A Planning Component for RETSINA Agents

M. Paolucci, D. Kalp, A. Pannu, O. Shehory, K. Sycara
The Robotics Institute
Carnegie Mellon University
5000 Forbes ave
Pittsburgh, PA 15213
{paolucci,kalp,pannu,onn,katia}@cs.cmu.edu

Abstract. In the RETSINA multi-agent system, each agent is provided with an internal planning component—the RETSINA planner. Each agent, using its internal planner, formulates detailed plans and executes them to achieve local and global goals. Knowledge of the domain is distributed among the agents, therefore each agent has only partial knowledge of the state of the world. Furthermore, the domain changes dynamically, therefore the knowledge available might become obsolete.

To deal with these issues, each agent's planner allows it to interleave planning and execution of information gathering actions, to overcome its partial knowledge of the domain and acquire information needed to complete and execute its plans. Information necessary for an agent's local plan can be acquired through cooperation by the local planner firing queries to other agents and monitoring for their results. In addition, the local planner deals with the dynamism of the domain by monitoring it to detect changes that can affect plan construction and execution. Teams of agents, each of which incorporates a local RETSINA planner have been implemented. These agents cooperate to solve problems in different domains that range from portfolio management to command and control decision support systems[1].

1 Introduction

We are developing the RETSINA[2] Multi-Agent System (MAS) [15] in which multiple agents receive goals from users and agents. Since RETSINA implementations are deployed in real-world, distributed, open environments, the state of the world may dynamically change or might be only partially known to agents in the system. This may result from either actual changes in the world or limited and incoherent knowledge of agents as a result of their distribution across the network and limited resources and expertise of each individual agent.

To satisfy their goals, the agents need to formulate detailed plans and execute these plans. However agent autonomy, distribution and limited information usually prohibit the creation of a global, comprehensive plan for the whole agent system. Therefore, as we suggest, each agent must be provided with a planning component as part of its internal architecture. Yet local and distributed planning brings about other problems which

[1] This research has been sponsored in part by the office of Naval Research grant N-00014-96-16-1-1222 by DARPA grant F-30602-98-2-0138

[2] REusable Task Structure based Intelligent Network Agents.

require resolution: an agent's local plans, once found conflicting with other agents' local plans, must be revised and re-planned for. Moreover, in MAS, the computation of an agent's local plan partly relies on other agents performing parts of the plan. This implies that agent i's local planning (and execution) may require that agent i suspends its planning while waiting for other agents to complete their plans and provide results which are preconditions to the rest of i's plan. Agent i should resume planning once these results arrive.

These unique requirements of planning within an open dynamic MAS (and in particular in RETSINA) pose difficulties in the use of existing planners. Although research on planning has dealt with most of the problems listed above, no planner addresses all the problems at once. Planners deal with partial knowledge by either planning for contingencies (e.g., [12]) or gathering information during planning (e.g., [7, 6]). Other planners deal with uncertainty in the domain by using probabilistic models [8] or by reacting to the environment in which they operate [5]. Open, dynamic multi-agent systems require that both partial knowledge and dynamism be resolved simultaneously.

We have developed a new planner as part of the internal architecture of RETSINA agents. This planner assumes that agents have only partial knowledge of the domain but it allows agents to gather information either by direct inspection of the domain or by querying other agents. In addition, the planner deals with dynamism in the domain by monitoring changes in the environment and predicting their effect on the plan.

Agents in a multi-agent system can take advantage of the collaboration of other agents in the system. Single agent planning techniques obviously cannot exploit these opportunities. In our approach, multiple agents exploit the intrinsic parallelism in the multi-agent system in two ways: (1) by having multiple agents working on accomplishing a common goal and locally planning for it; (2) by each local planner working on other parts of its plan (or on other partial plans) while waiting for other agents to compute requested information.

The RETSINA planner is a novel combination of existing planning methods. Agents that incorporate it as part of their internal architecture can interleave planning, execution and replanning in a dynamically changing environment despite having only partial knowledge of the domain. This functionality is supported by direct monitoring of the environment and cooperation with other agents.

