An Abstraction-Based Approach to Interleaving Planning and Execution in Partially-Observable Domains

James Helwig **Peter Haddawy**
Decision Systems and Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
University of Wisconsin-Milwaukee
Milwaukee, WI 53201
{*helwig,haddawy*}*@cs.uwm.edu*

Introduction

An intelligent reactive planning agent for partially observable stochastic domains requires a number of diverse capabilities. First, the agent must be able to intelligently allocate its resources. This means that it must be able to decide how much time to allocate to deliberation in a way that is responsive to the environment in which the agent finds itself. It must also be able to decide when to sense and how much time and effort to spend sensing.

The agent must have flexible planning abilities. The planning algorithm must be anytime but should not be myopic. The planner must be able to reason under uncertainty due to incomplete information and due to the stochastic nature of available actions. It should be able to incorporate infomation obtained from sensors. It should be able to update the space of possible plans once an action is chosen for execution.

Finally, the agent must be capable of coordinate planning and acting. This means that it must be able to recognize when an action is completed and should be able to deliberate while acting. It also may need to translate between the high-level action representation used by the planner and low-level commands to its effectors.

Since the question of meta-level control to determine optimal deliberation times has been well studied, in this paper we focus on the required planning capabilities, as well as on the coordination of planning with execution.

Autonomous Vehicle Navigation Domain

We demonstrate our approach to interleaving planning with execution in the context of the problem of autonomous vehicle navigation. This domain has the attraction of encompassing many of the characteristics and challenges of real problems where an effective system that interleaves planning and execution would be desirable. The general problem consists of using a robotic truck to transport gold ore from a quarry to a factory. We simulate execution of this task using the TRUCKWORLD simulator (Hanks, Pollack, & Cohen 1993). This gives us the ability to model the domain realistically.

Even though the general task seems simple, the planning problem is non-trivial. The truck can obtain a global weather report but this incurs a cost and may produce false readings. Exogenous events such as rain may occur. There are several routes which can be used to travel to the quarry area, each with pros and cons. One route may be shorter, but may experience rain more frequently. Another route with moderate distance may be paved with gravel instead of asphalt. The choice of speed with which the truck travels affects the chance of having an accident. Once the quarry area has been reached, the particular quarry must be selected, where each quarry has a different mineral distribution. Before the truck grabs the rocks, it may exercise the option of using a mineral sensor which can give an indication of the likely mineral content of a rock. The truck may require refueling and the various locations available may have different associated fuel costs. The layout of the roads and locations in the example domain can be found in Figure 5.

System Architecture

Our abstraction-based system for planning integrated with execution, ASPIRE, is composed of three distinct modules: the controller, the planner, and the executor, as illustrated in Figure 1. Each module groups a set of conceptually similar tasks. The controller coordinates the actions of the other modules and processes the information received from them. The planner generates plans used by the controller. The executor provides the interface between our system and the world or simulator.

This modular design facilitates experimentation with approaches to planning and execution. Imposing different meta-level control strategies requires only making changes to the controller. Since information flow is regulated by the controller, the planner and executor are effectively isolated from one another. So, for example, if the planner is changed one need only change the interfact between the planner and controller. When interfacing the system with a new simulator, changes are isolated to the execution module

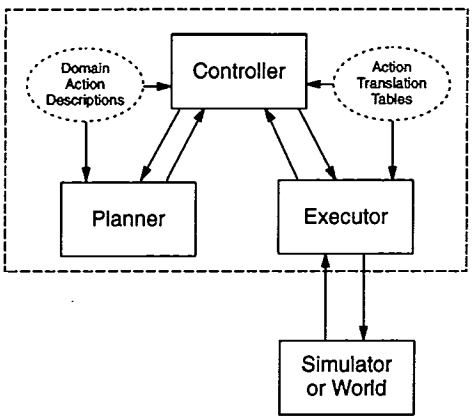

Figure 1: System architecture.

and the portion of the controller that interfaces to it.

