A Planner for Agents in Open, Dynamic MAS: The RETSINA Planner*

M. Paolucci, O. Shehory, K. Sycara
Robotics, Carnegie Mellon University
D. Kalp, A. Pannu
Intelligent/Digital, Inc.
{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 Introduction

We are developing the RETSINA[1] Multi-Agent System (MAS) [Sycara *et al.*, 1996] 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. Moreover, to prevent a single point of failure and a computation and communication bottleneck, as single, centralized planner should be avoided. Therefore, as we suggest, the agents must each be provided with a planing component as part of its internal architecture. Yet local and distributed planning brings about other problems which require resolution: an agent's local plans, once found conflicting with other agents' local plans, must be revised and replanned 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 local planning (and execution) of agent i's may require that i suspends planning to wait for other agents to complete actions and provide results which are preconditions to the rest of i's plan. 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., [Peot and Smith, 1992]) or gathering information during planning (e.g., [Knoblock, 1995; Golden *et al.*, 1996]). Other planners deal with uncertainty in the domain by using probabilistic models [Kushmerick *et al.*, 1995] or by reacting to the environment in which they operate [Firby, 1994]. Open, dynamic multi-agent systems require that these problems be resolved simultaneously. Moreover, agents in a multi-agent system can take advantage of the collaboration of other agents. This is not supported by planners designed for single agent systems.

To resolve the above problems simultaneously we developed, as part of the internal architecture of a RETSINA agent, a new type of planner. This planner

*This material is based on work supported in part by MURI contract N00014-96-1222 and CoABS Darpa contract F30602-98-2-0138.

[1] REusable Task Structure based Intelligent Network Agents.

relaxes restrictive assumption common in classical planning. For instance, classical STRIP planning assumes that: 1) the planning agent has complete knowledge of the domain; 2) the domain is static, or changes occur at a pace much slower than planning and execution cycle. These assumption do not hold in dynamic, open MAS.

The planner presented in this paper relaxes several classical STRIPS assumptions. For instance, it relaxes the assumption of complete domain knowledge. The limitation on domain knowledge is resolved by allowing agents to incorporate into their plan actions to acquire information, by either inspection of the domain or querying of other agents. The classical assumption that the domain is static (or changes are slower than planning and execution) is relaxed by allowing for changes to occur. The resulting additional complexity is resolved by introducing into the plan actions for monitoring changes in the domain as well as results from other agents and for predicting how they affect the plan. Monitoring other agents and the domain triggers a re-evaluation of the plan and eventually re-planning.

Some additional advantages result from our approach, since multiple planners 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 the plan (or on other partial plans) while waiting for other agents to compute requested information.

In our planning approach we make two assumptions (less restrictive than the classical ones), as follows: agents are fully cooperative, i.e., they provide the correct information and fully answer requests; the rate of change in the domain allows agents sufficient time to re-plan a (partial) new solution, i.e., agents can behave deliberately and not reactively.

2 The RETSINA Architecture

RETSINA is an open MAS that provides infrastructure for different type of agents. Each RETSINA agent [Sycara *et al.*, 1996] is 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 the 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, enabling parallelism.

The following data stores are part of the architecture of each individual RETSINA agent and are used by the RETSINA planner.

- The *objective-DB* is a dynamic store. It stores the objectives of the agent of which it is a component. Objectives with the highest priority are handled first by the planner. New objectives are inserted by the communicator and by the planner. As a result of planning, new objectives may be created.
- The *task-DB* is a dynamic data store of all tasks. The tasks may still be not ready for execution or cannot yet be reduced, so they wait in the task-DB until the required conditions for their reduction or execution are set to true. When this happens, the actions are considered enabled, and are scheduled for execution by the scheduler.
- The *task schema library* is a static[2] data store that holds tasks schemas. These are used by the planner for task instantiation. The task schema library is typically created off-line by a designer of agent behaviors.
- The *task reduction library* is a static data store that holds reductions of tasks. These are used by the planner for task decomposition. As with the task schema library, the task reduction library is typically created off-line and consists of generic as well as domain-specific data items.
- The *beliefs-DB* is a dynamic data store that maintains the agent's knowledge of the domain in which the plan is 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.