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 RETSINA agents [15] exhibits some of the ideas of BDI agents [13, 10]. RETSINA agents are composed of four autonomous functional modules: a communicator, a planner, a scheduler and an execution monitor. The communicator module receives requests from users or other agents in KQML format and transforms these requests into goals. It also sends out requests and replies. The planner module transforms goals into plans that solve those goals. Executable actions in the plans are scheduled for execution by the scheduler module. Execution of the actions and monitoring of this execution is performed by the execution monitor module. The four modules of a RETSINA agent are implemented

as autonomous threads of control to allow concurrent planning and actions' scheduling and execution. Furthermore, actions are also executed as separate threads and can run concurrently. In general, concurrency between actions is not virtual. Rather, since some actions require that the agent ask other agents for services, and since these agents are running on remote hosts, actual parallelism is enabled.

The following data stores are part of the architecture of each individual RETSINA agent and are used by the RETSINA planner. Their role in the overall architecture of the agent is displayed in Figure 1.

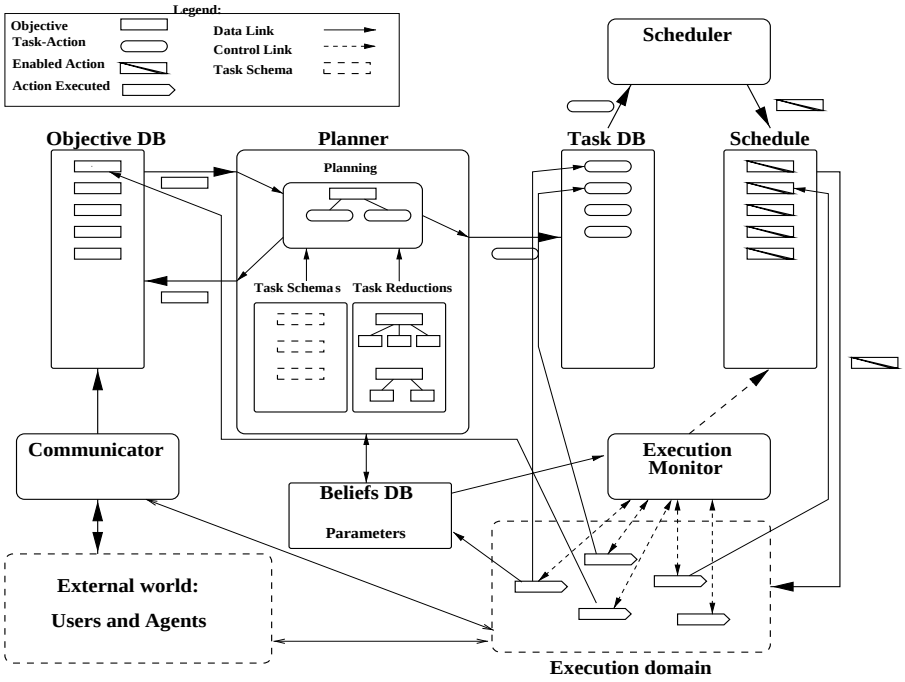

Fig. 1. The RETSINA planning architecture.

- The *objective-DB* is a dynamic store that holds the objectives of the agent of which it is a component. An objective-DB implements a queue with priorities, i.e., the objective with the highest priority on the queue is handled first by the planner. New objectives are inserted in the queue by the communicator and by the planner when complex objectives are decomposed in simpler objectives.
- The *task-DB* is a dynamic data store that holds the plan. Tasks are added by the planner when it recognizes that they contribute to the achievement of the objectives. Tasks are removed by the scheduler when they are ready for execution.
- The *task schema library* is a static data store that holds tasks schemas. These are used by the planner for task instantiation.

– The *task reduction library* is a static data store that holds reductions of tasks. These are used by the planner for task decomposition.
– The *beliefs-DB* is a dynamic data store that maintains the agent's knowledge of the domain in which the plan will be executed. The planner uses the beliefs-DB during planning as a source of facts that affect its planning decisions. Actions may affect the beliefs-DB by changing facts in the domain.

