Using a Planner for Coordination of Multiagent Team Behavior

Oliver Obst

Universität Koblenz-Landau, AI Research Group,
56070 Koblenz, Germany
fruit@uni-koblenz.de

Abstract. We present an approach to coordinate the behavior of a multiagent team using an HTN planning procedure. To coordinate teams, high level tasks have to be broken down into subtasks which is a basic operation in HTN planners. We are using planners in each of the agents to incorporate domain knowledge and to make agents follow a specified team strategy. With our approach, agents coordinate deliberatively and still maintain a high degree of reactivity. In our implementation for use in RoboCup Simulation League, first results were already very promising. Using a planner leads to better separation of agent code and expert knowledge.

1 Introduction

Coordination among different agents and the specification of strategies for multiagent systems (MAS) is a challenging task. For a human domain expert it is often very difficult to change the behavior of a multiagent system. This is especially true when not only general tasks should be specified, but also the way in which tasks are to be executed. Due to interdependencies simple changes in one place of the code may easily affect more than one situation during execution.

In this work, we suggest to use Hierarchical Task Network (HTN) planners in each of the agents in order to achieve coordinated team behavior which is in accordance with the strategy given by the human expert. The expert knowledge should be separated from the rest of the agent code in a way that it can easily be specified and changed. While pursuing the given strategy, agents should keep as much of their reactiveness as possible. HTN planning explicitly supports the use of domain specific strategies. To coordinate groups of agents, tasks usually have to be broken down into subtasks, which is one of the basic operations of HTN planning. Different levels of detail in the description of strategies further facilitate the generation of useful information for debugging or synchronization.

In classical planning, operators are deterministic and the single planning agent is the only reason for changes in the environment under consideration. We show how it is possible to use an HTN planner in the domain of robotic soccer, even though the robotic soccer environment is very different from classical planning domains. For our approach, we have chosen a team of agents using the RoboCup 3D Soccer Simulator [17] that was introduced at RoboCup-2004 in Lisbon [11].

R.H. Bordini et al. (Eds.): ProMAS 2005, LNAI 3862, pp. 90–100, 2006.

The following section describes our approach to coordinate the behavior of a multiagent team using an HTN planner. Section 3 contains the description of an implemented example. We present and discuss the results of our first tests, and a review of relevant related work. Finally, Sect. 6 concludes the paper.

2 HTN Planning for Multiagent Teams

The usual assumptions for HTN planning, like for classical planning approaches, are that we plan for a single agent who is the only cause for changes in the domain. When the plan is executed, all actions succeed as planned. Executing an action in a classical planning framework is instantaneous, it takes no time, and therefore the world is always in a defined state.

To plan for agents in a team and in a real-world domain, we have to relax some of these assumptions and find a way to deal with the new setting. Definition 1 is a way commonly used to define nondeterministic planning domains. An approach to deal with these kinds of domains is to use model checking (see for instance [3]). Depending on the problem and the desired properties of the results, the planner tries to compute solution plans that have a chance to succeed or solution plans that succeed no matter what the results of the non-deterministic actions of an agent are.

Definition 1. A *nondeterministic planning domain* is a triple $\Sigma = \langle S, A, \gamma \rangle$, where:

- S is a finite set of states.
- A is a finite set of actions.
- $\gamma \subseteq S \times A \times S$ is the state-transition relation. □

When the number of different possible results of γ is high, computing a plan can easily become intractable for domains where decisions have to be made quickly. Nevertheless, using a planner could still be useful to achieve high-level coordination for a team of several agents in a dynamic environment without using communication and without a centralized planning facility. For our approach, all planning should be done in a distributed fashion in each of the autonomous agents. The goal is that team behavior can easily be specified and extended, the task of the system is to automatically generate individual actions for the agents in accordance with those plans during execution. Despite using plans, agents should still be able to react to unforeseen changes in the environment.

2.1 Multiagent Team Behavior with HTN Plans

In Hierarchical Task Network (HTN, see also Definition 2) planning, the objective is to perform tasks. Tasks can be complex or primitive. HTN planners use *methods* to expand complex tasks into subtasks, until the tasks are primitive. Primitive tasks can be performed directly by using planning operators.

