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

February 5, 2000

Abstract

Agents in a multiagent system may 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, as provided by HITaP, plays a crucial role in an agent's ability 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 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 Introduction

In a multiagent system, it is impractical to maintain the classical distinction between planning and execution, in which an agent first constructs a plan and

[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 Josepth 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.

at a later time executes all the actions in the plan. The first problem with this schema is that multiagent systems are intrinsically dynamic: what an agent assumes to be true, may become false as a consequence of the actions of the other agents in the system. By following the rigid sequence between planning and execution the agent runs the risk of constructing a plan that is not valid because changes in the domain invalidate some preconditions.

An additional problem of the rigid sequence of planning and execution is that because of the distribution of knowledge and capabilities among agents in the system, one single agent cannot decide on the future course of action without exchanging information with the other agents in the system. The agent therefore should stop its planning, and query agents that can provide the needed information. Furthermore, if the system is deployed in a real world application, it may be the case that no agent has the necessary information, which can only be gathered by direct inspection of the world.

Exchange of information is just an instance of a more general problem: the agent may find out that, to achieve its goals, it needs the support of other agents. The agent then either subcontracts a task to agents that can provide it, or the agent forms an agent team and develops a joint plan with its teammates to overcome the problem in a mutually supportive way [2, 7]. Independently of whether the agent gathers information, subcontracts a task, or forms a team, it needs to interrupt the planning and interleave the execution of actions in the plan.

In this paper we present HITaP,[2] a planner that interleaves planning and execution within the general framework of HTN planning[3]. An agent develops its own plan until it detects that it needs some information, or that it needs to synchronize with other agents, at which point, the agent suspends planning and executes selected actions in the plan. The executed actions lead the agent to gather the needed information, or to complete the task that it could not perform otherwise.

The paper is organized as follow: we first introduce the internal architecture of RETSINA agents and the role of the planning component; then we present the HITaP planner and the algorithm for interleaving planning and execution, followed by two examples of application of the planner to problems of information gathering and team coordination in which interleaving of planning and execution plays an essential role. We conclude with differentiation from related work and a summary of the contributions of this paper.

2 The RETSINA Architecture

RETSINA is an open multi-agent system that provides infrastructure for different types of deliberative, goal directed agents. In this sense, the architecture of a RETSINA agent [17] exhibits some of the ideas of BDI agents [16, 12]. The architecture of a RETSINA agent is displayed in figure 1.

[2]Hierarchical Task network Planner

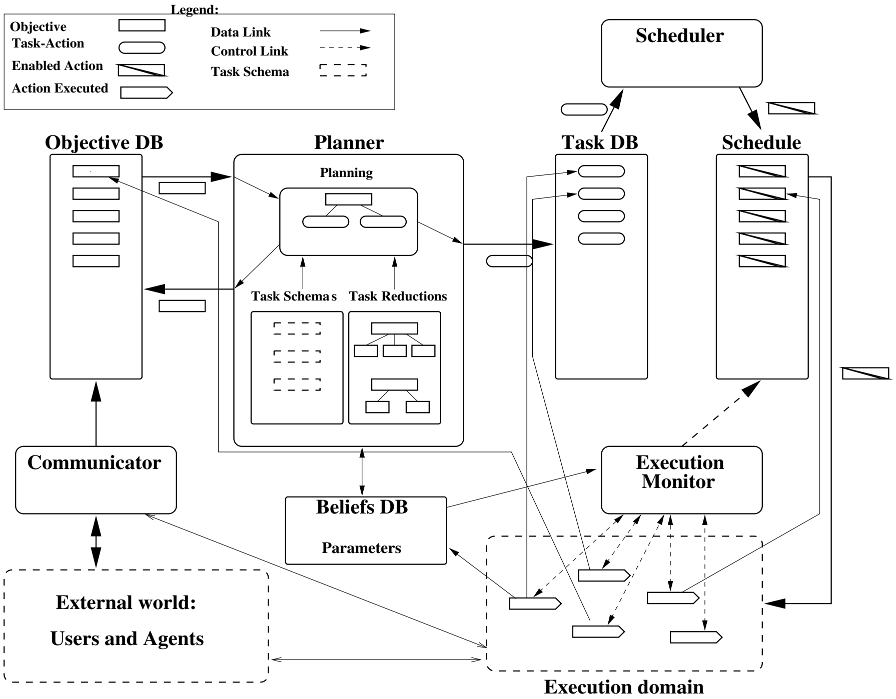

Figure 1: The RETSINA agent internal architecture.