- **Controller -** The controller performs the tasks of information management and coordination of the planner and executor. It must decide what the current model of the world is and when to update the planner's world model. It must decide what action to execute and when. The controller must the ability to recognize when a portion of a plan has failed and what steps should be taken for recovery.
- **Planner -** We assume that the planner is an anytime decision-theoretic planner that can plan in partially observable domains. It receives from the controller a planning problem and information about the state of the world. The planner can be stopped by the controller at any time to deliver the current planning results. The planner can then be presented with a revised planning problem and world model.
- **Executor -** The executor encapsulates the details of the interface between our system and the simulator or world. A primitive action within a plan is typically executed in the simulator by a sequence of commands. This translation is highly implementation dependent. The executor also retrieves information about the current state of the world. This is used by the controller to update its world model and determine the success of an action.

The DRIPS Planner

The planner we use is the DRIPS decision-theoretic refinement planning system (Haddawy & Suwandi 1994; Haddawy, Doan, & Goodwin 1995). DRIPS searches through the space of possible plans by using an abstraction hierarchy. By using abstraction, the planner is capable of eliminating suboptimal classes of plans without explicitly examining all the plans in a class. DRIPS finds the optimal plan by building abstract plans, comparing them, and refining only those that might yield the optimal plan. It begins with a set of abstract plans at the highest abstraction level, and subsequently refines the plans from more general to more specific. After each refinement, it eliminates suboptimal abstract plans. While an abstract plan is a complete plan in the sense that it is a complete sequence of actions, the actions in the plan are only partially specified in the sense that some of the attribute values are incompletely specified. In an appropriately structured abstraction hierarchy, the abstract action specifications describe only those aspects of the actions that have a high impact on expected utility. As the planner descends the abstraction hierarchy, it considers more of the less important details. The planner can be stopped at any time to yield the set of abstract plans which have not yet been shown to be suboptimal.

Representation

World Model We describe the world in terms of chronicles, where a chronicle is a complete specification of the world throughout time. We take time to be continuous and we describe chronicles by specifying the values of discrete and continuous attributes at various times, for example $\text{fuel}(t_0) = 10$. We express uncertainty concerning the state of the world with a set of probability distributions over chronicles. We express such a set by assigning probability intervals to (attribute, value) pairs at various times.

Action Model An action is represented by a finite set of tuples $\langle c_i, p_i, e_i \rangle$ called branches, where the c_i are a set of mutually exclusive and exhaustive conditions, the p_i are probabilities, and the e_i are effects. The intended meaning of an action is that if the condition c_i is satisfied at the beginning of the action then with probability p_i the effect e_i will be realized immediately following the action. This representation form is used in (Kushmerick, Hanks, & Weld 1994) and utilized by work in Markov Decision Process (Dean *et al.* 1993; Boutilier & Dearden 1994). In that work an action condition or effect is specified by a set of *propositional assignments*, such as *painted* $\wedge$ $\neg$ *hold-block*. We extend the representation by also allowing *metric assignments* in action conditions and effects, such as $fuel(t_2) = fuel(t_1) - 5$; we further allow branch probabilities p_i to be represented by intervals instead of single numeric points. These relaxations substantially enhance the expressiveness of the representation. We assume that changes to the world are limited to those effects explicitly described in the agent's action descriptions. A detailed discussion of the action and world models can be found in (Doan 1996).

On the left of Figure 2 are three example concrete action descriptions. The second branch of the *Drive slow on road 1* action says that if the weather is rainy (weather = 1) and the truck is ok (status = 0) then with probability 0.8 driving will consume 10 gallons of fuel (fule = fuel-10) and will take 240 minutes (time = time+240).