Definition 2. A *task network* is an acyclic directed graph $w = \langle N, A \rangle$, where N is the set of nodes, and A is the set of directed edges. Each node in N contains a task t_n. A task network is *primitive*, if all of its tasks are primitive, otherwise it is *nonprimitive*. □

Our approach of interleaving planning and acting and also of handling nondeterministic actions is similar to the one described in [1], where a HTN planner is used for navigation planning of a single robot. Here, like in most realistic environments, it is not enough to initially create a plan and blindly execute it, but after execution of each action the state of the world needs to be sensed in order to monitor progress. As a consequence, for generating HTN plans it is not absolutely necessary to generate a primitive task network from the beginning. Instead a HTN where the first tasks are primitive is sufficient, if we interleave planning and acting. Future tasks are left unexpanded or partially expanded until the present tasks are done and there is no other task in front. In dynamic and complex environments, creating a detailed plan can be considered as wasted time, because it is virtually impossible to predict the state of the world after only a few actions.

Rather than expanding complex tasks completely, our planner generates what is called *plan stub* in [1], a task network with a primitive task as the first task. As soon as a plan stub has been found, an agent can start executing its task. The algorithm in Fig. 1 expands a list of tasks to a plan stub, if it is not already in that form.

Function: plan$(s_{now}, \langle t_1, ..., t_k \rangle, O, M)$
Returns: (w, s), with w an ordered set of tasks, s a state; or *failure*

if $k = 0$ **then return** $(\emptyset, s_{now})$ `// i.e. the empty plan`
if t_1 *is a pending primitive task* **then**
  $active \leftarrow \{(a, \sigma) | a$ is a ground instance of an operator in O,
    σ is a substitution such that a is relevant for $\sigma(t_1)$,
    and a is applicable to $s_{now}\}$;
  if $active = \emptyset$ **then return** *failure*;
  nondeterministically choose any $(a, \sigma) \in active$;
  return $(\sigma(\langle t_1, ..., t_k \rangle), \gamma(s_{now}, a))$;
else if t_1 *is a pending complex task* **then**
  $active \leftarrow \{m | m$ is a ground instance of a method in M,
    σ is a substitution such that m is relevant for $\sigma(t_1)$,
    and m is applicable to $s_{now}\}$;
  if $active = \emptyset$ **then return** *failure*;
  nondeterministically choose any $(m, \sigma) \in active$;
  $w \leftarrow$ subtasks$(m).\sigma(\langle t_1, ..., t_k \rangle)$;
  set all tasks in front of t_1 to *pending*, set t_1 to *expanded*;
  return plan(s_{now}, w, O, M);
else
  `//` t_1 `is an already executed expanded task and can be removed`
  return plan$(s_{now}, \langle t_2, ..., t_k \rangle, O, M)$;

Fig. 1. Creating an initial plan stub (Notation according to [7])

In classical planning, executing an action takes no time. That means immediately after executing a planning operator, the world is in the successor state. In our approach we have to consider that actions are not instantaneous and might not even yield the desired result. The first problem is when to regard operators as finally executed: Depending on the actual domain agents are acting in, actions can be regarded as finished after a given amount of time or when a specified condition holds. This domain specific solution to this problem is not part of the algorithms in this paper.

A second problem is the computation of the successor state: as defined above, for non-deterministic environments γ is a relation with possibly several results for the same state-action pair. For our algorithms, we expect γ to be a function returning the *desired* successor state. Likewise, the effects of an operator describe the desired effects. The underlying assumption is that operators have a single purpose so that the desired successor state can be uniquely described. The desired effects can be used by the operators to coordinate actions of teammates during the same plan step. For this, we introduce multiagent operators in Definition 3, which is effectively a shortcut for defining a set of combinations of operators. Actions that are executed simultaneously but which do not contribute to the desired effects of the multiagent operator are simply not included. This makes it easy for the developer of a multiagent team to create team operators, but the disadvantage is that agents not part of the multiagent team cannot be regarded with our approach.

Definition 3 (Multiagent Operator). Let $o_1, ..., o_n$ be operators, and $\text{effects}^-(o_j) \cap \text{effects}^+(o_k) = \emptyset$ for all $j, k \in \{1, ..., n\}$. p is a new operator with $\text{name}(p) = \text{name}(o_1)$ `while` $\langle \text{name}(o_2), ..., \text{name}(o_n) \rangle$. The preconditions and effects of p are defined as unions over the preconditions and effects of all o_i, respectively:

$$\text{pre}(p) = \bigcup_{i=1,...,n} \text{pre}(o_i), \quad \text{and} \quad \text{effects}(p) = \bigcup_{i=1,...,n} \text{effects}(o_i)$$

□

At the same time, the desired successor state is used to check the success of the last operator application in the second algorithm (see Fig. 2). Here, the executed tasks are removed from the plan and the first algorithm is used again to create an updated plan stub.