3 The Planner Module

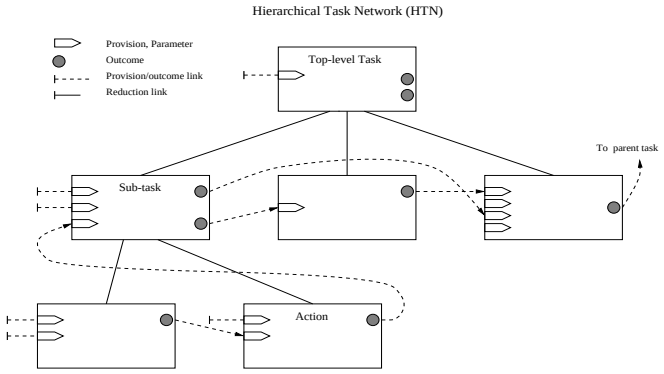

Fig. 2. A Hierarchical Task Network

The RETSINA Planner represents tasks using the Hierarchical Task Network (HTN) formalism [4]. Figure 2 displays the structure of an HTN. It consists of nodes that represent tasks and two types of edges. *Reduction links* describe the de-composition of a high-level tasks to subtasks (a tree structure). They are used to select the tasks that belong to the decomposition of the parent task. The second type of edges are *provision/outcome links* that represent 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 problem for the RETSINA Planner is defined by a tuple $< \mathcal{A}, \mathcal{C}, \mathcal{R}, \mathcal{B}, \mathcal{O}, \mathcal{T} >$, where $\mathcal{A}$ and $\mathcal{C}$ are sets of tasks schemas; $\mathcal{A}$ describes actions (primitive tasks) that the agent can perform directly, while $\mathcal{C}$ describes complex tasks that are performed by the composition of other primitive and complex tasks. $\mathcal{R}$ is the task reduction library. Each reduction schema in the library provides details on how to reduce a complex task in $\mathcal{C}$. A reduction schema for a complex task $C \in \mathcal{C}$ specifies the list of tasks that realizes *C*, and how their preconditions and effects are related to *C*'s preconditions and effects. In general, the correspondence between $\mathcal{R}$ and $\mathcal{C}$ is not one to one: there may be several reduction schemas for each complex task in $\mathcal{C}$, where each reduction schema

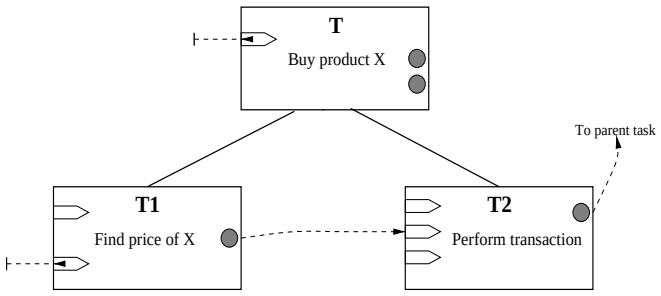

Fig. 3. An example of task de-composition

corresponds to one implementation of this task. $\mathcal{B}$ is the agent's beliefs-DB. It plays a role similar to the initial state's role in classical planning, though, when interleaving planning and execution are present (as in our planning mechanism), actions that are in execution stage may change facts in the beliefs-DB, thus affect the rest of the plan. $\mathcal{O}$ is the objective-DB, which holds the unachieved objectives of the agent. The goal of the planner is to remove all the objectives from this list. $\mathcal{T}$ is the task-DB which, by holding the tasks already added to the plan, describes the plan constructed by the agent.

```
RETSINA-Planner (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 flaws)
         then return P
      else do:
         remove a flaw f from P's objective-DB.
         partial-plans ← refinements of f in P
   return failure
```

Fig. 4. The Basic RETSINA Planning Algorithm

The detailed planning algorithm is described in Figure 4. It starts from an initial set of plans (*init-plans*) that provide alternative hypothesis 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 a flaw *f* from *P*'s objective-DB, to generate a new partial plan for each possible solution of *f*. This process is repeated until the planner generates a plan with an empty objective-DB. The planner fails if the list of partial plans empties before a solution plan is found.

The resulting plan is a tree of partially ordered tasks, similar to the plans generated by DPOCL [21]. The leaf nodes of the tree are actions in $\mathcal{A}$, while the internal nodes are

complex tasks in $\mathcal{C}$. 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 in the plan are used by the scheduler to synchronize the execution of primitive tasks as well as connection of the outcomes of computed tasks to the preconditions of tasks that were not yet executed.

