HTN Planning for Flexible Coordination Of Multiagent Team Behavior

Oliver Obst, Anita Maas, Joschka Boedecker

3/2005
HTN Planning for Flexible Coordination Of Multiagent Team Behavior

Oliver Obst      Anita Maas
Joschka Boedecker
AI Research Group, Universität Koblenz
Universitätsstr. 1,
56070 Koblenz, Germany
{fruit, maas, jboedeck}@uni-koblenz.de

May 20, 2005

Coordination of agents in a dynamic and nondeterministic environment is a difficult task. There are many approaches to this problem where agents are controlled reactively. In this paper we present an approach to coordinate the behavior of a multiagent team using an HTN (Hierarchical Task Network) 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 the RoboCup Simulation League, first results were already very promising. Using a planner helps to maintain a clear agent design, separating the agent code from the expert domain 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 reactivity 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 for the RoboCup 3D Soccer Simulator [Obst and Rollmann, 2005] that was introduced at RoboCup-2004 in Lisbon [Lima et al., 2005].

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 give a review of relevant related work. Finally, Section 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. The definition below 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 [Cimatti et al., 2003]). 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 nondeterministic actions of an agent are.

Definition 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.

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 give a review of relevant related work. Finally, Section 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. The definition below 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 [Cimatti et al., 2003]). 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 nondeterministic actions of an agent are.

Definition 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 $\gamma$ 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 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. A further goal of using a planner is that team behavior can easily be specified and extended, which is supported by the separation of agent code and expert domain knowledge.

2.1 Multiagent Team Behavior with HTN Plans

In Hierarchical Task Network (HTN, see Definition below) 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** A task network is an acyclic directed graph $w = (N, A)$, 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 [Belker et al., 2003] where an 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, an 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 becomes virtually impossible to predict the state of the world after only a few actions already.

Rather than expanding complex tasks completely, our planner generates what is called plan stub in [Belker et al., 2003], 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. The notation of our algorithms is similar to the one used in [Ghallab et al., 2004]: $(t_1, ..., t_k)$ is a set of tasks, $O$ is the set of operators, $M$ is the set of methods, subtasks($m$) stands
for the set of subtasks of a method \( m \), and the dot (\( . \)) used in the algorithms denotes a concatenation.

**Function**: \( \text{plan}(s_{\text{now}}, \langle t_1, ..., t_k \rangle, O, M) \)

**Returns**: (\( w, s \)), with \( w \) an ordered set of tasks, \( s \) a state; or failure

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

Figure 1: Creating an initial plan stub.

In classical planning, executing an action takes no time. This means that 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 nondeterministic environments \( \gamma \) is a relation with possibly several results for the same state-action pair. For our algorithms, we expect \( \gamma \) to be a function returning the desired successor state, more precisely a subset of the desired successor state. The returned state should describe those properties of the environment that are deliberately changed by an action. 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, 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 modeled as part of the multiagent team cannot be regarded with our approach.

**Definition (Multiagent Operator)**

Let \( o_1, \ldots, o_n \) be operators, effects\(^{-}(o) \) and effects\(^{+}(o) \) the negative and positive effects of an operator \( o \), respectively, and effects\(^{-}(o_j) \cap \text{effects}\(^{+}(o_k) = \emptyset \) for all \( j, k \in \{1, \ldots, n\} \).

\( p \) is a new operator with \( \text{name}(p) = \text{name}(o_1) \) while \( \langle \text{name}(o_2), \ldots, \text{name}(o_n) \rangle \). The preconditions and effects of \( p \) are defined as unions over the preconditions and effects of all \( o_i \), respectively:

\[
\text{pre}(p) = \bigcup_{i=1,\ldots,n} \text{pre}(o_i), \quad \text{and} \\
\text{effects}(p) = \bigcup_{i=1,\ldots,n} \text{effects}(o_i) 
\]

The multiagent operator describes the actions of several agents; the operator in front of the while is the one actually executed by the agent, and the operators after it are used to determine the collective preconditions and effects of the team action. In the algorithms, a multiagent operator is treated as regular operator with the difference that at execution time only the operator in front of the while leads to an action by the respective agent.