2.2 Handling Non-determinism

To handle non-determinism, we treat a plan as a stack. Tasks on this stack are marked as either *pending* or as *expanded*. Pending tasks are either about to be executed, if they are primitive, or waiting to be further expanded, if they are complex. Tasks marked as expanded are complex tasks which already have been expanded into subtasks. If a subtask of a complex task fails, all the remaining subtasks of that complex task are removed from the stack and it is checked if the complex task can be tried again. If a task was finished successfully, it is simply removed from the stack.

Function: step($s_{expected}, s_{now}, \langle t_1, ..., t_k \rangle, O, M$)
Returns: (w, s), with w a set of ordered tasks, s a state; or *failure*

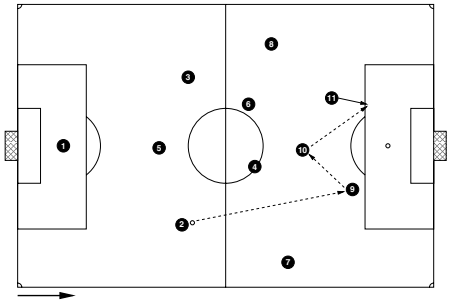
if $k = 0$ **then return** $(\emptyset, s_{now})$ `// i.e., the empty plan`
if t_1 *is a pending task* **then**
  if $s_{expected}$ *is valid in* s_{now} **then**
    $i \leftarrow$ the position of the first non-primitive task in the list;
    return *plan*($s_{now}, \langle t_i, ..., t_k \rangle, O, M$);
  else
    `//` t_1 `was unsuccessful; remove all pending children of our parent task`
    return *step*($s_{expected}, s_{now}, \langle t_2, ..., t_k \rangle, O, M$);
else
  `//` t_1 `is an unsuccessfully terminated expanded task, try to re-apply it`
  active $\leftarrow \{m | m$ is a ground instance of a method in M,
      σ is a substitution such that m is relevant for $\sigma(t_1)$,
      and m is applicable to $s_{now}\}$;
  if *active* $= \emptyset$ **then**
    `//` t_1 `cannot be re-applied, remove it from the list and recurse`
    return step($s_{expected}, s_{now}, \langle t_2, ..., t_k \rangle, O, M$);
  else
    nondeterministically choose any $(m, \sigma) \in$ *active*;
    $w \leftarrow$ subtasks(m).$\sigma(\langle t_1, ..., t_k \rangle)$;
    set all tasks in front of t_1 to *pending*, set t_1 to *expanded*;
    return plan(s_{now}, w, O, M);

Fig. 2. Remove the top primitive tasks and create a new plan stub

3 Robotic Soccer Sample Implementation

To give an example, we take an example from the simulated soccer domain [10, 11] and the complex top level task `play_soccer` has already been partially expanded as shown in Fig. 3. All the pending tasks in Fig. 3 are still complex tasks. To create a plan stub, the planner needs to further expand the top pending task. At this level of expansion, the plan still represents a team plan, as seen from a global perspective. When team tasks – like `pass(2,9)` – get further expanded to agent tasks, each agent has to find its role in the team task. In the soccer domain, agents usually have predefined roles which can be used to describe roles in specific tasks. An alternative possibility is a distance based role selection.

Agent #2 will expand `pass(2,9)` to `do_pass(9)`, agent #9 has to do a `do_receive_pass` for the same team task. The other agents position themselves relatively to the current ball position with `do_positioning` at the same time. The desired effect of `pass(2,9)` is the same for all the agents, even if the derived primitive task is different depending on the role of the agent. That means each agent has to execute a different action, which is realized as `C++` function call in our case, and at the same time an operator has to update the desired successor state independently. To express that an agent should execute the `do_positioning`



```
pending-pass(2,9)
pending-pass(9,10)
pending-leading-pass(10,11)
expanded-diagram-4
expanded-build_up_long_pass
expanded-build_up_play
pending-final_touch
pending-shooting
expanded-offensive_phase
expanded-play_soccer
```

Fig. 3. Soccer Example Situation (left) and plan stack during planning (right)

```
method pass(A,B)
pre [my_number(A)]
subtasks [do_pass(B) while pass(we,A,B),
do_positioning].

method pass(A,B)
pre [my_number(B)]
subtasks [do_receive_pass while pass(we,A,B)].

method pass(A,B)
pre [my_number(C),#\=(A,C),#\=(B,C)]
subtasks [do_positioning while pass(we,A,B)].
```

Fig. 4. Different methods to reduce the team task `pass(A,B)` to agent tasks

behavior while taking the effect of a simultaneous pass between two teammates into account, we are using terms like `do_positioning while pass(we,2,9)` in our planner. Figure 4 shows methods reducing the team task `pass(A,B)` to different primitive player tasks.