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 format [4] and transforms these requests into goals. It also sends out requests and replies to other agents.

Requests to the agent are transformed by the communicator into goals (referred to as objectives) for the planner. The agent's objectives are stored in its ObjectiveDB, 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 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

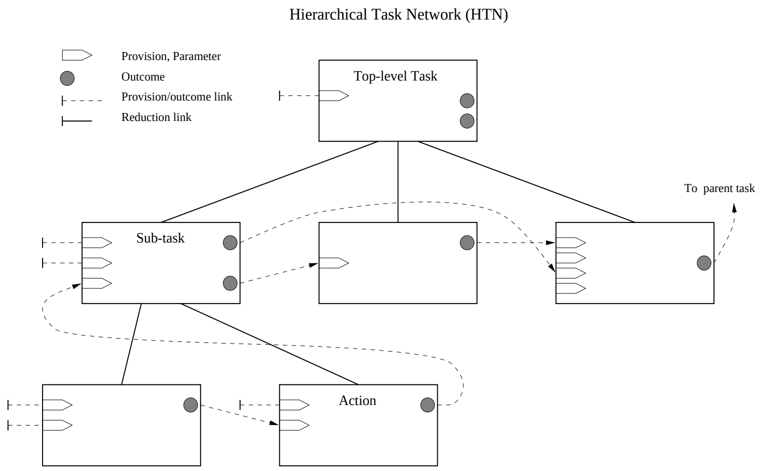

Figure 2: A Hierarchical Task Network

monitor that runs the actions and monitors their success and failure.

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 is enabled.

3 The Planner Module

HITaP is used as a planning module of RETSINA agents; HITaP represents tasks using the Hierarchical Task Network (HTN) formalism [3]. Figure 2 displays an example of an HTN. It consists of nodes that represent tasks and two types of edges. The first type of edges are *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 edges are *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 may decompose 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.

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

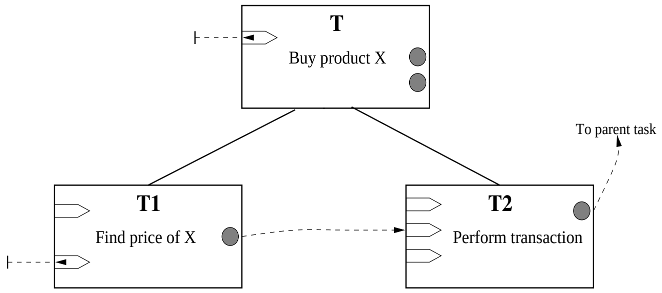

Figure 3: An example of task decomposition

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 compex task can be reduced in multiple ways, there may be several reduction schemas in $\mathcal{R}$ for the same task in $\mathcal{C}$. $\mathcal{B}$ is the BeliefsDb, it 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 represented formally by a tuple $< \mathcal{N}, \mathcal{P}_{ar}, \mathcal{D}_{par}, \mathcal{P}_{ro}, \mathcal{O}_{ut}, \mathcal{C}, \mathcal{E} >$ 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, provision are used to describe the flow of control within the plan. $\mathcal{O}_{ut}$ is the set of outcomes of the task. Finally, $\mathcal{C}$ 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 this example, $\mathcal{N}= \{$*Buy Product*$\}$, $\mathcal{P}_{ar} =\{$*Cash Expenses*$\}$, $\mathcal{P}_{ro} = \{$*Enable*$\}$, $\mathcal{O}_{ut} =\{$*PurchaseDone*$\}$, $\mathcal{C}=\{Cash > ProductPrice\}$, and $\mathcal{E} =\{Cash{=}Cash\text{-}ProductPrice\}$.

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 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

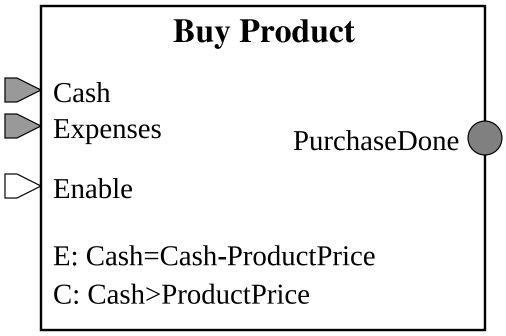

Figure 4: The Buy Product task has a provision and a parameter (on the left), an outcome (on the right), and an estimator and a constraint (denoted E and C, at the bottom).

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 above (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, in fact, the vehicle controlled by the agent has enough fuel to complete the trip. Provisions do not carry information about the domain, rather they 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 combined 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 latter tasks.