The desired successor state is used to check the success of the last operator application in the second algorithm (see Fig. 2). Both algorithms treat plans 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. The function \( \text{step} \) removes executed tasks from the plan, it is called whenever a step was finished. If the task was successfully executed, only the finished task is removed from the stack – and possibly also parent tasks if there are no further pending child tasks. If execution of the task failed, all subtasks of the parent task have to be removed. In this case, it is checked if the parent complex task can be tried again. Function \( \text{plan} \) from Fig. 1 is used to create an initial plan stub by calling the function with an initial task. It is also used to create an updated plan stub when called from \( \text{step} \).
### 3 Robotic Soccer Sample Implementation

To give an example, we take the simulated soccer domain [Kögler and Obst, 2004; Obst and Rollmann, 2005]. In [Dylla et al., 2005], we formalized soccer domain knowledge as it can be found in soccer theory books [Lucchesi, 2001]. Based on the diagrams in this book (see for example Fig. 3), we created HTN methods for the simulated soccer domain.

Figure 4 shows that part of the plan stack which contains the team plan for the situation depicted in Fig. 3. All pending tasks in this plan stack are still complex tasks on the team level, so that this stack could be part of any of the agents on the field. It was created by expanding the top level task play_soccer into offensive_phase. The task offensive_phase was expanded to build_up_play, final_touch and shooting. In the current situation, only the first task of this sequence, build_up_play, was already expanded to build_up_play_long_pass, which in turn was expanded to diagram-4. Finally, diagram-4 expanded to the sequence pass(2,9), pass(9,10) and leading_pass(10,11).

---

**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
    - $i$ ← the position of the first nonprimitive task in the list;
    - return plan($s_{now}$, $\langle t_1, ..., t_k \rangle$, 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
  - $active ← \{m | m$ is a ground instance of a method in $M$, $\sigma$ 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 $(m, \sigma) ∈ active$;
    - $w ← \text{subtasks}(m).\sigma(\langle t_1, ..., t_k \rangle)$;
    - set all tasks in front of $t_1$ to pending, set $t_1$ to expanded;
    - return plan($s_{now}$, $w$, O, M);

---

Figure 2: Remove the top primitive tasks and create a new plan stub.
To create a plan stub so that an action can be executed, the planner needs to further expand the top pending task, in this case pass(2,9). When team tasks get further expanded to agent tasks, each agent has to find its role in the team task: the HTN methods contain variables that need to be unified with actual uniform numbers. In our soccer example, the role finding is done via preconditions on the current formation, position and function of the respective players in the formation. This also means that symmetric situations are handled automatically (provided the formation of the team is also symmetric).

Further expanding the abstract plan, 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 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 5 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 [Belker et al., 2003]. The plan stubs created as first step for agent 9 and agent 11 are shown in Fig. 6 and 7. 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.
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)].

Figure 5: Different methods to reduce the team task pass(A,B) to agent 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. Figures 8 and 9 show the plan stub for the second step from the
diagram in Fig. 3. For player 11, the expansion leads to a plan stub with two primitive
tasks in a plan step while for player 9 there is only one task to be executed.

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
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,
...

Figure 6: Step 1: Player 9 receives the pass.

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

Figure 7: Step 1: While players 2 and 9 pass, player 11 stays in the formation.

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

Figure 8: Step 2: Player 9 passes to 10.

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

Figure 9: Step 2: Player 11 stays in the formation while player 9 passes to 10.
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 reactivity 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. A comparison to the results of other teams is not helpful here, because better results do not necessarily mean that the planning procedure is the reason for differences in the performance: in many cases, careful engineering can lead to implementations that perform well without using AI techniques.

