Towards a Logical Approach for Soccer Agents Engineering

Jan Murray, Oliver Obst, Frieder Stolzenburg

Universität Koblenz-Landau, Fachbereich Informatik
Rheinau 1, D–56075 Koblenz, GERMANY
{murray,fruit,stolzen}@uni-koblenz.de

Abstract. Building agents for a scenario such as the RoboCup simulation league requires not only methodologies for implementing high-level complex behavior, but also the careful and efficient programming of low-level facilities like ball interception. With this hypothesis in mind, the development of RoboLog Koblenz has been continued. As before, the focus is laid on the declarativity of the approach. This means, agents are implemented in a logic- and rule-based manner in the high-level and flexible logic programming language Prolog. Logic is used as a control language for deciding how an agent should behave in a situation where there possibly is more than one choice.
In order to describe the more procedural aspects of the agent's behavior, we employ state machines, which are represented by statecharts. Because of this, the script language for modeling multi-agent behavior in [8] has been revised, such that we are now able to specify plans with iterative parts and also reactive behavior, which is triggered by external events. In summary, multi-agent behavior can be described in a script language, where procedural aspects are specified by statecharts and declarative aspects by logical rules (in decision trees). Multi-agent scripts are implemented in Prolog. The RoboLog kernel is written in C++ and makes now use of the low-level skills of the CMUnited-99 simulator team.

1 Introduction and Overview

The RoboLog Koblenz system is based on a multi-layered architecture, as introduced in [8]. In the following, we will consider each layer in some detail and explain its (revised) functionality. Several aspects of some layers will be further elaborated in the subsequent sections. A picture of the overall architecture is given in Figure 1. The team that participated in the RoboCup-99 and its theoretical background are described in [5, 8]. See also the team description *RoboLog Koblenz 2000* in this volume.

1.1 A Layered Architecture

The RoboLog system allows us to program *complex actions* and *cooperation*. This is settled in the higher layers of our system architecture (look at Figure 1). For this, a script language has been developed, which is expressible enough to specify plans explicitly, where several agents may work in parallel, trying to achieve one goal together in collaboration. However, one problem with this approach was, that agents may have

P. Stone, T. Balch, and G. Kraetzschmar (Eds.): RoboCup 2000, LNAI 2019, pp. 199-208, 2001.

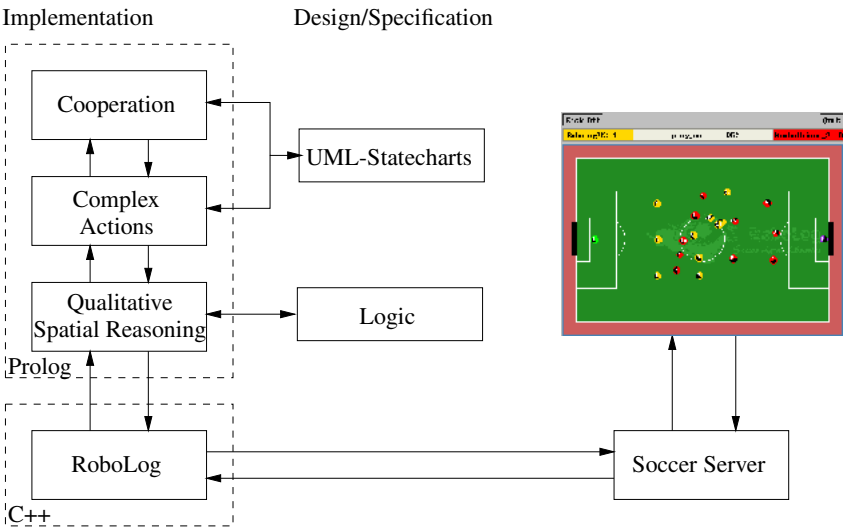

Fig. 1. Overall Architecture of the RoboLog system.

difficulties in reacting to external events or when parts of the intended plan fail. Therefore, we revised the script language and the execution model, such that we can now deal with such events by employing state machines. See Sections 2 and 3.

Logic programming enables us to program specific behaviors and strategies for RoboCup agents. Some of the currently implemented strategies will be explained in the beginning of Section 4. It turns out that besides explicit programming of teamwork, emergent cooperation is also quite helpful. Certainly, this high-level programming of agents requires the careful implementation of low-level skills as well as the possibility of abstracting quantitative sensor data onto a qualitative level, such that more general *qualitative reasoning* becomes feasible.

