Linköping Electronic Articles in
Computer and Information Science
Vol. 5(2000): nr 18

Interleaving Planning and Execution in a Multiagent Team Planning Environment

Paolucci, M.[1], Shehory, O.[2] and Sycara, K.[1]
1. The Robotics Institute, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
paolucci,katya@cs.cmu.edu
2. IBM Research Lab in Haifa, The Tel Aviv Site
2 Weizmann St., Tel Aviv, 61336 Israel
onn@il.ibm.com

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/2000/018/
Revised version

Original version published on December 21, 2000
revised version on March 20, 2001 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 5(2000): nr 18. http://www.ep.liu.se/ea/cis/2000/018/. *December 21, 2000.*

This URL will also contain a link to the author's home page.

Abstract

Agents in a multiagent system need to share information and services. For this, they need to be able to interleave deliberative planning with execution of actions. The deliberative planning is needed to decide which actions to perform to achieve an objective, whereas execution of some of the actions is needed to make a more informed decision on the other actions and to access services provided by other agents.

HITaP is a planner that interleaves planning and execution: using HITaP an agent can, during planning, gather information by either direct inspection of the domain or by firing queries to other agents and recording their answers. Interleaving planning and execution plays a crucial role in an agent's ability to handle uncertainty, to construct shared plans with other agents and to manage the negotiation process that leads to agreement with the agent's teammates on these plans.

HITaP is implemented and currently used as a planning module for agents in the RETSINA multiagent system. These agents cooperate to solve problems in different domains that range from portfolio management to command and control decision support systems.[1]

[1]The authors thank Dirk Kalp and Anandeep Pannu for their contribution to the initial implementation of the planner. In addition we are grateful to Joseph Giampapa for the stimulating discussions on team behavior. This research has been sponsored in part by ONR grant N-00014-96-16-1-1222 by DARPA grant F-30602-98-2-0138.

1 Introduction

Agents in a Multi-agent System (MAS) cannot adhere to the STRIP assumptions that are often made by single agents, because in a MAS knowledge and capabilities are distributed among all the agents. Since no single agent has complete knowledge of the world it is acting in, no agent can construct its plans without requesting information from other agents or by direct inspection through its own sensors. In either case, the agent needs to suspend the construction of its plan and gather the information that it needs.

Capabilities are also distributed within a MAS: no single agent is able to perform all the tasks that can be performed by all the agents in the system. In order to achieve their goals, agents have to interact with others, by subcontracting tasks to other agents, or by forming teams and developing a joint plan with their teammates to overcome the problem in a mutually supportive way [3, 10]. In addition, the presence of multiple agents generates uncertainty even in controlled environments because the actions of one agent may affect the course of action of another agent. Furthermore, the services provided by the agents in the MAS change dynamically, as new agents enter the system while others exit.

Agents that employ planning systems that maintain the classical distinction between planning and execution [7, 21, 2, 12, 13], in which an agent first constructs a plan and at a later time executes it, cannot interact with other agents to gather information, subcontract tasks or form a team. Such agents are therefore forced to construct their plans with little knowledge of the world around them or of the services available in the system. Finally, they cannot react gracefully to unexpected conditions, they just fail.

In this paper we present HITaP, a planner that is used as a component of RETSINA[2] agents [26]. The RETSINA agents interact on a peer to peer basis within the RETSINA MAS infrastructure [27] which supports agent coordination and self organization without the need of any centralized component as required in [15]. HITaP supports peer to peer coordination, by allowing agents to find out which information they need and to select other agents that can provide it. Information gathering is particularly crucial during planning, since it allows the agent to decide on the most appropriate course of action that leads to the desired goal. By supporting the interleaving of planning and execution, HITaP allows the agent to continue planning, while asking other agents for the needed information.

HITaP handles uncertainty in two ways: first, it allows the construction of contingency plans that specify how to react to unexpected events that may lead the agent to a failure. Second, it allows automatic replanning when an unexpected event prevents the agent from completing its plan. Finally, HITaP allows negotiation while planning; negotiation is a challenging task for classical planners because the agent does not know in advance for how long the negotiation will proceed, and what the teammates' reactions to its own proposals will be. HITaP solves this problem by relying (again) on interleaving planning and execution: the agent deliberates to decide what to propose to other agents and then, while still planning, executes communication actions that lead to the exchange of information with other agents until an agreement is reached.

The paper is organized as follow: section 2 introduces the internal architecture of RETSINA agents and the role of the planning component;

[2]RETSINA stands for Reusable Environment for Task Structured Intelligent Networked Agents

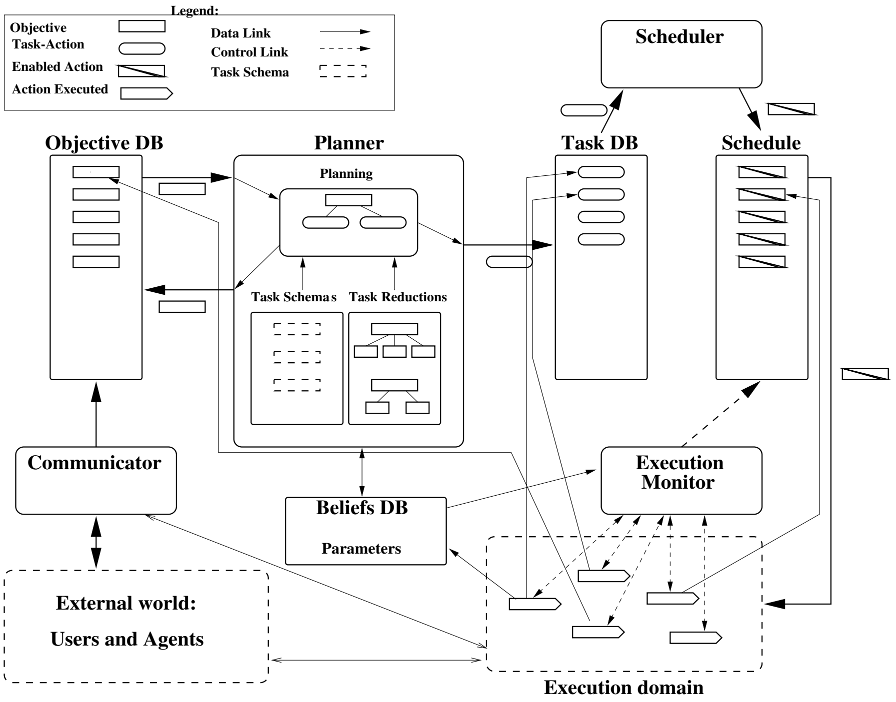

Figure 1: The RETSINA agent internal architecture.

section 3 presents the HITaP planner, the algorithm for interleaving planning and execution, and performance measures. Section 4 shows how the planner supports the agent when dealing with uncertainty and how it supports team behavior and negotiation toward the construction of a plan that is shared by all the agents in the team.