In different agents, the applicable methods for the top team task `pass(2,9)` lead to different plan stubs. This is an important difference to the work presented in [1]. The plan stubs created as first step for agent 9 and agent 11 are shown in Fig. 5. When a plan stub is found, the top primitive tasks are passed to the `C++` module of our agent and executed. A 'step' for a plan in our agents can consist of more than a single action, for example, we do not want the agent who passes the ball to stop acting while the ball is already moving to a teammate, but instead after the kick the agent should adjust its position relative to the ball until the ball reached its destination and the step is finished. If possible, the agent has to execute all pending primitive tasks until the next step in the plan starts. If there are pending primitive tasks after one step is finished, these agent tasks are simply removed from the plan stack and the next team task can be expanded. Figure 6 shows the plan stub for the second step from the diagram in Fig. 3. For player 9, the expansion leads to a plan stub with two primitive tasks in a plan step while for player 11 there is only one task to be executed.

```
pending-(do_receive_pass while
         pass(we, 2, 9)),
expanded-pass(2, 9),
pending-pass(9, 10),
pending-leading_pass(10, 11),
expanded-diagram-4,
...
```

```
pending-(do_positioning while
         pass(we, 2, 9)),
expanded-pass(2, 9),
pending-pass(9, 10),
pending-leading_pass(10, 11),
expanded-diagram-4,
...
```

Fig. 5. Step 1: Plan Stubs for player 9 and player 11

```
pending-(do_pass(10) while
         pass(we, 9, 10)),
pending-do_positioning,
expanded-pass(9, 10),
pending-leading_pass(10, 11),
expanded-diagram_4,
...
```

```
pending-(do_positioning while
         pass(we, 9, 10)),
expanded-pass(9, 10),
pending-leading_pass(10, 11),
expanded-diagram_4,
...
```

Fig. 6. Step 2: Plan Stubs for player 9 and player 11

What we did not address so far was the point in time when the transition from one plan step to the next step takes place. Here, the basic idea is the following: each step in plans for our team stops or starts with an agent being in ball possession. If any of the agents on the field is in ball possession, we can check for the desired effect of our previous action. If the action succeeded, the right agent possesses the ball and the planner can continue planning by generating the next plan stub. If an adversarial agent intercepted the ball, the last action failed and the planner needs to backtrack. For dribbling, the planner needs to check if the dribbling agent still possesses the ball and arrived at the desired destination in order to start with the next step.

4 Results and Discussion

For our approach of generating coordinated actions in a team we implemented an HTN planner in Prolog which supports interleaving of planning and acting. Our planner supports team actions by explicitly taking the effects of operators simultaneously used by teammates into account. The planner ensures that the agents follow the strategy specified by the user of the system by generating individual actions for each of the agents that are in accordance with it. The *lazy evaluation* in the expansion of subtasks which generates plan stubs rather than a full plan, makes the planning process very fast and enables the agents to stay reactive to unexpected changes in the environment. The reactiveness could, however, be increased by adding a situation evaluation mechanism that is used prior to invoking the planner. This would improve the ability to exploit sudden, short-lived opportunities during the game.

We implemented a distributed planning system in the sense that each of the agents uses its own planner. This was, however, somewhat facilitated by the fact

that agents in the RoboCup 3D Simulation League are equipped with sensors that provide them with a full (though possibly inaccurate) view of the world, similar to Middle-size League robots using omni-vision cameras.