Last but not least, the basic layer —the *RoboLog kernel*—, implemented in C++, processes the sensor information of an agent. We incorporated a subset of the low-level skills of the CMUnited-99 simulator team [9] into it. Therefore, a variety of skills is available, that can be used to program higher abilities in Prolog. Essentially, this layer provides an interface between the Soccer Server [2] and (now) SWI-Prolog [10]. We will describe our experiences with the new RoboLog kernel in Section 4.3.

1.2 Logical Control and Procedural Behavior

As said earlier, we lay the focus on the declarativity of the approach for specifying multi-agent behavior. We believe that not all aspects of an agent can be implemented by providing some primitive actions plus possibly hierarchically structured rules (for reactive behavior) and learning techniques for automatically improving the multi-agent system. It should always be possible for a programmer of multi-agent systems to manipulate the intended behavior in an explicit manner. Therefore, we provide logical rules for control purposes and statecharts for the procedural aspects of multi-agent systems in

our approach. Of course, this can be complemented with other more implicit techniques for agent control, e.g. from machine learning or artificial neural networks.

2 Execution Model for Flexible Multi-Agent Scripts

2.1 Predefined Multi-Agent Plans

In [8], a script language for programming multi-agent systems has been introduced. It formalizes some aspects present in belief-desire-intention architectures [7] and combines them with logic programming by means of rules of the following form: If an agent has a certain desire, i.e. aims at a certain goal, and the agent believes that the preconditions for achieving it are satisfied, then the intended plan for the given desire is executed. In general, intended (multi-agent) plans are arbitrary acyclic graphs, where each path corresponds to a sequence of actions for one of the agents that is involved in the whole multi-agent plan. For more details, the reader is referred to [8].

One of the advantages of this approach was, that it enabled programmers to specify multi-agent behavior in an explicit manner by means of simple (first-order) logic programming. But the rigidity of this model proved to be one of its major disadvantages. Once a plan had been selected for execution, it was difficult to interrupt or stop it, when external events or other reasons required this. Only time-limits were applied for the cancellation of scheduled actions. In order to overcome this problem, we changed our script language and also the execution model.

2.2 Interruptible Multi-Agent Behavior

Figure 2 shows the revised execution model. The agent now has internal states which represent its role in the script currently executed, its position within this script, and (possibly) the state of the skill the agent is currently executing. The agent knows e.g. that it is dribbling and has kicked the ball in the last cycle. Therefore it will probably send a *dash* command to the server.

At the beginning of an execution cycle, the RoboLog kernel transfers control to Prolog. First the agent's internal states are tested. If the current world state allows us to continue the running script, the agent updates the internal states and selects the next action for execution. It then returns control to the RoboLog kernel, which in turn sends the command to the Soccer Server and updates the agent's world model. If, however, the continuation of the current script is not appropriate in the present world state, the agent selects a new script, which may be a default script, for execution. This choice is based on external events (e.g. change of play mode) and the agent's knowledge of the world. The internal state is updated to record the fact that the agent starts the execution of a new script. Afterwards, the next action is selected, and the control is returned to the RoboLog kernel.

Multi-agent scripts can still be specified similar to the way described in [8], allowing several agents with different roles to interact in one cooperative action (see also Section 3.2). A clear advantage of the new execution model is its increased flexibility together with the persistence of the behaviors. The (multi-agent) scripts grant a certain

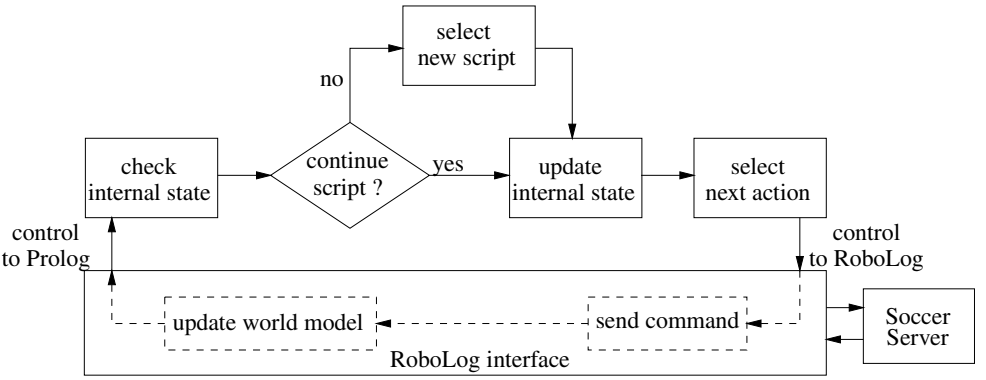

Fig. 2. The execution model.

continuity in the agent's behavior, while the repeated evaluation of the situation at the beginning of each cycle makes quick reactions to changes in the environment possible. But how can the overall behavior of an agent be described, including the transitions from one state to another? The answer will be given in the next section.