2 The RETSINA Agent Architecture

The RETSINA Agent Architecture [26] provides a framework for the creation of goal directed agents [25, 17] that can autonomously interact and self-organize within a MAS. The architecture of a RETSINA agent is displayed in figure 1.

RETSINA agents are composed of four autonomous modules: a communicator, a planner, a scheduler and an execution monitor. The communicator module receives requests from users or other agents in KQML [8] and transforms these requests into goals. It also sends out requests and replies to other agents.

Requests to an agent are transformed by the communicator into goals (referred to as objectives) for the planner. The agent's objectives are stored in its ObjectiveDB (the repository of goals of the agent), which is implemented as a priority queue. The objective with the highest priority is selected by the planner, which constructs a plan that achieves the goal. To construct its plans, the planner utilizes two data stores: the Task Schema Library and the Task Reduction Library. The Task Schema Library records the classes of tasks that can be accomplished by the agent. The Task Reduction Library describes how complex tasks are achievable via a composition of simpler tasks. In addition, the planner has access to the BeliefsDB, in which the agent stores its domain knowledge, and changes beliefs about the

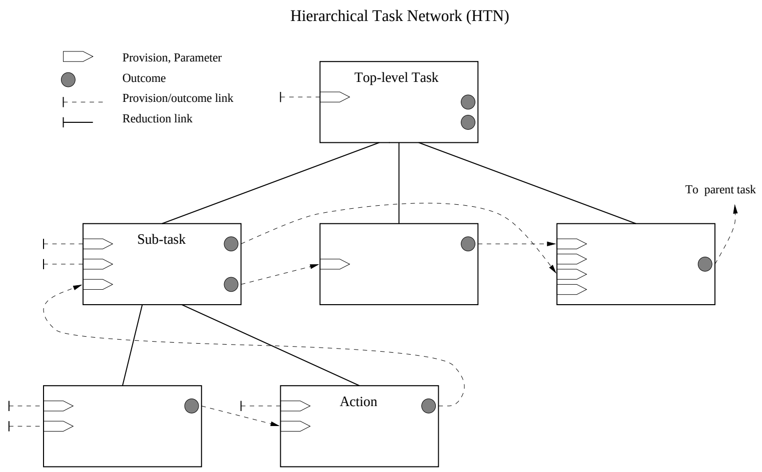

Figure 2: A Hierarchical Task Network

world as actions get executed or dynamic changes occur.

The plan constructed by the planner is stored in the TaskDB, which is an interface between the planner and the scheduler: tasks are added by the planner when they are ready to be executed, and they are removed by the scheduler that decides when to execute them. The resulting schedule is used by the execution monitor that runs the actions and monitors their success or failure.

Executed actions may activate the Communicator to send a reply to the querying agent or messages to other agents, or they can interact with the external world by manipulating it or gathering information. The result is often a change in the BeliefsDB as the agent finds new information or records the effect of some action.

The four modules of a RETSINA agent are implemented as autonomous threads of control to allow concurrent communication, planning and actions' scheduling and execution. Furthermore, actions themselves are executed as separate threads and can run concurrently. In general, concurrency between actions is not virtual. Rather, since some actions are requests of services to other agents running on remote hosts, actual parallelism and asynchrony is enabled.

3 HITaP

HITaP is used as a planning module of RETSINA agents; HITaP represents tasks using the Hierarchical Task Network (HTN) formalism [6]. That is, it plans by task decomposition rather then by satisfaction. Hierarchical planners proceed by reducing the tasks in the plan into simpler subtasks until they are reduced to primitive tasks that can be performed directly by the agent.

Figure 2 displays an example of an HTN. It consists of nodes that represent tasks and two types of edges. The first type of edge is *Reduction links*, they describe the decomposition of a high-level task to subtasks (a tree structure). These edges are used to select the tasks that belong to the decomposition of the parent task. The second type of edge is *Provision/outcome links*—they are used for value propagation between task-nodes. Provision/outcome links describe how the result of one task is propagated to other tasks. For instance in Figure 3, the task T represents the act of buying a product. T decomposes to finding the price (T_1) and performing the transaction (T_2). The reduction requires that T_1 is executed first to propagate the price outcome to T_2.

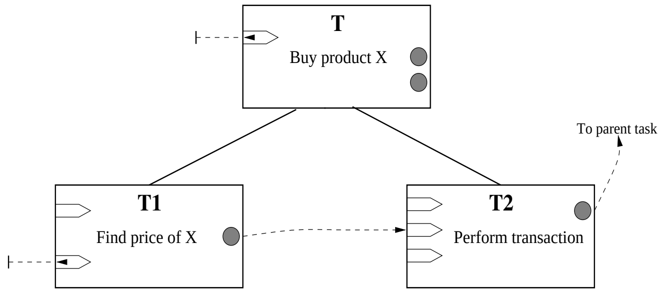

Figure 3: An example of task decomposition

Formally, a planning problem for HITaP is described by the tuple $<\mathcal{A},\mathcal{C},\mathcal{R},\mathcal{B},\mathcal{O},\mathcal{T}>$, where $\mathcal{A}$ is a set of actions (primitive tasks) that the agent can perform directly, $\mathcal{C}$ is a set of complex tasks that are implemented by the composition of actions and other complex tasks. $\mathcal{R}$ is a set of reduction schemas, where each reduction schema provides details on how complex tasks are reduced into simpler tasks which eventually are reduced into actions that the agent can perform. In addition, reduction schemas specify how tasks in the reduction are related to each other through achievement of effects and precondition satisfaction. Since a complex task can be reduced in multiple ways, there may be several reduction schemas in $\mathcal{R}$ for the same task in $\mathcal{C}$. $\mathcal{B}$, the BeliefsDB, contains the set of beliefs of the agent. $\mathcal{B}$ is not a static list—it changes depending on the results of the actions of the agent and the information gathered from the environment or other agents. $\mathcal{O}$ is the ObjectiveDB, which holds objectives not yet achieved by the agent. The goal of the planner is to achieve all the objectives in this list. $\mathcal{T}$ is the TaskDB which, by holding the tasks already added to the plan, describes the plan constructed by the agent.

