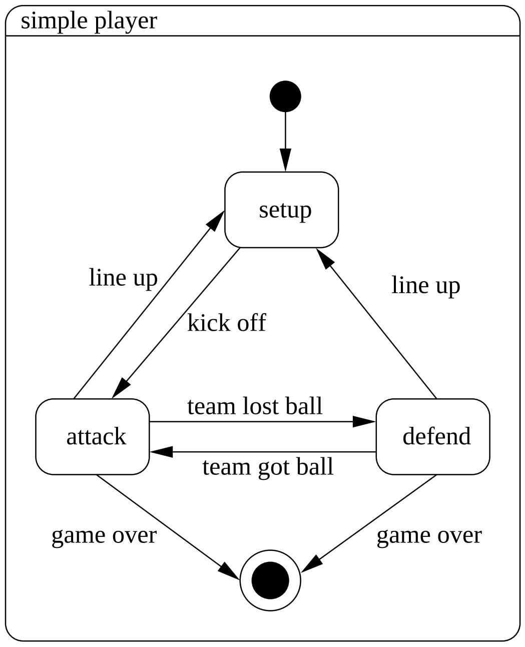

Figure 6: Modechart for a simple player.

4.2 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 all scripts, depending on the situations that can arise in the particular mode. If the agent is in the defend mode, it makes no sense for him to try a script that calls for ball possession, as the agent enters the defend mode only if the *other* team controls the ball. 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 [13] 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, more precisely by the statechart included in the 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 can easily be modeled by transitions originating from the edge of the composite state.

Consider the example of a *passing script* shown in Figure 7. The agent selects a teammate to kick the ball to, gets the coordinates to the teammate and finally kicks the ball to those coordinates. If the agent cannot find a teammate to pass to or loses the ball, the script is aborted.

The agent can lose the ball during either activity modeled by a substate, 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 modeling this activity.

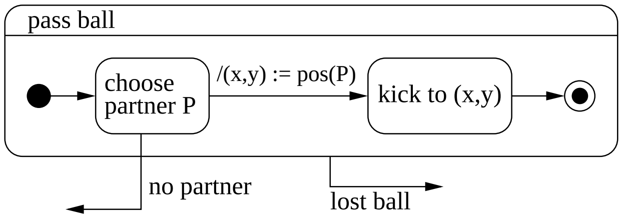

Figure 7: A script for passing.

4.3 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? As we stated above, 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 – can easily be modeled with the help of concurrent states. A multiagent script is specified as a concurrent state with a region for each role (or class of roles) that 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 the situation in Figure 8, where 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

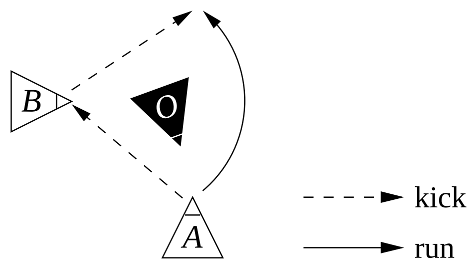

Figure 8: Double passing scenario.

B. The concurrent state modeling this script is shown by Figure 9. The simple states in the script correspond to the activities of the agents, which were printed in italics in our description above. 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. More transitions for modeling interruptions of the double pass, e.g. by loss of the ball, can easily be added to the edge of the double passing state.

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 that the agent control program will be based on 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 basically 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. This process is quite straightforward, since most of the time the agents’ behaviors are quite independent of each other.

The only spots that require special attention are the synch states. As we

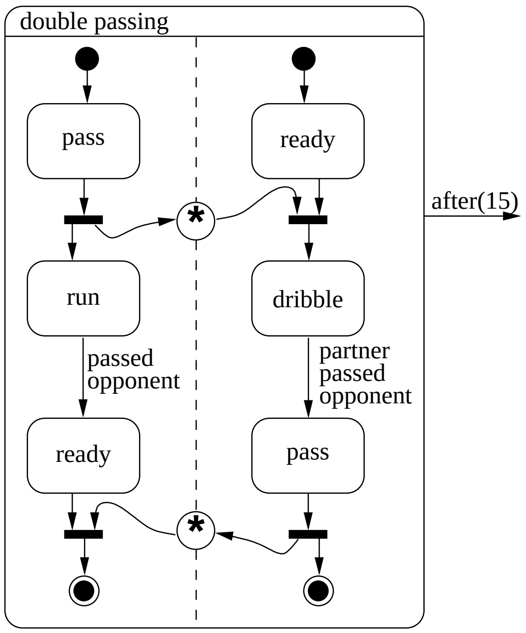

Figure 9: A multiagent script for double passing.