3.1 Flaw refinement

The flaw refinement algorithm is shown in Figure 5. The RETSINA Planner allows three different types of flaws: task-reduction flaws, suspension flaws, and execution flaws. Task-reduction flaws are associated with unreduced complex tasks in the task-DB. They are used to signal which tasks in the current partial plan should be reduced. Once a reduction flaw is selected, the planner applies all task reduction schemas in $\mathcal{R}$ 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 task-DB $\mathcal{T}$. 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 $\mathcal{B}$.

```
refinements of f in P
   if f is a reduction flaw then
      t ← the task corresponding to f
      evaluate estimators and constraints of t
      for each reduction r of t do
         new-plans ← apply r to P
   if f is a suspension flaw then
      add f to the flaws of P
      new-plans add P
   if f is an execution flaw then
      a ← the action corresponding to f
      if a completed successfully
         new-plans add P
      if a failed
         new-plans ← nil
      if a still running
         add f to the flaws of P
         new-plans add P
   Return new-plans
```

Fig. 5. The Refinement Algorithm

Execution flaws are used to monitor the execution of actions while planning. An execution flaw is created and added to the objective-DB $\mathcal{O}$ whenever an action is created.

Execution flaws 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 flaw is simply removed from the list of flaws and no action is taken; otherwise, when the execution fails or times out, the partial plan also fails and the planner backtracks.

Suspension flaws 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 flaws are delayed and transformed into reduction flaws 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 Task and Plan Representation

A task is a tuple $< \mathcal{N}, \mathcal{P}_{ar}, \mathcal{D}_{par}, \mathcal{P}_{ro}, \mathcal{O}_{ut}, \mathcal{C}, \mathcal{E} >$ where $\mathcal{N}$ is a unique identifier of the task; $\mathcal{P}_{ar}$, $\mathcal{D}_{par}$ and $\mathcal{P}_{ro}$ are different types of preconditions as discussed below, $\mathcal{O}_{ut}$ is the set of outcomes of the task, $\mathcal{C}$ is a set of constraints that should hold either before, after or during the execution of the task, and $\mathcal{E}$ is a set of estimators used by the planner to predict the effects of the task on some variables. An example of task is shown in Figure 6. In this example, $\mathcal{N}$ = Buy Product, $\mathcal{P}_{ar} = \{\mathrm{B}alance\}$, $\mathcal{P}_{ro} = \{\mathrm{E}xpenses\}$, $\mathcal{O}_{ut} = \{\mathrm{P}urchaseDone\}$, $\mathcal{C} = \{\mathrm{B}alance > 0\}$, and $\mathcal{E} = \{\mathrm{B}alance = \mathrm{B}alance - \mathrm{E}xpenses\}$.

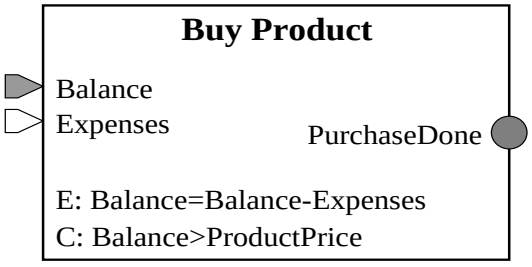

Fig. 6. The Buy Product task has a provision and a parameter (on the left), an outcome (on the right), and an estimator and a constraint (denoted E and C, at the bottom).

4.1 Estimators and Constraints

Estimators are used to evaluate the effect of a task on some variables. Estimators are used to predict the value of variables after a task is performs. Constraints are used to limit the values of variables to a specified range, when the plan would fail at execution time if such range is violated. For example, constraints and estimators are used to

evaluate the amount of resources needed to perform a task and to check whether these resources are available to the agent.

4.2 Parameters and Dynamic Parameters

Parameters are global variables stored in the beliefs-DB. They represent conditions that the planner expects to hold in the domain. Parameters are visible to all tasks. The value of parameters is monitored by the planner who modifies its plan when the value of a parameter changes. Since some conditions in the environment are modified by the agent, we distinguish a special class of parameters that we call dynamic parameters. Like parameters, dynamic parameters are visible to all tasks in the plan, however their value can change depending on the tasks performed by the agent without triggering the monitor.

There is an important distinction between parameters, dynamic parameters and precondition satisfaction in classical planning. Causal links [9] describe both how a precondition is achieved by the effect of a step in the plan, but they also describe a temporal precedence relation between the two steps. The satisfaction of parameters and dynamic parameters inherit the first aspect, but they do not express any temporal relation. The lack of a temporal dimension allows the representation of plans with concurrent actions that may consume the same resource. These plans cannot be represented in classical planning. Consider the following 2 steps of a plan for vehicle movement: `RunAirConditioner` and `GoToX`, both steps consume fuel. A representation in causal-link planning adds a causal link between the steps, thus imposing an order between them. As a result, either `RunAirConditioner` is executed first, and `GoToX` later, or, following the opposite order, the agent moves first in a hot environment and only when it arrives it runs the air conditioner to cool down. The RETSINA planner can generate a plan that overcomes this problem: it evaluates the estimators and constraints associated with both steps to compute how much fuel is needed, and to make sure that the agent has enough fuel to execute both actions. The temporal relation between the steps does not matter: if there is not enough fuel to run both the airconditioner and move, then the constraint on one of the actions will fail forcing the planner to backtrack.