3.1 Task Representation

A task is formally represented by a tuple $<\mathcal{N},\mathcal{P}_{ar},\mathcal{D}_{par},\mathcal{P}_{ro},\mathcal{O}_{ut},\mathcal{C}_{str},\mathcal{E}>$ where $\mathcal{N}$ is the name of the task; $\mathcal{P}_{ar}$, $\mathcal{D}_{par}$ and $\mathcal{P}_{ro}$ are sets of input conditions whose value affects the behavior of the task. $\mathcal{P}_{ar}$ and $\mathcal{D}_{par}$, called parameters and dynamic parameters, refer to the beliefs of the agent; $\mathcal{P}_{ro}$ is a set of provisions; provisions are used to describe the flow of control within the plan. $\mathcal{O}_{ut}$ is the set of outcomes of the task. Finally, $\mathcal{C}_{str}$ is a set of constraints that describe under which conditions the task can be successfully executed, and $\mathcal{E}$ is a set of estimators used by the planner to predict the outcomes of the task. An example of a task is shown in Figure 4. In that example, $\mathcal{N}=\{\textit{Buy Product}\}$, $\mathcal{P}_{ar}=\{\textit{Cash Expenses}\}$, $\mathcal{P}_{ro}=\{\textit{Enable}\}$, $\mathcal{O}_{ut}=\{\textit{PurchaseDone}\}$, $\mathcal{C}_{str}=\{Cash > \textit{ProductPrice}\}$, and $\mathcal{E}=\{\textit{Cash}=\textit{Cash-ProductPrice}\}$.

Parameters and dynamic parameters refer to the beliefs of the agent; both parameters and dynamic parameters are stored in the BeliefsDB as part of the domain knowledge of the agent. Parameters refer to beliefs on the state of the environment while Dynamic Parameters refer to beliefs on facts in the domain that are modified by the agent executing the plan. For

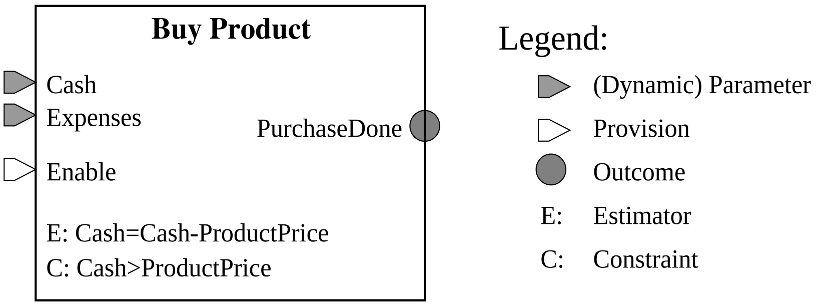

Figure 4: The Buy Product task has one provision (Enable), two parameters (Cash and Expenses), one outcome (PurchaseDone), and one estimator and one constraint (denoted E and C, at the bottom).

example, in a plan which involves moving vehicles, origin and destination are parameters, while the location of the vehicles and the amount of fuel are dynamic parameters since they are modified by the tasks in the plan. The distinction between parameters and dynamic parameters is motivated by the timing in which they are known to the agent: parameters are known at planning time, while dynamic parameters are known only at execution time since they are instantiated by the execution of the tasks in the plan.

Parameters and dynamic parameters are used as input to estimators and constraints. Estimators predict the effects of running a task, whereas constraints are used to verify that some necessary conditions for running an action are met. For instance, in the vehicle relocation example (Figure 5), estimators are used to find out how much fuel is needed to move from the origin to the destination, while a constraint is used to verify that the vehicle has enough fuel to complete the trip. Estimators and Constraints allow the planner to predict the plan's resource consumption and to verify that the agent will have enough resources to execute the plan successfully.

While parameters and Dynamic Parameters store information about the domain, Provision refer to variables that store control flow information. Provisions are used by the agent to enable an action to run, or to propagate the success or failure of a task in the plan. Provisions and dynamic parameters work in combination with outcomes: when an action is executed, its outcomes are established, and their values are transmitted through provision links to the provisions or dynamic parameters of other tasks in the plan, establishing the execution conditions of the subsequent tasks.

Figure 5, shows an example of task reduction that demonstrates how outcomes, provisions, parameters and dynamic parameters work together. The action **Select_Path** has three outcomes: *Completion*, *Path* and *Fail*. The execution of **Select_Path** may lead to two results: if the execution is successful, then outcome *Completion* is set to `true`, while the outcome *Path* is set to the path that the agent decided to follow; otherwise, the outcome *Fail* is set to true and no path is provided. In turn, the value of the outcomes *Completion* and *Path* is tranferred to the provision *Enabled* and to the Dynamic Parameter *Path* in **Computex_Fuel_Consumption**: *Enabled* signal that **Computex_Fuel_Consumption** is ready for execution, while *Path* provides the path on which to compute the fuel consumption. When0 instead, *Fail* is set to true, this value is transferred to the provision *Enabled* in **Send_Sorry_Message** allowing it to be executed.

The distinction between Provisions on one hand and Parameters and

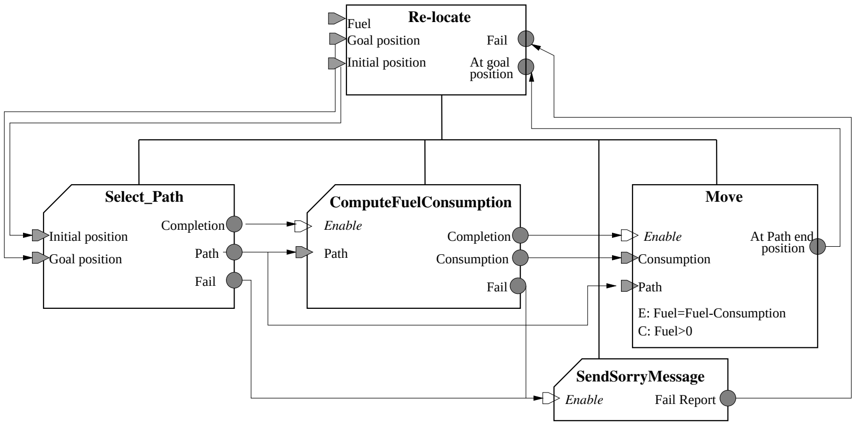

Figure 5: An example of a task reduction schema. Rectangles represent tasks that can be decomposed, while rectangles with a slanted upper left corner represent primitive tasks (actions) that cannot be decomposed any further

Dynamic Parameters on the other is reflected in the different type of information that they represent in the plan. Parameters and Dynamic Parameters record information about the domain and how it is affected by the execution of the plan. This information constitutes the beliefs and the expectations of the agent and it is stored in the BeliefsDB. Provisions instead represent the control flow of the plan to allow the agent the correct selection of next action given the outcome of the tasks executed so far. Control flow information is essential to handle action failures such as queries to agents that abruptly exited the system or that are too busy to reply. In such cases, to just assume that an answer has been received and carry on with the current plan would be totally irrational; rather, the agent needs to know which action to perform next in order to react to the failure and avoid a complete break down of the plan.