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 decide 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 modeling interruptions or errors are just taken from the script state. If such a transition starts on the edge of the composite state, it has to be copied to the edges 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 10. In this example synchronization is needed, because the ball has to be transported (kicked) from one player to another. The recipient can only continue the execution of its behavior when it controls the ball, i.e. the ball can be kicked by the agent.

As this is an observable event and does not involve the knowledge of internal states of the teammate, synchronization can easily be handled 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 modeling 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.

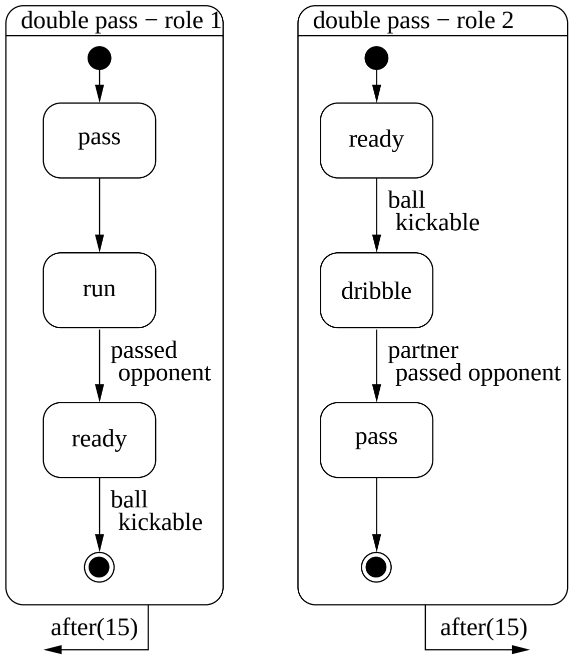

Figure 10: The derived agent states for double passing.

4.4 Skill Level

As we said before, the soccer server provides two different kinds of actions, namely *major* and *minor* actions. These actions are used by the agent to interact with its environment. In each simulation cycle at most one of the major actions *(dash, turn, kick, catch* and *move)* can be executed. In addition a number of minor actions can be sent to the soccer server in the same step. The latter include e.g. *turn_neck, say* and *change_view*.

With the use of the server commands alone, the ability of an agent to interact with its environment is very limited. Therefore there are procedures or functions that provide more sophisticated abilities for an agent at the bottom level of almost all RoboCup teams. Those abilities, which sometimes rely heavily on a precise knowledge of the way the world is modeled by the soccer server, are called *skills*. Their quality is very important for the success of a team. Normally skills cannot be transferred from players of one team to members of another, because the skills of an agent are usually very closely entwined with its world model.

Let us clarify this with the help of an example. In real soccer a player is said to be dribbling if he runs and takes the ball with him with a series of controlled kicks.

In the RoboCup scenario dribbling is modeled as a sequence of kicks and dashes as well. 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 11 shows a statechart describing a simplified dribbling skill. The transition labeled "lost ball" leads to an error handling routine.

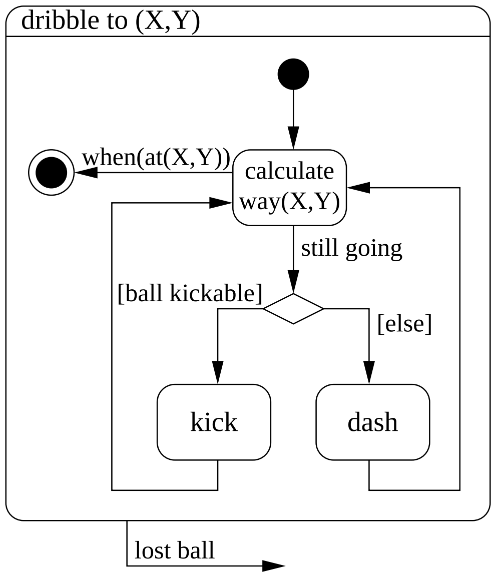

Figure 11: Simple dribbling skill.

But from the viewpoint of agent modeling the skill level adds nothing new, as the techniques used for modeling skills are pretty much the same as for modeling single agent behaviors.