Figure 5, shows an example of how outcomes, provisions, parameters and dynamic parameters work together. The task **Select_Path** has three outcomes: *Completion*, *Path* and *Fail*. The outcomes *Completion* and *Path* are set when the execution of the action is successful, in such a case the provision *Enabled* in **Ask Fuel Consumption** is set, while the value of *Path* is propagated to the dynamic parameter *Path* in **Compute_Fuel_Consumption**. Instead, when the task **Select_Path** fails, the outcome *Fail* is set, in such a case **Send_Sorry_Message** is enabled, while **Compute_Fuel_Consumption** is not. The same holds for the outcomes of **Compute_Fuel_Consumption**: if the action succeeds, the outcomes *Completion* and *Consumption* are set and their values are propagated to the **Move** task. When **Compute_Fuel_Consumption** fails, the outcome

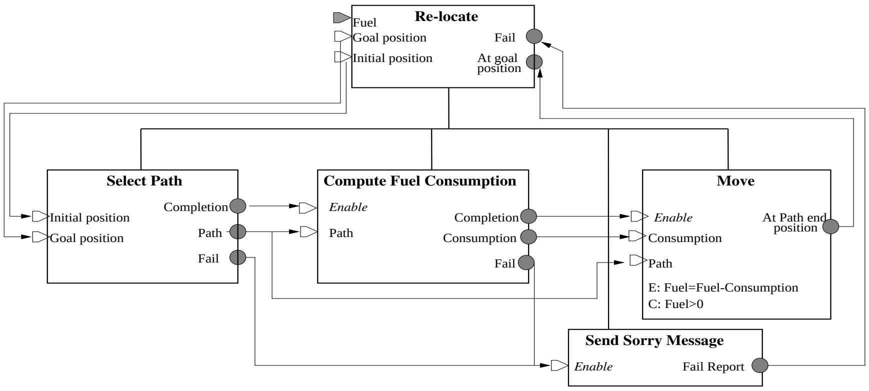

Figure 5: An example of a task reduction schema

Fail is established and **Send_Sorry_Message** is enabled.

The distinction between provisions and parameters allows a natural representation of contingencies in the plan [14]. At planning time the agent can predict possible outcomes of its own actions and construct a different thread of execution for each one of them. For example: the agent could have a plan to go to a store to by some products. Since the store might not have what the agent needs, then the agent can add to the plan a rule that says that if it cannot find the products in the first store, then it should check also in another store. Contingencies are represented in HITaP by generating different flows of control in the plan. For instance, in the example above, the agent predicts that one of the actions in the plan might fail, and therefore it prepares for such a contingency by planning to send a notification of failure (Send_Sorry_Message) to the requesting agent.

3.2 Task Reduction Schemas

Task reduction schemas are used to describe the implementation of complex tasks by compositions of other tasks. The tuple $< \mathcal{N}_{task}, \mathcal{T}_{list}, \mathcal{I}_{links}, \mathcal{P}_{links}, \mathcal{O}_{links} >$ formally represents a reduction schema. $\mathcal{N}_{task}$ 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 task **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**.