Task representation of HITaP shares some similarities to other HTN planners such as O-Plan [4], GraphHTN [14], SHOP and M-SHOP [18]. They all start from the STRIP representation [7] which defines an action on the basis of the inputs it requires and the outputs it produces. In addition HITaP shares with O-Plan and SHOP/M-SHOP the ability to introduce estimation functions and constraints to represent consumable resources. What distinguishes HITaP from other planners is the explicit representation of control flow information in the plan. This allows the introduction, in the plan, of *if* and *case* statements that are typical for programming languages. *Ifs* are actions with two alternative outcomes that allow the execution of different steps. Similarly, *Case* statements are represented by actions with multiple outcomes that trigger the execution of different portions of the plan. Furthermore, as shown below, the combination of action execution while planning and control flow information provides a way to plan for loops.

3.2 Task Reduction Schemas

Task reduction schemas are used to describe the implementation of complex tasks by compositions of other tasks. Reduction schemas are formally represented by the tuple $< \mathcal{N}_{task}, \mathcal{T}_{list}, \mathcal{I}_{links}, \mathcal{P}_{links}, \mathcal{O}_{links} >$. Where $\mathcal{N}_{task}$ is the name of the reduced task t; $\mathcal{T}_{list}$ is a set of primitive and complex tasks that define a method to implement t. $\mathcal{I}_{links}$ contains inheritance links that connect t's provisions to the provisions of the children tasks in $\mathcal{T}_{list}$. These links specify how the values of the provisions of the parent task t become values of the provisions of its children tasks (the members of $\mathcal{T}_{list}$). $\mathcal{P}_{links}$ specifies provision links between sibling tasks in the decomposition. These links are used to maintain a temporal order between tasks in the reduction. $\mathcal{O}_{links}$ is the set of outcome propagation links that connect the outcomes of the children tasks in $\mathcal{T}_{list}$ to the outcomes of the parent task. These links specify the effect of outcomes of the children tasks on the outcomes of their parent task t.

An example of task reduction is displayed in Figure 5. There, $\mathcal{N}_{task}$ is the name of the task **Re-Locate**, $\mathcal{T}_{list}$ is the list of tasks **Select_Path**, **Compute_Fuel_Consumption** and **Move**. $\mathcal{I}_{links}$ are the links that propagate from **Re-Locate** to the subtasks; $\mathcal{P}_{links}$ are the links that connect the subtasks; $\mathcal{O}_{links}$ are the links that propagate from the subtasks back to **Compute_Fuel_Consumption**.

Figure 5 shows how contingencies [22] are represented in the plan: the action `Compute_Fuel_Consumption` can either complete successfully enabling `Move` to execute. Or it can fail, for example by detecting that there is not enough fuel to complete the trip enabling `Send_Sorry_Message` to be executed which issues a failure notification message to the agent that requested the relocation.

3.3 The Planning Process

The planning algorithm is described in Figure 6. It starts from an initial set of plans (*init-plans*) that provide alternative hypotheses of solutions of the original goal. Initial plans are constructed by matching tasks to the initial objectives. The planner proceeds by selecting a partial plan P and an objective o from P's ObjectiveDB, to generate a new partial plan for each possible solution of o. This process is repeated until the planner generates a plan with an empty ObjectiveDB. The planner fails if the list of partial plans empties before a solution plan is found.

```
HITaP (goal)
    init-plans ← make initial plans.
    partial-plans ← init-plan.
    While partial-plans is not empty do:
        choose a partial plan P from partial-plans
        If (P has no objectives)
            then return P
        else do:
            remove an objective o from P's ObjectiveDB.
            partial-plans ← refinements of o in P
    return failure
```

Figure 6: The Basic HITaP Planning Algorithm

The resulting plan is a tree of partially ordered tasks in which the leaf nodes are actions, while the internal nodes are complex tasks. At execution time, actions are scheduled for execution and eventually they are mapped

```
refinements of o in P
    if o is a reduction objective then
       t ← the task corresponding to o
       evaluate estimators and constraints of t
       for each reduction r of t do
          new-plans ← apply r to P
    if o is a suspension objective then
          leave o in the ObjectiveDB of P
          while waiting for an unsuspending event
       new-plans add P
    if o is an execution objective then
       a ← the action corresponding to o
       if a completed successfully
          new-plans add P
       if a failed
          new-plans ← nil
       if a still running
          leave o to the ObjectiveDB of P
          new-plans add P
    Return new-plans
```

Figure 7: The Refinement Algorithm

to methods which in turn are executed by the agent's execution monitor. Complex tasks are used by the scheduler to synchronize the execution of primitive tasks as well as for the propagation of the outcomes of computed tasks to tasks that were not yet executed.

3.4 Establishment of Objectives

The algorithm to solve objectives is shown in Figure 7. The RETSINA Planner allows three different types of objectives: task-reduction objectives, suspension objectives, and execution objectives. Task-reduction objectives are associated with unreduced complex tasks in the TaskDB. They are used to signal which tasks in the current partial plan should be reduced. Once a reduction objective is selected, the planner applies all task reduction schemas associated with the task, generating a new partial plan in correspondence to each application of a schema. As a result, all the subtasks listed in the reduction schema are added to the partial-plan's TaskDB. Task reduction triggers the evaluation of constraints and estimators that are associated with the task being reduced, which in turn could trigger the execution of actions that inspect the environment and provide information that is not present in the BeliefsDB.

Execution objectives are used to monitor the execution of actions while planning. An execution objective is created and added to the ObjectiveDB $\mathcal{O}$ whenever an action is executed. Execution objectives are removed from $\mathcal{O}$ only when the corresponding action terminates. Their solution depends on the termination of the action: if the action terminates successfully, then the objective is simply removed from the list of objectives and no action is taken; otherwise, when the execution fails or times out, the partial plan also fails and the planner backtracks.

Suspension objectives are used to signal that the partial plan contains unreduced complex tasks whose solution depends on data that is not currently available to the agent. Suspension objectives are delayed and transformed into reduction objectives only after the occurrence of an unsuspending event, such as the successful completion of the execution of an action. Unsuspending events provide the data that the planner was waiting for, and they allow the completion of the reduction of the complex task.

3.5 Execution of Actions while Planning

The evaluation of estimators and constraints should be computed before the plan is completed. However, when an estimator needs the value of a dynamic parameter π that is not yet set, the agent can either use its own sensors to find this information or query other agents for the missing information. In either case, the completion of the plan is deferred until the value of π is provided.