4.3 Provisions and Outcomes

Parameters and dynamic parameters represent properties of the domain, however as explained above they do not include a notion for precondition satisfaction. Therefore, they do not offer a way to relate the preconditions of an action to the effects of another. In the RETSINA action representation, provisions and outcomes are used to describe the preconditions of a task and their effect.

Outcomes are conditions that are set by virtue of executing an action. As actions and complex tasks have a different behavior at execution time, outcomes mirror this difference: outcomes of an action are set by executing the method associated with the action, whereas outcomes of complex tasks are set by outcome propagation from the children to the parent task.

Provisions are local conditions of a task: they describe properties that should be satisfied for the action to be executable. Provisions are instantiated in one of two ways:

(1) they are set by precondition satisfaction when they receive a value from the outcome of a sibling task in the plan, or (2) they receive a value by inheritance from a provision in the parent task.

The relation between provisions and outcomes introduces the temporal precedence that is characteristic of precondition satisfaction in classical planning. Since outcomes propagate their values to provisions, the action that produces the outcome should be executed before the action that consumes the provision, thus the temporal precedence relation.

Provisions generalize the notion of run-time variables in Sage [7] and other planners [1, 6]. Run-time variables are assigned at execution time by running other actions in the plan. Provisions can be set both at execution time playing the role of run-time variables, or at planning time by inheritance from other provisions or parameters in the parent task.

4.4 Task Reduction Schemas

Task reduction schemas are used to describe how complex tasks are implemented by composition of other tasks. A reduction schema is a tuple $< \mathcal{N}_{task}, \mathcal{T}_{list}, \mathcal{I}_{links}, \mathcal{P}_{links}, \mathcal{O}_{links} >$. $\mathcal{N}_{task}$ is a unique identifier 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 similar to causal links in SNLP planning [9, 18], in that they show how the effects of one task affect the preconditions of another task. In addition, they 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 how the outcomes of the parent task *t* are affected by the outcomes of its children tasks.

5 Interleaving Planning and Information Gathering

The evaluation of estimators and constraints should be computed before the plan is completed. However when an estimator needs the value of a provision 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. For example, the agent should estimate the fuel needed to follow a route before moving. This estimation involves computing the length of the path, the expected fuel rate consumption and the amount of fuel available. The agent could use its own sensors as in reading the fuel gauge, and it can ask other agents what type or terrain and fuel consumption is expected on the way to the destination. The plan is not complete until both actions end and return their values.

The execution of information actions during planning is controlled by the suspension algorithm (Figure 7). 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 flaw *f* for *t*, marked as suspended. The flaw *f* 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 flaw *e* to monitor the outcome of t_π is added to the list of plan *P*'s flaws.

```
suspension of t in P
   f ← task-reduction flaw for t
   add f to the flaws of P
   set f as suspended
   set unsuspension trigger to a provision π
   Find task t_π that sets π
   Schedule t_π for execution
   e ← execution flaw for t_π
   add e to the flaws of P
```

Fig. 7. The Suspension Algorithm

As described above, the flaws *f* and *e* are not removed from the list of flaws until t_π completes its execution. The completion of t_π removes the suspension on *f*, which in turns allows *t*'s estimators and constraints to be evaluated and *t* to be reducted.

The use of suspension and monitoring flaws to control action execution has important consequences. First and foremost, it closely ties action execution and planning: since a plan is not completed until all flaws are resolved. The use of suspension and execution flaws 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 flaw. Furthermore, using flaws 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.

6 Example

In this example we show how the planner is used in RETSINA agents organized in a multiagent system that supports joint mission planning [3]. In this scenario, three army commanders discuss a rendezvous location for their platoons. Then, each commander constructs its own plan, assisted by a planningAgent that constructs a route, taking into account fuel limitations, ground and weather.

The system includes, among others, the following agents: the FuelExpertAgent that computes how much fuel is needed to accomplish a mission; the weatherAgent that provides weather forecasts, and a Matchmaker that matches service provider agents with consumer agents.