To truly evaluate the approach we presented, it would be necessary to measure the effort it takes to create a team and compare it to other approaches to create a team exhibiting the same behavior. We strongly believe that our approach leads to a modular behavior design and facilitates rapid specification of team behavior for *users* of our agents, but we cannot present numbers here. Our plans can describe plays as introduced in [2], which have shown to be useful for synchronization in a team. There are some important differences to plays, however. First, our approach supports different levels of abstraction in plans. That means there are different levels of detail available to describe what our team and each single agent is actually doing, from very abstract tasks down to the agent level tasks. A second important difference is that the planner can find alternative ways to achieve tasks. This is possible if plays are specified in terms of player roles or properties rather than fixed player numbers. The approach in [2] was used for Small Size League, where the numbers of players and the number of alternative ways of doing plays is low. That means in Small Size League, a plan is either applicable or not. For Simulation League or larger teams in general, more opportunities are possible for which an approach using fixed teammates seems to restrictive. On the other hand, the approach in [2] supports adaptation by changing weights for the selection of successful plays. In our approach, the corresponding functionality could be achieved by changing the order in which HTN methods are used to reduce tasks. At this point in time, our approach does not support this yet. As soon as we do have an adaptive component in our approach, it makes sense to compare results of our team with and without adaptation.

The way our plans are created and executed, we assume synchronous actions for all our agents. Our team actions are geared to actions of the player in ball possession, so this simplification can be made. There are a few situations in soccer, where more detailed reasoning over the time actions take would be useful. This includes for instance all situations where a ball receiver should appear at the receiving position *just in time* to surprise the opponent. In our approach, we make this possible by synchronizing the behavior of two agents in the *current step* by using both ball and agent velocity to estimate interception times, in the operator implementations outside of the planning procedure. Inside our planning procedure, we do not reason about durations, which would be useful to make asynchronous actions possible.

Although more detailed evaluations have to be carried out, the first tests using the planner seem very promising and indicate that our approach provides a flexible, easily extendable method for coordinating a team of agents in dynamic domains like the RoboCup 3D Simulation League.

5 Related Work

Several approaches that use a planning component in a MAS can be found in the literature.

In [6], the authors describe a formalism to integrate the HTN planning system SHOP [16] with the IMPACT [21] multiagent environment (A-SHOP). The preconditions and effects used in SHOP are modified so that preconditions are evaluated using the code-call mechanism of the framework, and effects change the state of agents. While the environment of this work clearly is a multiagent system, the planning is carried out centralized by a single agent. This is a contrast to our approach, which uses a planner in each of the agents to coordinate the agents actions.

A general HTN planning framework for agents in dynamic environments has been presented in [9]. The authors show how to integrate task decomposition of HTN planning, action execution, program updates, and plan modifications. The planning process is done via abstract task decomposition and is augmented to include additional information such as the history of action execution for the plans to enable their incremental modification. Rules are given for plan modifications after having executed certain actions or after program updates. In the robotic soccer domain, however, the results of actions like e.g. kicking the ball cannot be undone. Thus, the plan modification mechanism given in [9] does not apply and could not easily be used for our purposes.

HTN planning has also been studied in the context of creating intelligent, cooperating Non-Player Characters in computer games. In [13], an HTN planner is used to enable agents in the highly dynamic environment of the Unreal Tournament game to pursue a grand strategy designed for the team of agents.

Bowling *et al.* [2] presents a strategy system that makes use of *plays* (essentially being multiagent plans) to coordinate team behavior of robots in the RoboCup Small Size League. Multiple plays are managed in a *playbook* which is responsible to choose appropriate plays, and evaluate them for adaption purposes. The plays are specified using a special language designed with ease of readability and extensibility in mind. Preconditions can be specified that determine when a play can be executed. Furthermore, plays contain termination conditions, role assignments and sequences of individual behaviors. While the use of preconditions resembles a classical planning approach, the effects of individual plays are not specified due to the difficulties in predicting the outcome of operators in the dynamic environment. This is in contrast to our approach, as we use *desired effects* of the operators in our plans. Another difference is that in [2] the planning component is also centralized.

Other approaches towards multiagent collaboration like [5, 8] are based on negotiations between the agents in a multiagent system. However, as pointed out in [20], this kind of complex communication might take too much time or might even be infeasible in highly dynamic real-time domains like robotic soccer.

The work in [15, 14] describes the approach to creating our agents so far: We used UML statecharts to specify behaviors for agents in a multiagent system. The agents were designed in a top-down manner with a layered architecture. At the highest level global patterns of behavior are specified in an abstract way, representing the different states the agent can be in. For each of these states, an agent has a repertoire of skeleton plans in the next layer. These are applicable

as long as the state does not change. Explicit specification of cooperation and multiagent behaviors can be realized. The third and lowest level of the architecture encompasses the descriptions for the simple and complex actions the agents can execute, which are used by the scripts in the level above.

This hierarchical decomposition of agent behaviors is similar to the HTN plans described in this work. However, the separation of domain description knowledge and the reasoning formalism accomplished through the use of the HTN planner within our agents provides us with much greater flexibility in respect to the extensibility of methods and operators, compared to the amount of work needed to change the state machine description.