The execution of information actions during planning is controlled by the suspension algorithm (Figure 8). Since estimators and constraints are evaluated in the task reduction step, the planner records that the reduction of a task t is suspended by adding a new task-reduction objective o for t, marked as suspended. The objective o records that t is not reduced yet and the completion of the plan is deferred. Then, the planner looks for a primitive task t_π in the plan, that if executed would set π. t_π is found by tracking backward inheritance links and provision links that end in π. The task t_π is then scheduled for execution and a new execution objective e is added to the list of plan P's objectives. The objective e is used to monitor the outcome of t_π

```
suspension of t in P
    o ← task-reduction objective for t
    add o to the ObjectiveDB of P
    set o as suspended
    set unsuspension trigger to a dynamic parameter π
    Find task t_π that sets π
    Schedule t_π for execution
    e ← execution objective for t_π
    add e to the ObjectiveDB of P
```

Figure 8: The Suspension Algorithm

As described above, the objectives o and e are not removed from the list of objectives until t_π completes its execution. The successful completion of t_π sets the dynamic parameter π which removes the suspension on o, which in turns allows t's estimators and constraints to be evaluated and t to be reduced.

The use of suspension and monitoring objectives to control action execution has important consequences. First and foremost, it closely ties action execution and planning because a plan is not completed until all objectives are resolved. The use of suspension and execution objectives guarantees that all scheduled actions are successfully executed before the plan is considered a solution of the problem. In addition, if an executing action fails, the f ailure will be detected as soon as the planner refines the corresponding execution objective. Furthermore, using objectives to suspend and monitor action execution allows the planner to work on other parts of the plan while it waits for the completion of information gathering actions.

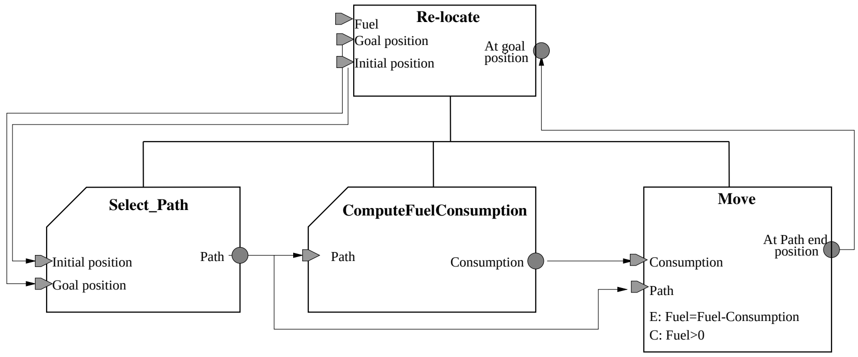

Figure 9: Simplified task reduction schema to relocate

3.5.1 Gathering Information while Planning

In this section we show an example of how the machinery described above allows an agent to gather information while planning. The example is taken from an application of the planner to a command and control problem in which the RETSINA multiagent system [26] is used to support the decision making of three commanders of tank platoons.[3] In this scenario, three army commanders discuss a rendezvous location for their platoons. Then, each commander constructs its own plan, assisted by a missionAgent that finds a route for the commander's platoon taking into account fuel limitations, terrain features and weather conditions.

Each commander asks its missionAgent to find a route to the rendezvous point. MissionAgents transform the request into an objective that is achieved using the reduction schema shown in Figure 9. Following the reduction schema[4], the plan adopted consists of the tasks **Select_Path**, **Compute_Fuel_Consumption**, and **Move**. Since the task **Move** can be further reduced, the planner adds one reduction objective to the ObjectiveDB.

Following the algorithm shown in Figure 7, reduction objectives trigger the evaluation of the estimators associated with the task being reduced. The estimator associated with **Move** depend on the value of the unknown dynamic parameter `Consumption` that can be set only by executing **Compute_Fuel_Consumption**. The execution of a step while planning is controlled by the suspension algorithm described in Figure 8. The planner first suspends the reduction of **Move** until the dynamic parameter `Consumption` is set; then it schedules **Compute_Fuel_Consumption** for execution. Since **Compute_Fuel_Consumption** needs the value of `Path`, **Select_Path** is also scheduled for execution.

The missionAgent executes first the action **Select_Path** using its own path planning capabilities, and it propagates the value of `Path` for the task **Compute_Fuel_Consumption**. Then the missionAgent executes **Compute_Fuel_Consumption** by sending a request for information to an agent that is specialized in estimating fuel consumption. From the point of view of the missionAgent, there is no difference between the two tasks: a request to another agent is an action as any other. As a result of performing the task **Compute_Fuel_Consumption** the value of `Consumption` is estimated and

[3]For more information on the system see http://www.cs.cmu.edu/~softagents/muri.html

[4]The schema in figure 9 is a simplified version of the schema used in figure 5

propagated to **Move**. Finally the agent can compute its estimator of the task **Move** and verify whether there is enough fuel to reach the goal.

3.5.2 Performance Evaluation

To assess the performance of our algorithm for interleaving planning and execution, we tested the planner in two conditions: *informed* and *uninformed*. In the uninformed condition the agent was not provided with sufficient information to complete the plan, hence it had to gather information by running appropriate actions during planning. In the informed condition, the agent was provided with all the information needed to construct its plan, hence it did not need to execute actions for acquiring information during planning. To prevent irrelevant external influence on our measurements, we slightly modified actions that require access to resources external to the agent. In particular, actions that require network access or assistance from other agents may induce non-local delays. We simplified such actions to avoid actual requests to other agents, replacing these by simulated requests, thus maintaining the same plan structure, yet avoiding the external delays. In both conditions we implemented the same problem. The plan the agent constructed to solve the problem included 16 tasks, of which 10 are actions. In the uninformed case, the agent executes 8 actions (for finding missing information) while planning. In the informed case the agent runs no actions during planning.

In both conditions we measured the total time needed by the agent to construct the plan and the user's waiting time: i.e. the time passed between the request and the answer. The latter time includes the planning time, but also the time required to manage the communication on both ends, and the scheduling and execution of the plan. The results reported in table 1 are an average over 10 runs of the planner on the same input. The experiment has been peformed on a Sun Ultra, using Java 1.1.7.

Uninformed Condition		**Informed Condition**	
Planning Time	User Wait Time	Planning Time	User Wait Time
4.65 sec	5.76 sec	0.959 sec	2.656 sec

Table 1: Planning time in the two conditions

Not surprisingly, the results in table 1 shows that interleaving planning and execution slows down the planning process. This is expected, since the planner in the uninformed condition performs all the work done in the informed condition, yet in addition it handles tasks' suspension and execution of actions that acquire information.