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 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

```
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

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.

4 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 failure will be detected as soon as the planner refines the corresponding execution objective. Furthermore,

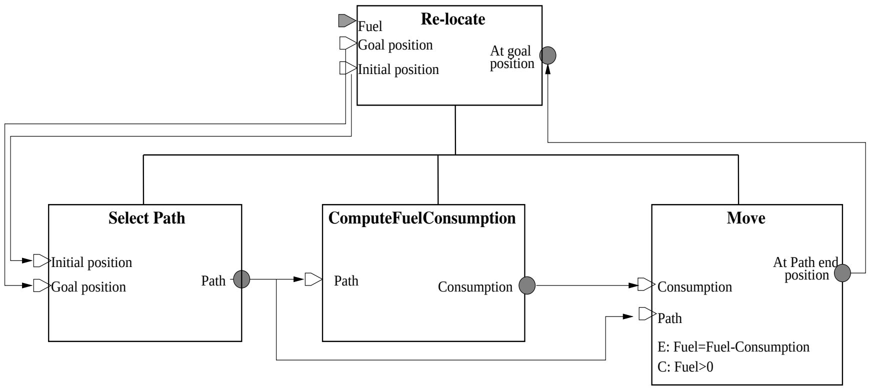

Figure 9: An example of a task reduction schema

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.

5 Planning while Gathering Information

The initial motivation for the HITaP was the need to gather information while planning, because, since we do not make the closed world assumption, the agent would have not been able to make an intelligent decision on which plan to adopt. In this section we show an example of how the machinery explained above allows the agent to gather the needed information. The example is taken from an application of the planner to a command and control problem in which the RETSINA multiagent system [17] 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 to an objective that is achieved using the reduction schema shown in Figure 9. Following the reduction schema[4], the plan adopted is **Select_Path**, **Compute_Fuel_Consumption**, and **Move**. Since the three actions can be further reduced, the planner adds three reduction objectives 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** depends on the value of the unknown dynamic param-

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

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

eter `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 task **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 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.

To assess the performance of our interleaving planning and execution algorithm, 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, from the time of tasking the agent to the time of receiving results. The results reported in table 1 are an average over 10 runs of the planner on the same input.

Uninformed Condition		**Informed Condition**	
Planning Time	Waiting Time	Planning Time	Waiting 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 Java 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 shows 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.

The results of the experiment suggest that while the time needed to find which action to suspend is very low and therefore the algorithms presented in this paper can be efficiently implemented, 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 it will take to produce an answer. Nevertheless, this delay is traded with a very important gain in the functionalities of the agent.

6 Planning a Team Activity

The commanders in the example described in the previous section managed the coordination directly, while the planning agents were used only to find a route to the rendezvous point. In a second experiment, we applied the HITaP to the planning of a joint activity of the three army commanders in their command and control exercise. The objective of the commanders is again to move their 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 commanders need to support each other to present a unified front to the enemies and minimize their losses.

The planning agents support the commanders by negotiating the role of each platoon, i.e. whether a given platoon will cover the left, or the right side or it will go 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 on joint activity [2, 8] stress the importance of the development of joint intentions among the teammates. Joint intentions are used to create a shared plan that satisfy 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.

We structured the team activity into four phases: first the agents decide which role each of them will play in the overall team activity; second, they will decide which route to follow; third, they will decide how to monitor each other's progress so as to maintain coordination during execution and finally the agents will execute following the route planned while monitoring each other's progress. To for joint intentions, the agents need to negotiate until they reach an agreement. The agents first negotiate which role to take in the overall activity (left,center or right); then they negotiate the routes that they are going to follow, and whether they are going to support each other; and finally, they negotiate how to monitor their progress toward the achievement of the goal.

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 has to plan for a route in that area, it cannot move on the center or on the left without changing its 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 lose 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 that is displayed in Figure 10.

Following the negotiation protocol, 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. The problem of this protocol is that 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

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 10: The negotiation protocol used to achieve mutual agreement

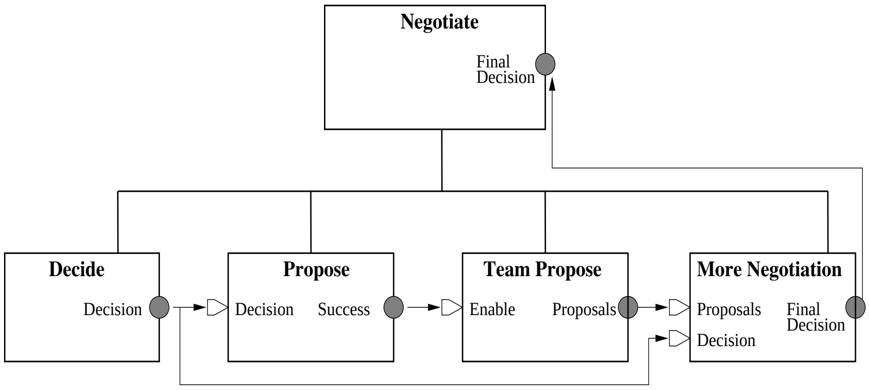

Figure 11: Task reduction schema that implement the negotiation protocol

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

The negotiation process shown in Figure 10 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 a crucial 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 then 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 11 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 12. When there is agreement between the teammates, the first decomposition