The modules described above and the connections between them are depicted in Figure 1.

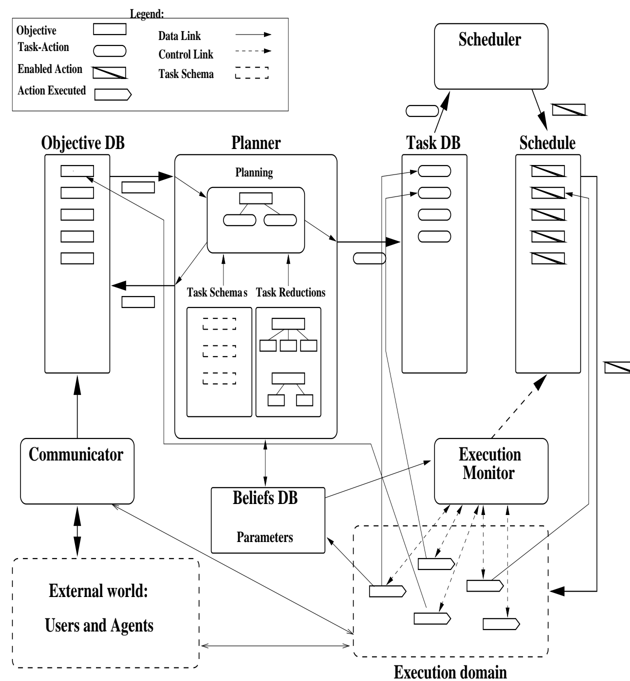

Figure 1: The RETSINA planning architecture.

[2]Task schema library and task reduction library are typically used statically. It is possible to use them dynamically.

3 The Planner Module

Agents in the RETSINA multi-agent architecture implement interleaving planning, information gathering and execution for goal satisfaction. During the planning process a RETSINA agent converts goals to tasks and gradually de-composes high-level tasks to sub-tasks. The lowest-level primitive tasks, (annotated as *actions*), are not decomposable and consist of executable code. Tasks at any level of the plan may each have several preconditions. The preconditions[3] are: *provisions*, *parameters* and *dynamic parameters*. Parameters and dynamic parameters are preconditions which are required for enabling the reduction of a task to sub-tasks. Having all of its parameters set, a task can be reduced regardless of the condition of its provisions. Provisions are preconditions which are necessary for the execution of actions. An action becomes executable when all of its preconditions, both parameters and provisions, are satisfied. Then it is scheduled for execution and executed when appropriate.

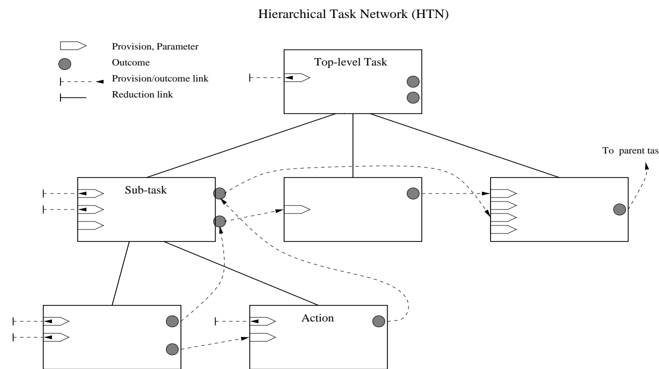

Figure 2: A Hierarchical Task Network

The RETSINA Planner uses a Hierarchical Task Network (HTN) formalization [Erol *et al.*, 1994]. HTNs are hierarchical networks as depicted in figure 2. They consist of task-nodes which are connected by two types of edges. Reduction link edges describe the de-composition of a high-level goal to tasks and subtasks (a tree structure). Provision/outcome link edges represent value propagation between task-nodes. Provision/outcome propagation allows one task T_1, as it completes execution, to propagate its outcome to the provision of a sibling task or to an outcome of the parent task. For instance, suppose T is the task of buying a product (see figure 3). T may de-compose to finding the price (T_1) and performing the transaction (T_2). The latter (T_2) requires that T_1 be executed, and therefore T_1 should propagate a price outcome to T_2 when completed successfully. This outcome is a provision for T_2. HTNs allow task-nodes to have multiple provisions and outcomes. A task is executable if it cannot be further reduced and all of its preconditions are set (as a result of either outcomes of other tasks or a setting from an outside source).