The execution of an action is the result of many operations that include the scheduling of the action, the actual time that the action requires to run its own code, the time needed by the Java Virtual Machine to allocate and start a new thread of execution for the action, and the time required to propagate the results of the action. To evaluate the effect of these operations on the total time, we measured three factors: the time consumed by the planner to suspend actions, the time spent waiting for the result of the action, and the time needed for the action to run. The results reported in table 2 are an average over 10 runs of the planner on the same input.

Suspension	Scheduling and Execution	Action Execution Time
0.140sec	3.412 sec	0.467 sec

Table 2: Uninformed Condition: itemized computing time

The results show that the cost of finding which action to suspend is very low, while the real cost of executing actions lays in the recurrent scheduling and monitoring execution. This time includes action execution, which in our case is very low since we used simplified actions for the experiment, but, it is potentially unbound. In conclusion, the algorithms presented in this paper can be efficiently implemented, nevertheless, the agent's operations are slowed down by the recurrent need to schedule actions and monitor their execution. In general, the more actions are scheduled (and the more complicated these actions are) the longer they will take to produce an answer. Ultimately, this delay is traded with a very important gain in the functionalities of the agent.

4 Beyond Information Gathering

The initial motivation for the development of HITaP was the need of gathering information while planning to take advantage of the distribution of knowledge in the MAS. Nevertheless, the suspension algorithm that resulted does not distinguish between information gathering actions and other types of actions. Indeed this is hardly surprising, since the agent itself does not make this distinction either. Therefore, a RETSINA agent using HITaP may run any action while planning, be it an action to gather information or to manipulate the world. Furthermore, while it is widely assumed that information actions do not change the state of the world, they often do: since not all information is free, the agent may have to pay for it, or the agent may have to go somewhere, use resources, or perform some actions to find the information it seeks.

World manipulation while planning introduces a source of uncertainty since changes produced by the actions executed may invalidate the preconditions of other parts of the plan. In the following section we show how HITaP deals with uncertainty by predicting contingencies and replanning when needed. Finally, we will show how HITaP's interleaving of planning and execution can help the agent to negotiate its own activity with the activity of other agents and ultimately to construct a shared plan.

4.1 Handling contingencies in dynamic environments

HITaP deals with uncertainty in the plan in two ways: at planning time it constructs a plan that specifies how to react to possible contingencies, while at execution time, it allows replanning when the plan constructed is no longer valid. In this section we explore these issues in more detail.

Contingency planners [23, 22, 1, 20, 5] are an extension of classical planners that do not assume that the agent has complete knowledge of its world and that the actions it performs have deterministic effects. As an example of problems solved by contingency planners consider the problem of going to a ski resort proposed by Peyot and Smith [22]. The agent is asked to plan a trip to a ski resort called Snowbird (see Figure 10), but the

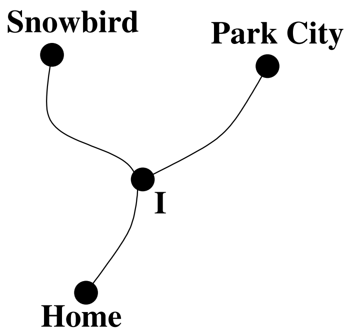

Figure 10: The road map to go to ski

agent does not know whether the road from I to the resort is clear of snow, or closed. Thus, the agent constructs a plan to go to I and there, observe whether the road is open or closed. If the road is open then the agent will proceed to the resort (Snowbird) and if the road is closed it will detour to another resort, e.g., Park City. The plan generated may be similar to the following: go from home to I, observe whether the road from I to Snowbird is open, if it is then proceed, otherwise go to Park City.

HITaP generates contingencies by using provisions and provision links. The operator in figure 9 already contains two contingencies: one to handle the failure of action **SelectPath**, one to handle the success case. The problem of generating alternative plans to go to ski is solved in a similar way by planning to go to I, to observe whether the road is open or closed. If it is open, then enable only actions that lead to Snowbird, otherwise enable only actions that take the agent to Park City. Since provisions control the execution flow of the plan, if the road turns out to be open after all, the agent will ignore the plan to go to Park City; while, if the road is close, it will go to Park City and ignore the plan to Snowbird.

Crucially, contingency planners generate ahead of time the course of actions for all possible contingencies in the plan, which enormously complicates the planning process. The agent is forced to develop a plan for many (possibly unlikely) contingencies, which will not all happen at execution time. A better solution would be to carefully select contingencies to be handled, and to replan when forced to deal with an unplanned contingency.

HITaP solves this problem by following the interleaving of planning and execution approach as discussed in the previous sections. For example, HITaP can generate a plan in which the agent attempts to go to Snowbird; if it finds that the road is closed, it replans and attempts to go to Park City. The operator to produce this plan is shown in figure 11. Here, the observation action does not distinguish between different contingencies, rather it reports its findings to `Move`. The `Move` task can be added to the plan only if the constraint `Status=RoadOpen` is satisfied. Following the suspension algorithm above, the evaluation of the constraint triggers the execution of the observation action and all the actions that establish its preconditions, including the `GoTo`, leads the agent to go to intersection `I` and observe whether the road ahead is open. If the constraint fails, i.e. the road is not open, the planner discards the plan, and backtracks to select another plan that takes the agent to another resort, such as Park City, instead.

HITaP leaves to the programmer the decision on which contingencies should be expanded at plan time, and which should be handled at execution time. The programmer can also implement operators to handle the contin-

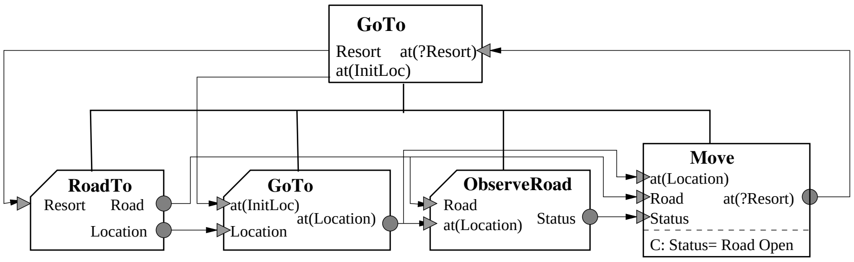

Figure 11: Task structure to go somewhere

gency in both ways, and leave the selection to the heuristic function. The work on Mahinur [19] shows how this selection can be done in a principled way, by considering the probability of the observation outcomes and the utilities of the agent.