6 Conclusion and Future Work

We presented a novel approach that uses an HTN planning component to coordinate the behavior of multiple agents in a dynamic MAS. We formalized expert domain knowledge and used it in the planning methods to subdivide the given tasks. The hierarchical structure of the plans speeds up the planning and also helps to generate useful debugging output for development. Furthermore, the system is easily extensible as the planning logic and the domain knowledge are separated.

In order to use the system in the RoboCup competitions, we plan to integrate a lot more subdivision strategies for the different tasks as described in the diagrams in [12]. A desirable enhancement to our work would be the integration of an adaption mechanism. Monitoring the success of different strategies against a certain opponent, and using this information in the choice of several applicable action possibilities, as e.g. outlined in [2], should be explored. The introduction of *durative actions* into the planner (see for instance [4]) would give a more fine grained control over the parallelism in the multiagent plans. *Simple Temporal Networks* as used in [19] seem to be well suited for this purpose. Furthermore, a situation assessment will be added to the agents to be able to exploit unforeseen situations in a more reactive manner. Finally, we want to restrict the sensors of the agents to receive only partial information about the current world state, and address the issues that result for the distributed planning process.

References

1. Thorsten Belker, Martin Hammel, and Joachim Hertzberg. Learning to optimize mobile robot navigation based on HTN plans. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2003)*, pages 4136–4141, Taipei, Taiwan, September 2003.
2. Michael Bowling, Brett Browning, and Manuela Veloso. Plays as team plans for coordination and adaptation. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, Vancouver, June 2004.
3. Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

4. Alex M. Coddington, Maria Fox, and Derek Long. Handling durative actions in classical planning frameworks. In John Levine, editor, *Proceedings of the 20th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 44–58. University of Edinburgh, December 2001.
5. Philip R. Cohen, Hector J. Levesque, and Ira Smith. On team formation. *Contemporary Action Theory*, 1998.
6. Jürgen Dix, Héctor Muñoz-Avila, and Dana Nau. IMPACTing SHOP: Planning in a Multi-Agent Environment. In Fariba Sadri and Ken Satoh, editors, *Proceedings of CLIMA 2000, Workshop at CL 2000*, pages 30–42. Imperial College, 2000.
7. Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning Theory and Practice*. Morgan Kaufmann, San Francisco, CA, USA, 2004.
8. Barbara J. Grosz. AAAI-94 presidential address: Collaborative systems. *AI Magazine*, 17(2):67–85, 1996.
9. Hisashi Hayashi, Kenta Cho, and Akihiko Ohsuga. A new HTN planning framework for agents in dynamic environments. In Jürgen Dix and João Leite, editors, *CLIMA IV 2004*, number 3259 in Lecture Notes in Computer Science, pages 108–133. Springer, Berlin, Heidelberg, New York, 2004.
10. Marco Kögler and Oliver Obst. Simulation league: The next generation. In Polani et al. [18], pages 458–469.
11. Pedro Lima, Luís Custódio, Levent Akin, Adam Jacoff, Gerhard Kraezschmar, Beng Kiat Ng, Oliver Obst, Thomas Röfer, Yasutake Takahashi, and Changjiu Zhou. Robocup 2004 competitions and symposium: A small kick for robots, a giant score for science. *AI Magazine*, 2005. To appear.
12. Massimo Lucchesi. *Coaching the 3-4-1-2 and 4-2-3-1*. Reedswain Publishing, 2001.
13. Héctor Muñoz-Avila and Todd Fisher. Strategic planning for Unreal Tournament bots. In *Proceedings of AAAI-04 Workshop on Challenges on Game AI*. AAAI Press, 2004.
14. Jan Murray. Specifying agent behaviors with UML statecharts and StatEdit. In Polani et al. [18], pages 145–156.
15. 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*, pages 526–530. Springer, Berlin, Heidelberg, New York, 2002. Team description.
16. Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*, pages 968–975, 1999.
17. Oliver Obst and Markus Rollmann. SPARK – A Generic Simulator for Physical Multiagent Simulations. *Engineering Intelligent Systems*, 13, 2005. To appear.
18. Daniel Polani, Brett Browning, Andrea Bonarini, and Kazuo Yoshida, editors. volume 3020 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.
19. Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, Toulouse, France, April 2002.
20. Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 1999.
21. V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems*. MIT Press/AAAI Press, Cambridge, MA, USA, 2000.