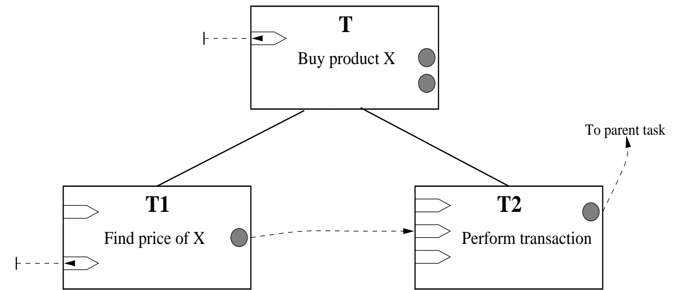

Figure 3: An example of task de-composition

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 the agent with details on how to reduce a complex task in $\mathcal{C}$. Specifically, a reduction schema for a complex task 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 corresponds to one implementation of this task. $\mathcal{B}$ is the agent's beliefs-DB. It plays a similar role as the initial state does 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.

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

```
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
```

Figure 4: The Basic RETSINA Planning Algorithm

[3]Details in the Task and Plan Representation Section.

goal. It 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 [Young *et al.*, 1994]. 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. It is based on the type of the flaw. The RETSINA Planner allows for 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. As a result, all the subtasks listed in the reduction schema are added to the partial-plan's task-DB $\mathcal{T}$. A new partial plan is generated for each successful application of task reduction. Task reduction triggers evaluation of constraints and estimators that are associated with the task being reduced. This estimation can trigger the execution of actions in the plan that inspect the environment and provide information not present in $\mathcal{B}$.

Execution flaws are used to monitor the execution of actions while planning. They are created when the action they monitor is scheduled for execution; they are removed from the list of flaws only when the action terminates. If the action terminates successfully, then the flaw is simply removed from the list of flaws; 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. They are delayed and transformed into reduction flaws only after an unsuspending event occurs. An example of an unsuspending event is the successful completion of the execution of an action that returns its outcome to the planner. In such a case, the data for which the planner was waiting becomes available, and the suspension flaw can be reduced.

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

```
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
```

Figure 5: The Refinement Algorithm

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, $\mathcal{P}_{ar} = \{\text{B}alance\}$, $\mathcal{P}_{ro} = \{\text{E}xpenses\}$, $\mathcal{O}_{ut} = \{\text{P}urchaseDone\}$, $\mathcal{C} = \{\text{B}alance > 0\}$, and $\mathcal{E} = \{\text{B}alance = \text{B}alance - \text{E}xpenses\}$.

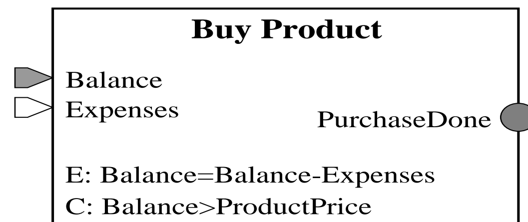

Figure 6: The Buy Product task has a provision and a parameter (left), an outcome (right), and an estimator and a constraint (denoted E and C, at the bottom).