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

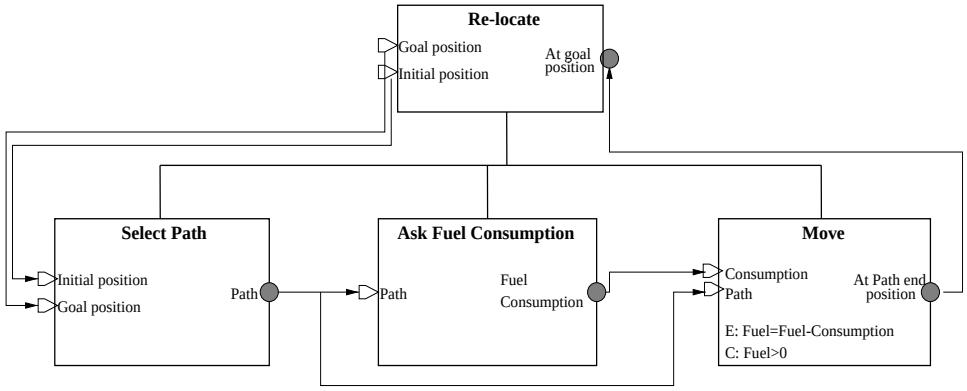

Fig. 8. An example of a task reduction schema

Each commander asks its planningAgent to find a route to the rendezvous point. PlanningAgents transform the request to an objective that is achieved using the reduction schema shown in Figure 8. Following the reduction schema, the plan adopted is **SelectPath**, **AskFuelConsumption**, and **Move**. Since the three actions can be further reduced, the planner adds three reduction objectives to the ObjectiveDB.

Following the reduction algorithm shown in Figure 5, reduction objectives trigger the evaluation of the estimators associated with the action being reduced. The estimator associated with **Move** depends on the value of the unknown provision `Consumption` that can be set only by executing **AskFuelConsumption**.

The execution of a step while planning is controlled by the suspension algorithm described in Figure 7. The planner first suspends the reduction of **Move** until `Consumption` is set; then it schedules **AskFuelConsumption** for execution. Since **AskFuelConsumption** needs the value of `Path`, **SelectPath** is also scheduled for execution.

Select Path is executed directly by the agent, while **AskFuelConsumption** is a request from information to the FuelExpertAgent. From the point of view of the planningAgent, there is no difference between the two tasks: a request to another agent is an action as any other.

The agent FuelExpertAgent computes the expected fuel consumption following decomposition schema shown in Figure 9. The result is a plan that contains three actions: **SurveyTerrain**, **ForecastWeather**, and **ComputeFuelUsage**. During the execution of the action **ForecastWeather** the FuelExpertAgent sends a query to the WeatherAgent that computes a weather forecast. The results returned by the WeatherAgent are passed back to the FuelExpertAgent that uses them to execute **ComputeFuelUsage**[4]. Finally, the FuelExpertAgent reports the result to the planningAgent.

The execution of **AskFuelConsumption** sets the value of `Consumption`, that releases the suspension of the reduction of the action **Move** which is finally reduced.

[4] Weather information is needed because tanks have different fuel consumption rates when they travel on different terrains, e.g. dry soil vs. wet soil

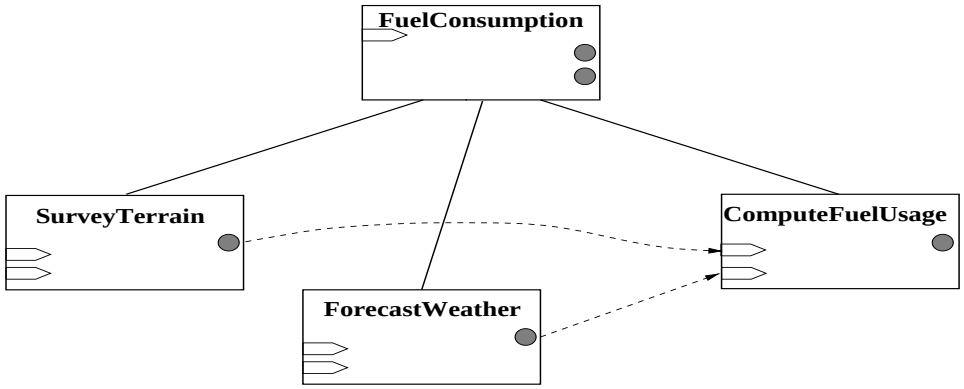

Fig. 9. An example of a task reduction schema

One aspect is important to stress: the computation of the fuel consumption is transparent to the planningAgent, which is unaware that the FuelExpertAgent needs additional information. Agents do not need to model how other agents solve problems or what they require to solve a problem.