3 The Specification of Multi-Agent Scripts by Statecharts

The specification and programming of multi-agent behavior should be as concise as possible. It is desirable that the description language for multi-agent behavior is very flexible, so that the understanding of an agent's behavior and the modification and maintenance of existing code is easily feasible. The best would be, if a specification of multi-agent system can be read as an executable agent program, or at least the translation into running code can be done automatically. Then, no additional step for code verification is necessary.

The specification of multi-agent behavior requires the modeling of activities. In the case of complex actions, i.e. a sequence of actions, it must be possible to model different states an agent can be in. Therefore, it seems to be a good idea to adopt the state machine formalism for our purpose. A well-known means in this context are statecharts, which have a clean semantics—namely via the theory of finite automata—, and they are widely accepted as a means for specifying software. Statecharts are part of the unified modeling language (UML) [6].

3.1 Making Use of Statecharts

The main ingredients of statecharts are states and transitions. In order to make the understanding of statecharts for specifying multi-agent behavior in the RoboCup as easy as possible, we make the assumption, that each transition (including self-transitions) corresponds to exactly one simulator cycle step. What happens along one transition, is described in our execution model (see Section 2). In statecharts, transitions are annotated with events, further conditions and actions.

The most simple agent is just a reactive agent with only one state, but many transitions, that are self-loops. The basic structure of a reactive agent is sketched in Figure 3. There, we identify *events* with the play modes in the RoboCup. The (guard) *conditions*, written in square brackets, may request the actual world model, which is represented in logic, it is the belief of the agent. Only one *action* per transition is expressible by statecharts. But this corresponds quite well to the fact that the Soccer Server permits only one action per simulation step (with few exceptions, namely *change_view*, *say* and *turn_neck*).

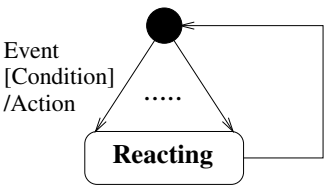

Fig. 3. A purely reactive agent.

More flexible agents, of course, should not only be reactive. They have internal states. Then it becomes possible to express complex actions that require the performance of a sequence of primitive actions, which are provided by the RoboLog kernel. Thus, e.g. before the kick-off, a player should move to its start position only once. Afterwards, the player could scan the field, until the game is actually started. Depending on the situation the agents are in, different more complex scripts can be initiated.

3.2 Multi-Agent Scripts with Iteration

Complex decision processes can be expressed by choice-points, that are pseudo-states, drawn as bullets. This is shown on the right in Figure 4, where one possible action is to perform double passing. This could also be viewed by means of decision trees, which are employed in [1] (see also Section 5.1). Double passing is itself a complex behavior, represented by a composite state. There are two actors in a double passing situation. At first, actor 1 passes the ball to actor 2, then actor 1 runs towards the opponent goal, and expects a pass from actor 2. Look at Figure 5, where this is sketched. For the sake of readability, some labels are omitted.

In principle, double passing is realized by two sequences of actions for the two roles in the script. The sub-script of actor 1 can be seen on the left, and the one for actor 2 on the right of Figure 5. Note that there are two links from the initial state. The whole script corresponds to the acyclic graph script in [8]. As one can see, scripts are now interruptible because of the outgoing edges, where time-limits or other breaking conditions may be annotated. This means, both complex and reactive behavior can be combined easily.