The difference between the RETSINA representation and the classical one is justified by RETSINA's need to deal with a partially known dynamic environment as opposed to the deterministic and totally known evironment of classical planners. At execution time this distinction becomes evident: in the classic case, execution just steps through the plan constructed; RETSINA agents instead have to monitor plan execution to detect failures and differences from what was planned. The RETSINA action representation takes into account the split between planning time and execution time by providing tools to sim-

ulate the execution while planning, through Estimators and Constraints, to prevent the construction of invalid plans that would fail at execution time. Also, RETSINA distingushes between planning time preconditions (paramenters and dynamic parameters) and run time preconditions (provisions).

4.1 Estimators and Constraints

Estimators are used to evaluate the effect of a task on some variable, for example to predict the amount of resources that are expected to be needed by the agent to perform a task. Constraints are used to limit the values of variables to a specified range, because the plan would fail and execution time if such range is violated. Constraints combined with estimators provide a computational mechanism to compute how much resources are needed to perform a planned task and to check whether the agent indeed has all the needed resources.

4.2 Parameters and Dynamic Parameters

Parameters are global variables stored in the beliefs-DB and represent some condition that the planner expects to hold in the domain. Parameters are visible from all tasks and all tasks that use a particular parameter share the same value. The value of parameters is monitored by the planner that modifies its plan when the value of a parameter changes.

Parametes cannot represent conditions that are affected by the execution of the plan. For example, the amount of a resource such as fuel decreases when an agent-controlled vehicle moves from one place to another. If we represent fuel amount as a parameter and we monitor its value and replan whenever the value is changed, then a minor movement would consume fuel and trigger the monitor, thus forcing the planner to replan. To solve this problem we defined dynamic parameters. Like parameters, dynamic parameters are visible by 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 parameter, dynamic parameters and precondition satisfaction in classical planning. Causal links [McAllester and Rosenblitt, 1991] describe both how a precondition is achieved by the effect of a step in the plan, and also they 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 dimention 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 the agent moves in a hot environment and only when it arrives it runs the air conditioner. A RETSINA planner can generate a plan that overcomes this problem: it evaluates the estimators and constraints associated with both steps to compute the fuel needed and make sure 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.

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 behaviour 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 [Knoblock, 1995] and other planners [Ambros-Ingerson and Steel, 1988], [Golden *et al.*, 1996]. 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 varialbles, 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 as a composition of other complex and primitive 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 [McAllester and Rosenblitt, 1991], [Weld, 1994] 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