Above, we assumed that the planningAgent knows that the FuelExpertAgent is part of the system. This assumption is too strong; RETSINA is an open system which agents join and leave dynamically. Agents joining the system advertise and unadvertise with middle agents[2]. The advertisement is a declaration of what tasks the agent can perform; whenever an agent wants to outsource parts of the computation it asks the Matchmaker, a type of middle agent, for contact information of agents that can perform the task.

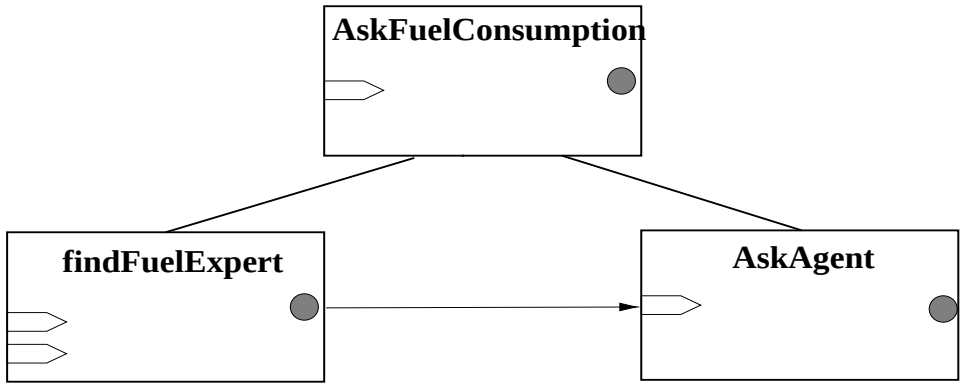

Fig. 10. Retduction Schema for AskFuelConsumption

Figure 10 shows the decomposition of the action **AskFuelConsumption**, it contains two sub-actions: **findFuelExpert** that is a request to the Matchmaker for the address of a fuel expert agent. The action **AskAgent** is a request addressing the fuel expert agent to provide the expected consumption. Since Matchmakers are agents, a request to a Matchmaker is solved as described above and it does not require additional computational machinery.

7 Monitoring Conditions during Planning

The construction of a plan and the scheduling and execution of the plan's actions take time. During this time the environment may change, invalidating the plan. In the example above, any change in the expected fuel consumption due to changes in the weather or other conditions, may lead the platoon to run out of fuel before its destination is reached.

RETSINA agents implement Rationale Based Monitors [17] to detect changes in the environment that are relevant to the plan. Specifically, constraints are the only mean for the planner to evaluate the validity of a plan: when a constraint fails, the plan is no longer valid. RETSINA agents monitor the value of parameters and provisions that are arguments of constraints in the plan; when one of the monitors detects a change, the agent re-evaluates its constraints to verify whether the plan is still valid.

Agents decide which action to monitor at planning time. When constraints are added to the plan, the agent looks for the actions that set the arguments of the constraint, then it transforms these actions into monitors: requests of information are transformed into requests to monitor, while sensing actions become monitoring actions. Monitors are implemented as information periodic gathering actions that iterate until they are stopped by the agent [20]. While virtually every AI planner uses exclusively "single shot" actions that are removed from the plan as soon as completed, RETSINA's periodic actions, are not removed from the plan. Rather, at the end of their execution, they are reinstantiated by the scheduler to run again. Monitors are stopped by the agent during plan execution when they do not have any associated constraint.

The reaction to a change in the environment depends on the state of the plan. If the domain change violates a constraint in a partial plan that has outstanding flaws, then the partial plan is no longer expanded because it is not valid and the planner backtracks. If, instead, the change violates a constraint in an action that is scheduled for execution, then the agent abandons the plan and constructs a new plan to fulfill the goal. Finally, violations of constraints of actions already under execution are not considered, the agent waits for the success or failure of the action.