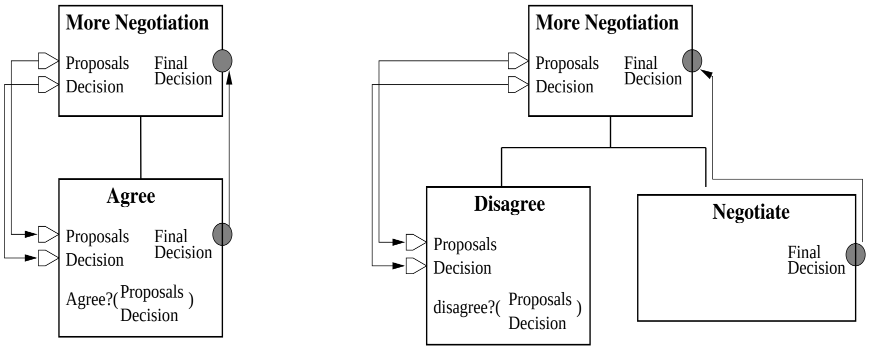

Figure 12: Task reduction schema to decide whether to stop the negotiation

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.

7 Related Work

HITaP has some similarities with Knoblock's *Sage* [9], mainly in the concurrency of planning and information gathering and the close connection between the planner and the execution monitor through monitoring objectives. Nevertheless, the two planners differ in many important respects: while *Sage* is a partial order planner that extends UCPOP [13], HITaP plans by task reduction rather than from first principles; in addition, HITaP extends *Sage*'s functionalities through the constant monitoring of the information gathered and replanning when needed. Other planners relax STRIPS' omniscience assumption by interleaving planning and execution of information gathering actions, e.g., *XII*[6]. Our approach is different from theirs. First, as in the case of *Sage*, we use a different planning paradigm: HTN instead of SNLP. In addition we cannot assume the Local Close World Assumption because the information gathered might change while planning.

ConGolog [1] extends the expressivity of HTN planning to constructs like loops and if-structures and sequences while dealing with incomplete knowledge. HITaP achieves the same expressivity with a pure HTN approach: loops [19] are planned for and used to implement monitoring actions. The negotiation protocol in figure 10 shows how loops can be constructed with the help of interleaving of planning and execution. The agent there follows a tail recursive procedure that simulates a four steps loop: decide, propose and wait, until an agreement is reached. HITaP implements if-structures' through provision satisfaction as shown above in the discussion about contingencies.

HITaP bears also some 'similarities to PRS [5]. Both planners decompose abstract tasks into primitive actions. Yet, they follow different planning algorithms: PRS is a reactive planner, while HITaP executes selected actions while still planning. For this reason the two planners strike a different balance between execution and deliberation. The negotiation task proves to be challenging for reactive planners. During negotiation, the agent needs to deliberate on what is the best proposal to make to its teammates. This deliberation requires the analysis of different proposals and the forecast of the consequences of a decision. Such forecast is difficult, if not impossible, for reactive planners [10]. For instance, when choosing which role to use, a deliberative planner can analyze the different alternatives, but a reactive agent would commit to the first alternative, without analyzing any of the others.

In conclusion, neither deliberative nor reactive planners can handle negotiation properly. Deliberative planners cannot break the neat sequence of planning and execution, hence they are unable to plan a negotiation process. Reactive planners suffer from the complementary problem, namely, they cannot be uncommitted while investigating different hypotheses. HITaP strikes a balance between reaction and deliberation: it deliberates as necessary when decision making is required, but it behaves as a reactive planner when it follows a precise protocol to communicate with its teammates.

The HITaP 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 [18], where the planning process is centralized, but the execution distributed.

The resulting collaborative process is consistent with the theoretical framework layed out by Grosz et al [8] and Cohen et al [2], 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.

Other implementation work on cooperation bears some similarities with the work presented here. The STEAM system [11] and TOPI [15] are implemented systems in which agents cooperate to achieve a common goal. Agents in both STEAM and TOPI are *handed* a role and a plan to follow, and they execute the plan in a collaborative way. The research performed with HITaP attempts to overcome these limitations by leaving to the agents the construction of a shared and distributed team plan in which the agents decide which roles to fill and which plans to follow.

8 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 counter 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.

References

[1] Chitta Baral and Tran Cao Son. Extending ConGolog to allow partial ordering. 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.

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

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

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

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

[6] Keith Golden, Oren Etzioni, and Daniel Weld. Planning with execution and incomplete information. Technical Report UW-CSE-96-01-09, Department of Computer Science and Engeneering, University of Washington, 1996.

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

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

[9] Craig A. Knoblock. Planning, executing, sensing and replanning for information gathering. In *Proceedings of IJCAI95*, 1995.

[10] Jaeho Lee and Suk I. Yoo. Reactive-system approaches to agent architectures. 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.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] Mike Williamson, Keith Decker, and Katia Sycara. Executing decision-theoretic plans in multi-agent evnironments. In *Plan Execution Problems and Issues*, 1996. Also appears as AAAI Tech Report FS-96-01.