In general, 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 destiantion. 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 backtracking 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
```

Figure 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 completition 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 monitoring 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

Figure 8 shows the reduction schema for the action **Relocate** that moves an agent from a location `Initial position` to `Goal position`. Following the reduction schema, the agent moves to the goal position by selecting its path, estimating the amount of fuel needed to follow the path, and finally, moving along the selected path if fuel constraints are not violated. Applying the reduction adds three tasks to the plan: **Select Path**, **Ask Fuel Consumption** and **Move**. In addition, a reduction objective for each new step is added to the plan. Since the task **Re-locate** has no evaluators or constraints, its reduction is completed. The planner now turns to other objectives in the objective-DB and eventually it will fetch objectives corresponding to the actions **Select Path**, **Ask Fuel Consumption** and **Move**. For simplicity, we assume that the first two actions do not require decomposition. Since they have no estimator or constraint associated, the corresponding objectives are removed from the objective-DB without further ado; yet, the outcome `Path` in **Select Path** and the corresponding provision in **Ask Fuel Consumption** and the outcome `Consumption` are still unkown.

The action **Move** has an estimator and a constraint that should be evaluated, but they depend on the value of the unknown provision `Consumption`. The planner uses the suspension algorithm described above to find the needed information: the reduction of **Move** is suspended until `Consumption` is set to a value and **Ask Fuel Consumption** is scheduled for execution. Since **Ask Fuel Consumption** needs the value of `Path`, **Select Path** is also scheduled for execution. The execution of **Select Path** requires local computation, whereas the execution of **Ask Fuel Consumption** leads to asking another agent X to provide the estimate. From the point of view of our agent, there is no conceptual difference between these two actions. When the value of `Consumption` is provided (after agent X replies), the estimator and constraint of the **Move** task can be evaluated and if the constraint succeeds, the application of the reduction was successful, otherwise the planner backtracks to other so-

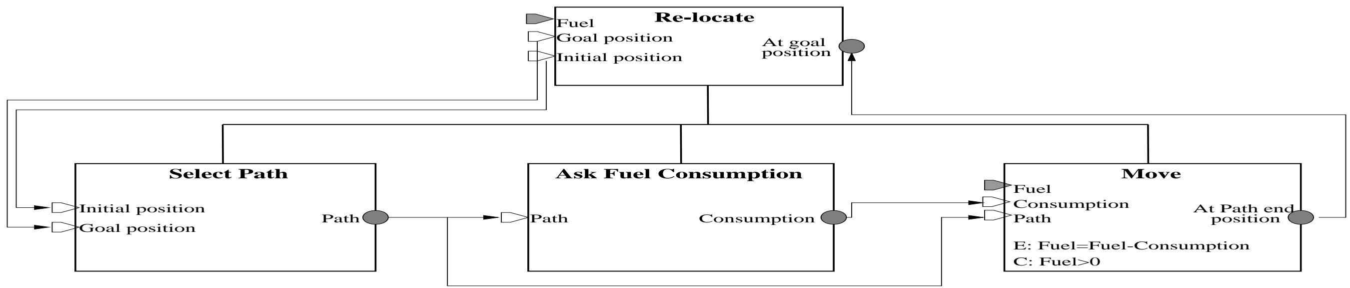

Figure 8: An example of a task reduction schema

lutions if they exist, or fails.

6.1 Multi-Agent Implications

The plan in the example is simple but may generate complex interaction among agents. Assume to agents (in addition to A): a fuel consumption expert agent F and a weather expert agent W. When A executes **Ask Fuel Consumption**, it sends a request to F to provide the required information. A does not need a model of how F computes the fuel consumption for the path or what additional information F needs for this computation. F, who employs the RETSINA planner too, automatically searches for its information needs. For instance, if the fuel consumption depends on the weather (due to, e.g., change in terrain conditions), then F will automatically query W for relevant weather information.

The example illustrates the difference between planning within distributed multi-agent systems and single agent (centralized) planning. A single agent system dealing with multiple sources of information should produce a plan that satisfies all inputs and outputs of all the sources of information. A plan generated for the example above will consist of a step there A requests W to provide information to F, and a proceeding step where F provides information to A. The RETSINA planner can produce this plan and, indeed, there are situations in which this plan is the best solution. Though, as shown in the example, our planner can also produce a distributed plan. Such a plan exploits the autonomy of agents in the system to distribute the planning effort. The advantages of producing a distributed plan are twofolds. First, there is a reduction in the amount of knowledge engineering needed to represent information sources. This results from the fact that the planner does not have to handle the input needs of other agents involved in the plan. Second, the computational overhead is reduced due to concurrency of planning, which in turn results from the agents running on different hosts.

7 Monitoring During Planning

A dynamic environment implies that when the planner concludes the plan, the actions are scheduled and the execution monitor proceeds with execution, the plan may be invalidated. This may result from value change of some assumptions on which the plan is based. For instance in the example above, if the weather changes so that the vehicle controlled by the agent consumes more fuel then estimated, then the vehicle may run out of gas before reaching its destination. We adopt a solution to this challenge by monitoring the values of parameters and provisions used in the plan. When the monitors detect a change in a value, the planner re-assesses the validity of the plan and, if a constraint in the plan is violated, the agent replans. This approach has some similarities to the Rational Based Monitoring [Veloso *et al.*, 1998], however differences are shown below.