Another way planners handle uncertainty is by greedily executing actions as soon as they are planned for. This is the case with reactive planners, such as PRS [9]. Both HITaP and PRS decompose complex tasks into primitive actions, but PRS executes primitive actions as soon as they are inserted in the plan, while HITaP executes selected actions that allows it to complete the plan. Ultimately, the two planners strike a different balance between execution and deliberation, in which HITaP delays the commitment to a plan pushing deliberation as far as possible, while PRS greedily commits to a plan executing its actions as soon as possible.

4.2 Planning a Team Activity

Negotiation is a difficult domain for planners because the agent needs to deliberate to decide what to propose and how to answer to counterproposals, but it cannot construct a complete plan ahead of time since it does not know the reaction of the other agents until it formulates a proposal and receive an answer. Neither classical planners, nor contingency planners can handle negotiation because they are forced to plan ahead of time when there is not enough information to make a proposal. They cannot decide how many rounds of negotiation the agents will need to reach an agreement, or what counterproposals to construct. Reactive planners, on the other hand, are inadequate because they do not allow the agent to perform enough deliberation while planning, as a consequence the agent may settle for a suboptimal agreement just because it did not think enough. We claim that HITaP can handle negotiation because it strikes a better tradeoff between execution and deliberation. HITaP allows the agent to deliberate on the proposals to make and how to answer to the counterproposals, but when a decision is reached, HITaP executes to communicate and waits for the answer(s).

We applied the HITaP to the planning of a joint activity of three army platoons controlled by agents in a command and control exercise. The objective of the agents is to move the platoons from their initial location to a final destination through a territory in which enemy movements have been reported. When likely to encounter enemies, the platoons need to support each other to present a unified front to the enemies and minimize their losses. The agents negotiate the position of the platoons, i.e. which platoon covers the left side, which the right side and which one stays in the middle ready to assist the other platoons when they are in trouble. In addition, the agents decide which route each platoon will follow, and how to monitor each other's progress so that effective assistance can be given in case of attack.

Theories of joint activity [3, 11] stress the importance of the development of joint intentions among the teammates. Joint intentions are used to create a shared plan that satisfies the overall goals of the team, and to guide each agent in the planning of its own activity within the team. Nevertheless, these theories do not explain how joint intentions are created in the first place, nor do they describe the process that brings these joint intentions about.

Addressing this challenge, we structured the team activity for the platoon example into four phases: first, the agents decide which position each of them covers (left, center, or right); second, they decide which route to follow; third, they decide how to monitor each other's progress so as to maintain coordination during execution; fourth, the agents execute plans while monitoring each other's progress. To form joint intentions, the agents negotiate until they reach an agreement.

The results of each phase of the joint activity is used to make a decision in the following phases. For example, if an agent decides that its platoon will take to the right, then it is committed to plan for a route in that area, it cannot move to the center or to the left without negotiating its change of commitment. Similarly, the decision on the route affects the position of the checkpoints used by the agent while moving. Because of this strict interconnection between the different parts of the activity, each agent needs to construct a unified plan which incorporates all four phases. Such a plan would automatically record the rationale of the decisions taken, so that the agent does not loose track of the reason behind the choices it made, and if a problem arises the agent could find an alternative plan or backtrack onto previous choices to construct a different plan.

The agents follow a prescribed negotiation protocol while planning their own activity as displayed in Figure 12.

Negotiation Protocol

- Make a decision
- Propose the decision to the teammates
- Wait for the teammates proposals
- Any agreement is reached?
  - If yes, then output the decision made
  - If no, then follow another round of negotiation

Figure 12: The negotiation protocol used to achieve mutual agreement

The protocol dictates that to reach an agreement, the agents first make a decision on the matter, then propose their decision to their teammates, wait for the teammates proposals, and verify whether any agreement has been reached. If there is no agreement, the agents proceed with another round of negotiation. Crucially, in order to decide whether to continue the negotiation, the agent needs to have executed the previous three steps, because it cannot decide whether there is agreement without hearing from the teammates, and it cannot hear without proposing, this in turn cannot be done without making a decision on what to propose. This process continues

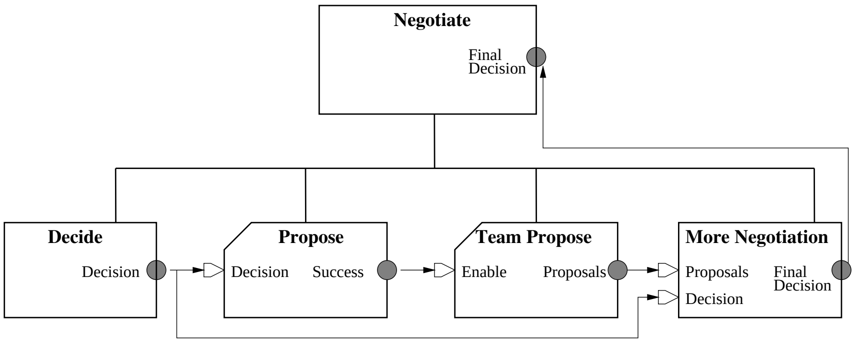

Figure 13: Task reduction schema that implements the negotiation protocol

until an agreement is reached or all the alternatives have been explored.

The negotiation process shown in Figure 12 is the mechanism that allows all the agents in the team to achieve mutual agreement on the joint activity that they perform. As seen above, this negotiation process makes use of the interleaving of planning and execution. The agent uses deliberative planning to make the most informed decision on the plan to follow, but it has to execute actions in the plan to communicate with the teammates and decide whether any agreement has been reached. Any agent that does not interleave planning and execution could neither propose to the teammates its own decisions, nor record the decisions of the teammates, nor detect when agreement has been reached and move on to other phases of the joint activity.

The negotiation protocol above has been implemented in HITaP. Figure 13 shows the task reduction structure used to manage the negotiation protocol. The first three tasks correspond to the three initial steps of the negotiation protocol, while the fourth step is used to decide whether to continue the negotiation or not.

The task structures for the task `MoreNegotiation` are displayed in Figure 14. When there is agreement between the teammates, the first decomposition is used, whereas in the case of no agreement the second decomposition is used. The constraints in the actions in the decomposition trigger the execution of the tasks.

The constraints `agree(Proposal,Decision)` determine whether an agreement has been reached among the agents. In order to establish this constraint the agent needs to formulate its own proposal and gather the decision of the teammates. This information is collected by running the subtasks of the Negotiate operator shown in Figure 13. The agent therefore decides what its proposal is, proposes it to the teammates, gathers their feedbacks and finally evaluates the constraint to find out whether there is any agreement with its teammates. If no agreement is found, the agent proceeds to another round of negotiation.