We can go even one step further. One disadvantage of the script in Figure 5 is that the double passing cannot be iterated. Since in [8], scripts correspond to Prolog rules,

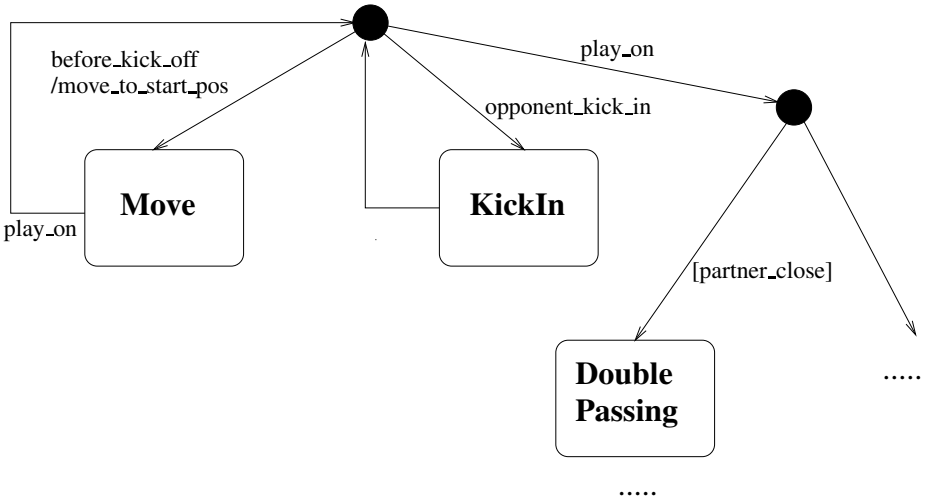

Fig. 4. Overall behavior of a soccer agent.

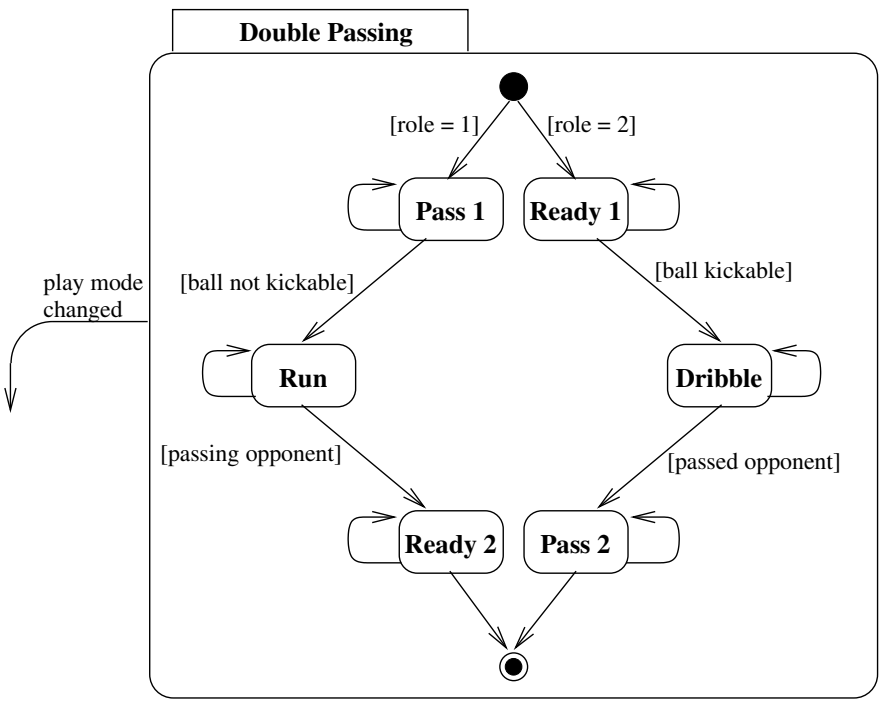

Fig. 5. Double passing script.

namely one non-recursive rule for each role, iteration could not be expressed. But now this is possible, and this is shown in Figure 6. This means, that we again have two actors in the script. On the left-hand side, the two states correspond to the role of actor 1. On the right-hand side, there are two states corresponding to the role of actor 2. But as one can see, they swap their roles at the end, so that continuous passing is possible.