The monitoring approach poses two problems: (1) among all conditions, which ones should be monitored? (2) for how long should each condition be monitored? In principle, every provision instantiated at planning time and every parameter and dynamic parameter may be monitored. In practice, this is overly conservative. At planning time, constraints are the only means to predict the plan's success. Parameters and provisions which are not involved in constraints do not change the initial prediction of the planner and they need not be monitored. The problem of terminating monitors activity is still open. In the Rational Based Monitoring, monitors expire when the plan is generated. In our case, actions executed by the agent consume time during which domain conditions may change, invalidating the plan. The solution adopted for the RETSINA planner is to monitor a condition until the task that uses the monitored value is executed. Our monitoring schema monitors only conditions that can invalidate the plan, whereas the schema presented in [Veloso *et al.*, 1998] is also aimed at plan optimization. Yet, our schema extends monitoring to include both planning and execution time.

8 Related Work

The RETSINA planner has some similarities with Knoblock's *Sage*, 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 planner differ in many important respects: while *Sage* is a partial order planner that extends UCPOP [Penberthy and Weld, 1992], our planner is based on a HTN and plans by task reduction rather then precondition satisfaction; in addition, we extend the functionalities of the planner through the constant monitoring of the correctness of the information gathered and re-planning when needed.

Other planners relax STRIP's omniscience assumption by interleaving planning and execution of information gathering actions, e.g., *XII*[Golden *et al.*, 1996]. Our approach is different. As in *Sage*, 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. In addition, 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 [Peot and Smith, 1992; Draper *et al.*, 1994]. 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 the reliable information while planning and monitor it fairly cheaply.

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. From this prespective, our enterprise is very different from the planning architecture proposed in [Wilkins and Mayers, 1998], where the planning process is distributed among agents, however they are centrally controlled.

9 Conclusion

Planning in a dynamic open MAS imposes a unique combination of problems. These are each resolved, separately, by existing planning approaches. However, 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 unique architecture described in this paper. This architecture is implemented in RETSINA agents, which are deployed in several real-world environments [Shehory *et al.*, 1999; Sycara *et al.*, 1998].

References

[Ambros-Ingerson and Steel, 1988] J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proc. of AAAI-88*, pages 83–88, St. Paul, MI, 1988.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering. In *Proc. of AIPS-94*, pages 31–36, 1994.

[Erol *et al.*, 1994] K. Erol, J. Hendler, and D. Nau. Htn planning: Complexity and expressivity. In *Proc. AAAI-94*, Seattle, 1994.

[Firby, 1994] J. Firby. Tasks networks for controlling continuous processes: Issues in reactive planning. In *Proc. AIPS–94*, 1994.

[Golden *et al.*, 1996] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. TR UW-CSE-96-01-09, Dept. of CS and Engeneering, Univ. of Washington, 1996.

[Knoblock, 1995] C. Knoblock. Planning, executing, sensing and replanning for information gathering. In *Proc. of IJCAI-95*, 1995.

[Kushmerick *et al.*, 1995] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 1995.

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. of AAAI-91*, pages 634–639, Anaheim, CA, 1991.

[Penberthy and Weld, 1992] S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of KR-92*, pages 103–114, 1992.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional nonlinear planning. In *Proc. of AIPS-92*, pages 189–197, College Park, MD, 1992.

[Shehory *et al.*, 1999] O. Shehory, K. Sycara, G. Sukthankar, and V. Mukherjee. Agent aided aircraft maintenance. In *Proc. of Agents-99*, Seattle, 1999.

[Sycara *et al.*, 1996] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert, Intelligent Systems and their Applications*, 11(6):36–45, 1996.

[Sycara *et al.*, 1998] K. Sycara, K. Decker, and D. Zeng. Intelligent agents in portfolio management. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology*, pages 267–283. Springer Verlag, 1998.

[Veloso *et al.*, 1998] M. Veloso, M. Pollack, and M. Cox. A rationale-based monitoring for planning in dynamic environments. In *Proc. of AIPS–98*, 1998.

[Weld, 1994] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Wilkins and Mayers, 1998] D. Wilkins and K. Mayers. A multiagent planning architecture. In *Proc. of AIPS–98*, 1998.

[Young *et al.*, 1994] M. Young, , M. Pollack, and J. Moore. Decomposition and causality in partial-order planning. In *Proc. of AIPS-94*, pages 188–193, Chicago, 1994.