The STEAM system [16] and TOPI [24] also implement systems in which agents act in a team to achieve a common goal. In contrast to the example above, agents in both STEAM and TOPI are *handed* a role and a plan to follow, and their collaboration is restricted to the execution of that plan. The research performed with HITaP attempts to overcome these limitations by leaving the construction of a shared and distributed team plan to the agents. Thus, the agents themselves decide which roles to fill and which plans to follow.

This example also shows a very interesting side effect of HITaP: the im-

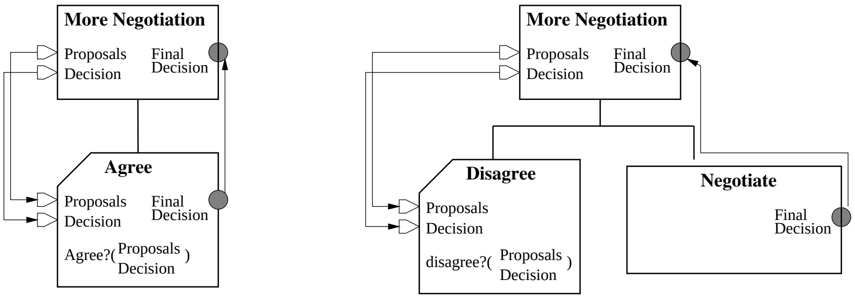

Figure 14: Task reduction schema to decide whether to stop the negotiation

plementation of loops as tail recursive functions. Notice that the negotiation protocol shown above (in Figure 12) is expressed as a tail recursive function. Its implementation in HITaP through the operators Negotiate and More Negotiation implements exactly the same function. This observation leads us to conclude how loops can be implemented via interleaving of planning and execution: any loop that repeats a sequence of steps, step1 to stepN, until a stop condition is reached, can be expressed by an operator that first plans for the sequence step1 to stepN, and then forces their execution by checking the validity of the stop condition. This process is repeated until the stop condition is satisfied.

5 Conclusion

Planners that deliberate but do not allow execution while planning are not of much use for agents in multiagent systems because they do not allow flexible interaction with other agents or with the domain while the agent plans locally. The lack of this interaction prevents the agents from gathering important information that is essential to the construction of the plan. Reactive planners suffer from the converse problem: they do not allow the agent to deliberate enough on its own decisions which may lead to gross suboptimality and conflict in the interaction with other agents.

HITaP strikes a balance between deliberative only planners and reactive planners. HITaP deliberates on the actions to take until it realizes that it needs to gather information or to contact other agents in the system; at which point, HITaP executes selected actions in the plan to acquire the needed information and interact with other agents. This interleaving of planning and execution allows agents that implement HITaP to collaborate as teammates in the construction of shared and distributed plans.

The HITaP planner is a first step towards a distributed planning scheme based on a peer to peer cooperation between agents that is not based on hierarchy or control relationships. We call this type of cooperation *capability-based*. In this respect, our approach is very different from the planning architecture proposed in [28], where the planning process is centralized, but the execution is distributed.

The resulting collaborative process is consistent with the theoretical framework layed out by Grosz et al [11] and Cohen et al [3], which concentrates on the commitment of the agent to the team activity as a means to achieving the team goal. In this respect, the private intentions of the agent depend on its intention to achieve the team goal and on the agent's commitment to play its own role in the overall team activity. The HITaP planner provides a way to implement these ideas while tackling problems that are not faced in the theoretical works: namely how is the agreement achieved and how the intentions are formed in the first place.

References

[1] Jim Blythe and Manuela Veloso. Analogical replay for efficient conditional planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 668–673, Providence, RI, 1997.

[2] Jamie G Carbonell and the PRODIGY Research Group. Prodigy 4.0:the manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, 1992.

[3] Philip R. Cohen and Hector Levesque. Teamwork. *Noûs*, 25(4):487–512, 1991.

[4] Ken Currie and Austin Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[5] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering. In *Proceedings of AIPS-94*, pages 31–36, 1994.

[6] Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of AAAI94*, Seattle, 1994.

[7] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5:189–208, 1971.

[8] Tim Finin, Yannis Labrou, and James Mayfield. Kqml as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[9] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI87*, pages 677–682, Seattle, WA, 1987.

[10] Barbara J. Grosz. Collaborative systems. *AI Magazine*, 17(2):67–85, Summer 1996.

[11] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artifical Intelligence*, 86:269–357, 1996.

[12] Kautz H. and Selman B. Blackbox: A new approach to the application of theorem proving to problem solving. In *Proceedings of the Workshop Planning as Combinatorial Search, AIPS-98*. AAAI Press, 1998.

[13] Amnon Lotem, Dana S. Nau, and James A. Hendler. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[14] Amnon Lotem, Dana S. Nau, and James A. Hendler. Using planning graphs for solving htn planning problems. In *AAAI99*. AAAI Press, 1999.

[15] David Martin, Adam Cheyer, and Douglas Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):92–128, 1999.

[16] Tambe Milind. Towards flexible teamwork. *Journal of Artifical Intelligence Research*, 7:83–124, 1997.

[17] Jörg P. Müller. *The Design of Intelligent Agents.* Springer, 1996.

[18] D.S. Nau, Y. Cao, A. Lotem, and H. Muoz-Avila. Shop and m-shop: Planning with ordered task decomposition. Technical Report CS TR 4157, University of Maryland College Park, MD, 2000.

[19] Nulufer Onder and Martha Pollack. Contingency selection in plan generation. In *ECP-97*, 1997.

[20] Nulufer Onder, Martha Pollack, and John F. Horty. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *AAAI-99*, 1999.

[21] J. Scott Penberthy and Daniel Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 103–114, Cambridge, MA, 1992.

[22] Mark Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of AIPS-92*, pages 189–197, College Park, MD, 1992.

[23] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Resarch*, 4:287–339, 1996.

[24] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward team-oriented programming. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.

[25] Anand S. Rao and Michael P. Georgeff. Modelling rational agents within a bdi-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, 1991.

[26] Katia Sycara, Keith Decker, Anadeep Pannu, Mike Williamson, and Dajun Zeng. Distributed intelligent agents. *IEEE Expert, Intelligent Systems and their Applications*, 11(6):36–45, 1996.

[27] Katia Sycara, Massimo Paolucci, Martin van Velsen, and Joseph Giampapa. The retsina mas infrastructure. Submitted for publication; available at http://www.cs.cmu.edu/ softagents/publications.html.

[28] David E. Wilkins and Karen L. Mayers. A multiagent planning architecture. In *Proceedings of AIPS-98*, 1998.