Like scripts, skills are modeled as sequences of states, that usually correspond to (sub-)activities executed by the agent. If an activity is finished or as a response to an external event, the agent may change its state or leave the statechart corresponding to the whole activity. There are, however, no analogues to *multiagent scripts*, as skills are abilities of *one* agent only.

4.5 Additional Uses of Statecharts

In this section we show two more uses for UML statecharts in an agent specification.

Some events in the world have such a high priority, that the agent has to react to the occurrence of such an event immediately, regardless of what it is currently doing. In the soccer domain such events could for example occur, if the agent does not know its position on the field or loses track of the ball. Both events are so important, that the agent has to react by interrupting its current behavior to localize itself or search for the ball. After this has been done, the interrupted behavior may be continued if it is still appropriate.

In UML such a scenario is modeled with the help of a *history state*. The statechart modeling the primary behavior of an agent, i.e. the modes and scripts, is left in response to a high priority event. The (composite) state corresponding the activity triggered by the event is entered. When it is completed, the primary statechart is re-entered via a history state, so the interrupted behavior is automatically resumed.

Sometimes an agent has to perform a simple recurring or continuous task in parallel to its main activities. Those activities are independent from the main processes modeled by the modechart, yet they are executed in parallel. Therefore it is appropriate to model such additional concurrent activities in a concurrent state in the agent specification. Such a state contains several regions. One of them contains the modechart describing the agent's main activities. The other model subordinate, yet concurrent tasks of the agent.

In the RoboCup scenario such a subordinate activity could be the announcement of the agent's world model. Every agent announces a part of its world model at certain intervals in order to enhance its teammates' belief about the world. By this a primitive form of sensor fusion is established.

Figure 12 gives a schematic presentation of the specification of an agent with UML statecharts. The primary behavior of the agent is specified in the region called *mode chart*. Additional behaviors may be specified in the regions that are combined in the regions entitled *concurrent activities*. Interruptions are modeled with the help of a history state.

5 Conclusion

This section closes the paper with some concluding remarks. We will give some experimental results and present an overview of related work. Finally a summary and outlook are presented.

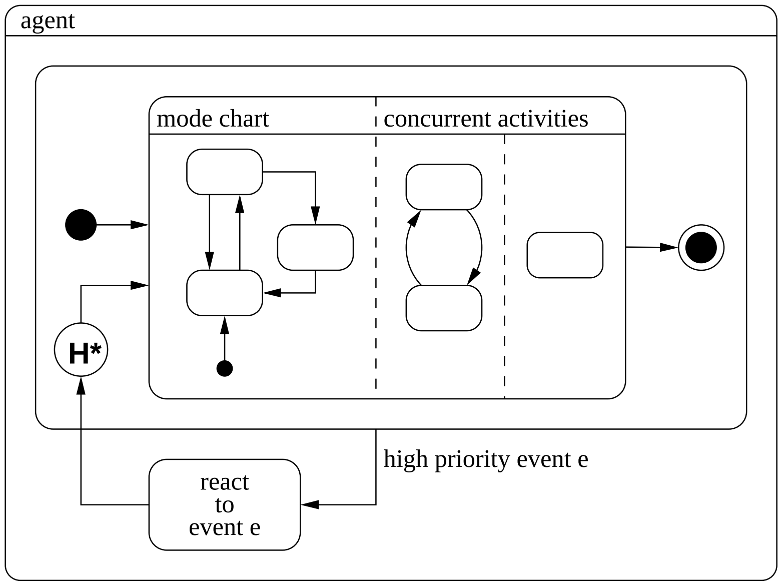

Figure 12: Schematic specification of an agent.

5.1 Experimental Results

We implemented two different teams for the RoboCup simulation league with this approach. The implementation of the UTOPiA team [5] strictly followed the specification of modes and scripts with UML statecharts. The actual implementation was done in SWI Prolog [14] together with the RoboLog API [10].

The RoboLog Koblenz 2001 team [6], which participated in the RoboCup 2001 World Cup in Seattle, is based upon the UTOPiA team, but a state machine has explicitly been built into the team, so that no transformations from the specification to the implementation of the scripts are necessary.

Both teams proved to be competitive to teams that finished in the upper third in the RoboCup world championships 2000 and 2001.