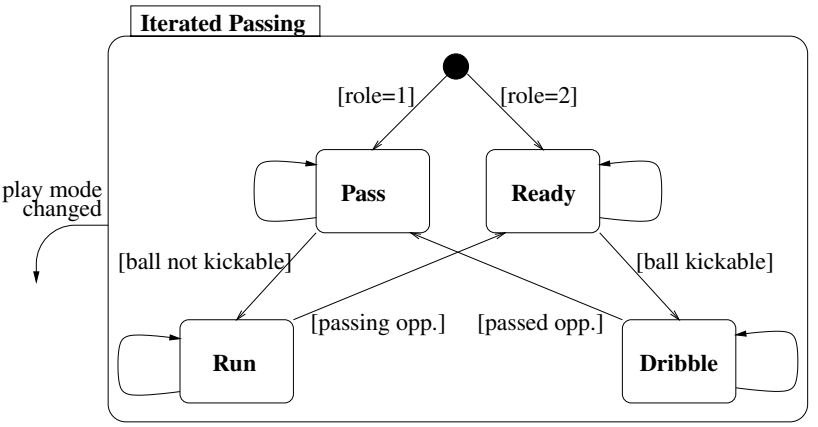

Fig. 6. Iterated passing.

4 Implementing Agents with RoboLog

4.1 Basic Strategies for Soccer Agents

As stated in the previous sections, behaviors and strategies of agents can be specified declaratively. We started implementing soccer agents which behave as follows: If the match is in *play-on* mode, then the agents kick the ball towards the opponents' goal if possible. If the ball is too far away, they try to intercept the ball or just look where it is. Formally, the latter procedure can easily be realized by the following program rules:

```
go_to_ball :-
    my_side(S),
    my_number(N),
    player_interceptionable(S,N),
    player_interception_point(S,N,X,Y),
    go_to_point(X,Y,0,100).
go_to_ball :-
    face_ball.
```

The RoboLog Koblenz team plays according to the following system: There are three defenders, and two attackers, i.e. we play with a double-forward, who go to their

respective home positions, whenever the opponent team has a free kick or a kick-in. The rest are midfielders who try to mark one of the opponent players, if they cannot get to the ball. This basic strategy of our team can be observed in the snapshot of one test game, shown in Figure 7.

Interestingly, with this simple strategy, our team shows some (emergent) cooperative behavior and appears to be competitive with many other simulation league teams. For instance, even without performing a pass explicitly, many situations arise in test games where apparently team-mates are passing the ball to each other, because they try to intercept the ball. In addition, the defenders implicitly build an offside-trap, since they follow the ball if it is played in the opponent half. Nevertheless, the overall performance has been improved by implementing these cooperative behaviors explicitly.

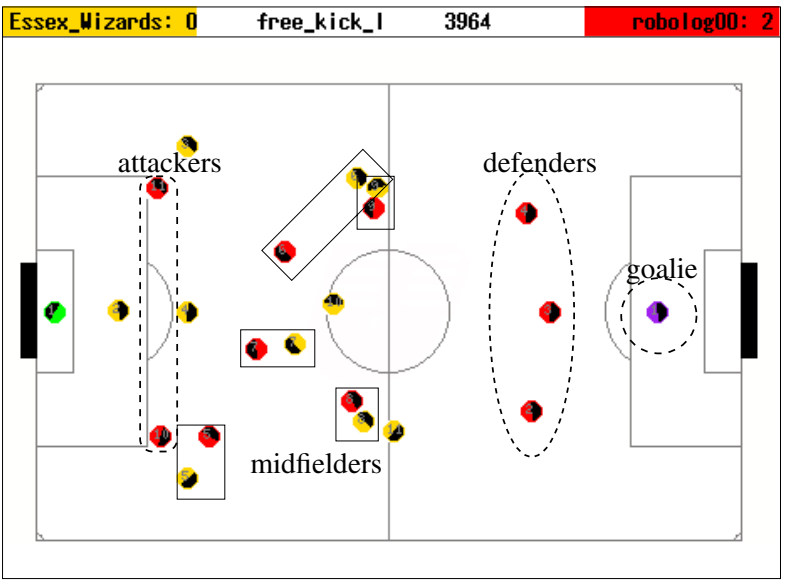

Fig. 7. Basic strategy and positioning.

4.2 Implementation in Prolog

The implementation of state machines with Prolog as specified in Section 3 is straightforward and has been done (manually up to now). The current state is stored in the Prolog database and is updated in each step. Each state transition can then be implemented by the following scheme:

```
behave :-
    current_state(S),
```

```
event(E),
test_conditions(S,E,C),
schedule_action(S,E,C,A),
update_state(S,E,C,A).
```

4.3 The New RoboLog Kernel

The kernel for RoboLog-2000 is built upon the low-level skills of the CMUnited-99 simulator team [9]. The main difference between the RoboLog-99 kernel and the RoboLog-2000 kernel lies in the overall control flow. In RoboLog-99, the Prolog predicates were proven in (quasi-)parallel to the execution of the low level kernel code. This led to uncertain timings and sometimes even to inconsistencies. Now, in RoboLog-2000, Prolog is called just after the internal world model of an agent is updated. After selecting an action, Prolog returns control to the RoboLog kernel. This is indicated in Figure 2 by the dashed arrows and boxes.