Our plans can describe plays as introduced in [Bowling et al., 2004], 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 [Bowling et al., 2004] 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 [Bowling et al., 2004] 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 [Dix et al., 2000], the authors describe a formalism to integrate the HTN planning system SHOP [Nau et al., 1999] with the IMPACT [Subrahmanian et al., 2000] 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.

Planning in each of the agents in the RETSINA multiagent system [Paolucci et al., 2000] is also HTN based. Additionally to the planning module, RETSINA agents consist of a scheduler, a communicator and an execution monitor. The architecture of the system is targeted towards agents that interact by exchanging informations, in contrast to our approach where agents basically cooperate by physical actions. RETSINA uses a special mechanism to suspend tasks that need to wait for information gathering processes. To decide if the execution of a task failed, RETSINA uses sets of constraints describing
conditions that should hold during or after the execution. The basic planning algorithm in RETSINA returns partial solution plans, then they are scheduled for execution and finally executed by the execution monitor. In our approach, the planner returns plan stubs where the first task is already executable.

A general HTN planning framework for agents in dynamic environments has been presented in [Hayashi et al., 2004]. 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 [Hayashi et al., 2004] 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 [Muñoz-Avila and Fisher, 2004], 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. [Bowling et al., 2004] 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 [Bowling et al., 2004] the planning component is also centralized.

A centralized planner is also used in [Riley and Veloso, 2002] to generate team plans for distributed execution. A coach agent observes the opponents agents and uses opponent models in the planning process. It communicates the plan to the agents periodically and the agents use this information to maintain consistency in their cooperating behavior. The team plans are represented as Simple Temporal Networks which are essentially directed graphs describing the temporal constraints between events. Using this representation, the specification of parallel events is facilitated and can also be used for monitoring purposes. Despite those appealing features of Simple Temporal Networks for multiagent plan specification, we used a rather more traditional representation without any explicit modeling of execution times for the operators for the sake of easier integration into the planner. They might, however, be beneficial for a more fine grained control
over the parallelism in our plans.

Other approaches towards multiagent collaboration like [Cohen et al., 1998; Grosz, 1996] are based on negotiations between the agents in a multiagent system. However, as pointed out in [Stone and Veloso, 1999], 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 [Murray et al., 2002; Murray, 2003] 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 [Lucchesi, 2001]. 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 [Bowling et al., 2004], should be explored. The introduction of durative actions into the planner (see for instance [Coddington et al., 2001]) would give a more fine grained control over the parallelism in the multiagent plans. Simple Temporal Networks as used in [Riley and Veloso, 2002] 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


References


Available Research Reports (since 2000):

2005


2004


8/2004 Achim Rettinger. Learning from Recorded Games: A Scoring Policy for Simulated Soccer Agents.


2003


15/2003 Peter Baumgartner, Ulrich Furbach, Margret Gross-Hardt, Thomas Kleemann, Christoph Wernhard. KRHyper Inside — Model Based Deduction in Applications.

14/2003 Christoph Wernhard. System Description: KRHyper.


9/2003 Nicholas Kushmerick, Bernd Thomas. Adaptive information extraction: Core technologies for information agents.


7/2003 Ulrich Furbach. AI - A Multiple Book Review.


5/2003 Oliver Obst. Using Model-Based Diagnosis to Build Hypotheses about Spatial Environments.
13/2001 Annette Pook. Schlussbericht “FUN - Funkunterrichtsnetzwerk”.


9/2001 Andreas Winter. Exchanging Graphs with GXL.

8/2001 Marianne Valerius, Anna Simon. Slicing Book Technology — eine neue Technik für eine neue Lehre?.


2/2001 Carola Lange, Harry M. Sneed, Andreas Winter. Applying GUPRO to GEOS – A Case Study.


2000


2/2000  *Peter Baumgartner, Fabio Massaci.* The Taming of the (X)OR.