5.2 Related Work

Designing Multiagent Systems with UML

Bergenti and Poggi [3]state that agent oriented software engineering adds another level of abstraction to the process of modeling a software system. This level, the *agent level*, treats agents as atomic units and models multiagent systems as interactions between agents. They present four agent oriented diagrams that are used to describe various aspects of a multiagent system on the agent level. All diagrams use standard UML notation. *Ontology dia-*

grams are special forms of *class diagrams* that represent classes of entities in an ontology and the relations among them, as well as support the inter-agent communication. *Architecture diagrams* employ class diagrams to model heterogeneous multiagent systems in terms of different *agent classes* and the relations among them. UML's *Deployment diagrams* can be employed to model the instantiation of a class of multiagent systems specified by an architecture diagram. *Protocol diagrams* – based on *collaboration diagrams* – and *role diagrams* – special *classes* – are closely connected. Protocol diagrams are used to model the permitted interactions between agents. A role diagram models the different roles that can be played by agents in such an interaction.

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 modeling of interactions between agents as well as describing the behavior of a single agent, i.e. the internals of an agent. In addition to that only one formalism is used for both tasks.

Agent UML (AUML)

In [11] a number of extensions to UML for modeling multiagent systems or interactions between agents are proposed under the name of *AUML (Agent UML)*. A layered approach to specifying interaction protocols for agents is presented. On each of the three levels UML or an extension thereof can be employed.

At the top level a pattern for an agent interaction protocol is specified as a whole with *sequence diagrams*. For a particular situation such a pattern can then be instantiated with the concrete agents that participate and the roles they play. The second level models the interactions among agents in greater detail. To this end a variety of UML diagrams can be employed. Still mainly (extended) *sequence diagrams* – or *collaboration diagram* for readability – are used to describe the agent interaction protocol in more detail. *Activity diagrams* and *statecharts* are also used on this level to emphasize certain aspects of the specification. On the third and lowest level intra-agent processes implementing an interaction protocol are specified with extensions of *activity diagrams* and *statecharts*. Further extensions to UML, e.g. the inclusion of role markers, are meant to improve the readability the diagrams that model interactions between several agents.

[2] continues this approach 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.

UML and Utility Functions

In [9] the approach presented in this work is combined with the use of utility functions. The usual transition triggers, i.e. events and guards, are replaced with preference functions which return the preference of the agent for executing an action given its current belief and the expected utility of the action. By this a more flexible decision procedure should be realized.

5.3 Summary and Outlook

In this paper we presented an approach to the specification of agents and multiagent systems with the help of UML statecharts. With this approach it is possible to specify both intra- and inter-agent aspects with just one formalism. The publicity of UML in software engineering helps in understanding such a specification and supports the acceptance of the approach. We have demonstrated the feasibility of the procedure with examples from the domain of robotic soccer and added some experimental results.

Future work includes investigating if the presented methods can be applied to other applications as well. First steps in this direction have already been taken [1].

In addition to that the design process has to be put on a rigidly formal basis. This will enable us to apply formal methods to the agent design. Reasoning about interesting properties of the specification and verification thereof will be possible.

References

[1] Toshiaki Arai and Frieder Stolzenburg. Multiagent systems specification by uml statecharts aiming at intelligent manufacturing. Submitted, 2001.

[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] Johan Lilius and Iván Porres Paltor. The semantics of UML state machines. Technical Report 273, TUCS - Turku Centre for Computer Science, 1999.

[5] Jan Murray. Soccer Agents Think in UML. Diplomarbeit (D 610), Universität Koblenz-Landau, 2001.

[6] Jan Murray, Oliver Obst, and Frieder Stolzenburg. RoboLog Koblenz 2001. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer WorldCup V*. Springer, 2001. To appear.

[7] Itsuki Noda. Soccer server: a simulator for RoboCup. In *JSAI AI-Symposium*, 1995.

[8] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, September 2001. Version 1.4.

[9] Oliver Obst. Specifying rational agents with statecharts and utility functions. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-01: Robot Soccer WorldCup V*. Springer, 2001. To appear.

[10] Oliver Obst, Jan Murray, Marco Dettori, and Frieder Stolzenburg. *The RoboLog Soccer Server Interface*. Universität Koblenz-Landau, 2000.

[11] 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.

[12] 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.

[13] 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.

[14] Jan Wielemaker. *SWI-Prolog 3.3 Reference Manual*. University of Amsterdam, The Netherlands, June 2000.