5 Related and Future Work

5.1 Related Work

There is (at least) one other related approach that employs logic and also procedural aspects, such as iteration, sequencing and non-deterministic choice between actions, namely *(Con)Golog* [4, 3]. It can be seen as an extension of logic programming, that allows us to reason about actions, their preconditions and effects in the situation calculus. Additional constructs for concurrent execution and handling exogenous action triggered by external events are present in this framework.

The advantage of this approach is that explicit reasoning about actions is possible, which allows agents to plan and extrapolate future behavior. However, it is a more or less single-agent centered system. Our approach provides a clean means to specify multi-agent behavior. Teamwork can be modeled explicitly and adapted easily. Nevertheless, external events and conflicting goals can be resolved by means of logic.

The approach in [1] is also related. It is based on *decision trees*, providing priorities in behaviors. Interruption and termination of behaviors and sequential and concurrent behaviors are also considered in this framework. Explicit communication among agents is sparsely used; coordination and collaboration is a more or less implied property. Although there are several similarities with the approach presented here, our approach is more formal, by making use of state machines and logic, such that we have a clean syntax and semantics for specifying multi-agent systems. Furthermore, in our approach the notion of (internal) states is represented more explicitly.

5.2 Conclusions and Future Work

In this paper, we have shown how multi-agent systems can be specified declaratively by means of logic programming, allowing to program cooperative and reactive behavior. Playing with early versions of our team, consisting of simple *kick-and-run* players, we beat some of the top ten teams of RoboCup-99. This shows that the performance of RoboCup agents depend highly on well implemented low-level skills. On the

other hand, the high-level parts contribute as yet an insignificant part to the success of most of the teams. However, looking at the top ten teams of this year's competition, it seems that they have similar low-level skills—in fact, several teams also made use of the CMUnited-99 library—, but different high-level capabilities such as attacking with more than one player or sophisticated defense strategies.

Our further work will concentrate on incorporating (even) more explicit teamwork and more sophisticated strategies into the system. We will follow the goal that explicit agent programming is possible, by combining logical and procedural techniques for the specification and implementation of multi-agent systems. Furthermore, formalizing the development process for multi-agent systems also enables us to apply techniques for the formal analysis and verification of the systems such as model checking.

References

1. S. Coradeschi and L. Karlsson. A role-based decision-mechanism for teams of reactive and coordinating agents. In H. Kitano, editor, *RoboCup-97: Robot Soccer WorldCup I*, LNAI 1395, pages 112–122, Berlin, Heidelberg, New York, 1998. Springer.
2. E. Corten, K. Dorer, F. Heintz, K. Kostiadis, J. Kummeneje, H. Myritz, I. Noda, J. Riekki, P. Riley, P. Stone, and T. Yeap. *Soccerserver Manual*, 5th edition, May 1999. For Soccerserver Version 5.00 and later.
3. G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In M. E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, 1997. IJCAI Inc., San Mateo, CA, Morgan Kaufmann, Los Altos, CA. Volume 2.
4. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
5. J. Murray, O. Obst, and F. Stolzenburg. RoboLog Koblenz. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, LNAI 1856, pages 628–631. Springer, Berlin, Heidelberg, New York, 2000. Team description.
6. Object Management Group, Inc. *OMG Unified Modeling Language Specification*, 1999. Version 1.3, June 1999.
7. M. P. Singh, A. S. Rao, and M. P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In G. Weiss, editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, chapter 8, pages 331–376. The MIT Press, Cambridge, MA, London, 1999.
8. F. Stolzenburg, O. Obst, J. Murray, and B. Bremer. Spatial agents implemented in a logical expressible language. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, LNAI 1856, pages 481–494. Springer, Berlin, Heidelberg, New York, 2000.
9. P. Stone, P. Riley, and M. Veloso. The CMUnited-99 champion simulator team. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, LNAI 1856, pages 35–48. Springer, Berlin, Heidelberg, New York, 2000.
10. J. Wielemaker. *SWI-Prolog 3.3 Reference Manual*. University of Amsterdam, The Netherlands, 2000.