8 Related Work

The RETSINA planner has some similarities with Knoblock's *Sage* [7], mainly in the concurrency of planning and information gathering and the close connection between the planner and the execution monitor through monitoring flaws. Nevertheless, the two planners differ in many important respects: while *Sage* is a partial order planner that

extends UCPOP [11], our planner is based on a HTN and plans by task reduction rather than from first principles; in addition, we extend the functionalities of the planner through the constant monitoring of the correctness of the information gathered and do re-planning 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* above, we use a different planning paradigm: HTN instead of SNLP style partial order planning. In addition we cannot assume the Local Close World Assumption because the information gathered might change while planning. Moreover, our planner supports a coarse description of information sources such as other agents. Specifically, the planner should know what they provide but not what their requirements are, since each agent is able to scout for the information it needs. This distinguishes the RETSINA planner from planners such as *XII* or *Sage*.

A completely different approach to planning and information gathering is followed by contingency planners [12, 3]. While we execute information gathering actions during planning, contingency planners plan for all possible outcomes of the information actions, then select the proper branch at execution time when the information is available. The two types of planners can be used to solve different problems: contingency planning is appropriate if information is very expensive or not available or unstable and changing rapidly; gathering information during planning, as discussed in this paper, is appropriate when the agent can gather reliable information while planning and monitor it fairly cheaply.

The RETSINA contribution to Rationale Based Monitoring is twofold. RETSINA describes how Rationale Based Monitors can be applied to HTN planning. In addition, RETSINA expands the use of monitors to the scheduling phase when the plan is completed and under execution, whereas Rationale Based Monitors in [17] are used only during planning time. On the other hand, RETSINA's use of monitors is more restrictive than in [17], because RETSINA agents do not attempt to select the best plan from a pool of alternative plans, given the modified environment.

The RETSINA planning process is a first step towards a distributed planning scheme based on a peer to peer cooperation between agents. No hierarchy or control relationships are present among the agents. They have each objectives to solve and they cooperate with one another to achieve their goals. We call this type of cooperation capability-based. From this prospective, our enterprise is very different from the planning architecture proposed in [19], where the planning process is distributed among agents, however they are centrally controlled.

9 Conclusion

Planning in a dynamic open MAS imposes a combination of problems that range from partial domain information to dynamism of the environment. These problems are each resolved, separately, by existing planning approaches; but no solution prior to the RETSINA planner addresses this combination as such. The RETSINA planner solves the problem of partial domain knowledge by interleaving planning and execution of information gathering actions; it handles dynamic changes in the domain by monitoring for changes

that may affect planning and execution; it supports cooperation by allowing query delegation to other agents; it enables re-planning when changes in the domain arise. These properties are provided by the architecture described in this paper. This architecture is implemented in RETSINA agents, which are deployed in several real-world environments [14, 16].

References

1. Jose' A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 83–88, St. Paul, MI, 1988.
2. Keith Decker, Katia Sycara, and Mike Williamson. Middle-agents for the internet. In *Proceedings of the Sixteen International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.
3. Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, pages 31–36, 1994.
4. Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, 1994.
5. R. James Firby. Tasks networks for controlling continuous processes: Issues in reactive planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS–94)*, 1994.
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. Craig A. Knoblock. Planning, executing, sensing and replanning for information gathering. In *Proceedings of the Fifthteen International Joint Conference on Artificial Intelligence (IJCAI95)*, 1995.
8. Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. Technical Report 93-06-03, University of Washington Department of Computer Science and Engineering, 1993.
9. David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, Anaheim, CA, 1991.
10. Jörg P. Müller. *The Design of Intelligent Agents*. Springer, 1996.
11. 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.
12. Mark Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on AI Planning Systems (AIPS-92)*, pages 189–197, College Park, MD, 1992.
13. 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.
14. O. Shehory, K. Sycara, G. Sukthankar, and V. Mukherjee. Agent aided aircraft maintenance. In *Proceeding of Agents-99*, Seattle, 1999.

15. 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.
16. Katia P. Sycara, Keith Decker, and Dajung Zeng. Intelligent agents in portfolio management. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology*, pages 267–283. Springer Verlag, 1998.
17. Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. A rationale-based monitoring for planning in dynamic environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS–98)*, 1998.
18. Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.
19. David E. Wilkins and Karen L. Mayers. A multiagent planning architecture. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS–98)*, 1998.
20. 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.
21. R. Michael Young, , Martha E. Pollack, and Johanna D. Moore. Decomposition and causality in partial-order planning. In *Proceedings of the 2nd International Conference on AI Planning Systems*, pages 188–193, Chicago, 1994.