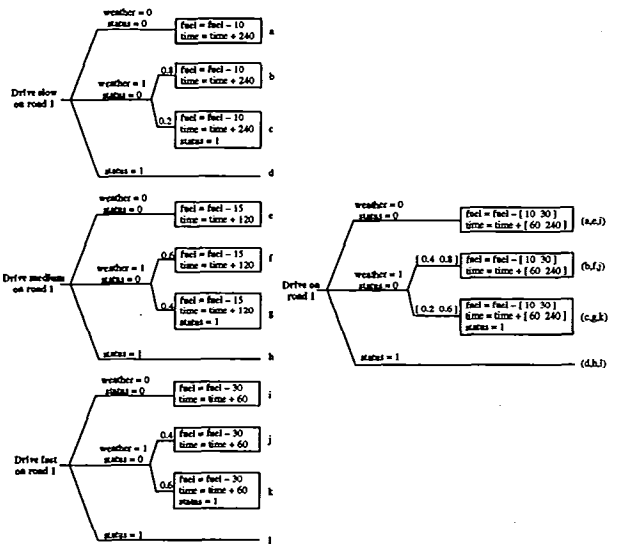

Figure 2: Concrete and abstract action descriptions.

Abstracting Actions

The DRIPS planner primarily uses two types of abstraction: interaction-abstraction and sequential abstraction. In this section, we briefly describe each. For a formal theory and detailed discussion of action abstraction that includes both these types of abstraction, see (Doan 1995).

The idea behind inter-action abstraction is to group together a set of analogous actions. The set is characterized by the features common to all the actions in the set. We then can plan with the abstract action and infer properties of a plan involving any of its instances. Formally, an inter-action abstraction of a set of actions $\{a_1, a_2, ...a_n\}$ is an action that represents the disjunction of the actions in the set. The actions in the set are called the *instantiations* of the abstract action and are considered to be alternative ways of realizing the abstract action. Thus the a_i are assumed to be mutually exclusive. In Figure 2, the action *Drive on road 1* is an abstraction of the three drive actions on the left of the figure.

A sequential abstraction is essentially a macro operator that specifies the end effects of a sequence of actions, as well as the initial conditions under which those effects are achieved, without specifying changes that occur as intermediate steps due to the individual actions within the sequence. Thus the information about the state of the world during the execution of the sequence of actions is abstracted away. We abstract an action sequence $a_1 a_2$ by pairing every branch of a_1 with every branch of a_2 and create an abstract branch for each pairing.

The DRIPS Planning Algorithm

A planning problem is described in terms of probability interval constraints on the values of attributes in the initial state, a set of action descriptions, and a utility function. The space of possible plans is described by an abstraction/decomposition network, supplied by the user. An abstract action has one or more sub-actions, which themselves may be abstractions or primitive actions. A decomposable action has a subplan that must be executed in sequence. The description of the abstract actions are created by inter-action abstraction and the descriptions of the decomposable actions are created by sequential abstraction. An example network is shown in Figure 3. A plan is simply a sequence of actions obtained from the net. The planning task is to find the sequence of actions for those represented in the network that maximizes expected utility relative to the given probability and utility models.

DRIPS finds the optimal plan by building abstract plans, comparing them, and refining only those that might yield the optimal plan. It begins with a set of abstract plans at the highest abstraction level, and subsequently refines the plans from more general to more specific. Since projecting abstract plans results in inferring probability intervals and attribute ranges, an abstract plan is assigned an expected utility interval, which includes the expected utilities of all possible instances of that abstract plan. An abstract plan can be eliminated if the upper bound of its expected utility interval is lower than the lower bound of the expected utility interval for any other plan. Eliminating an abstract plan eliminates all its possible instantiations. When abstract plans have overlapping expected utility intervals, the planner refines one of the plans by instantiating one of its actions. Successively instantiating abstract plans will narrow the range of expected utility and allow more plans to be pruned.

Given the abstraction/decomposition network, we evaluate plans at the abstract level, eliminate suboptimal plans, and refine remaining candidate plans further until only optimal plans remain. The algorithm works as follows.

1. Create a plan consisting of the single top-level action and put it into the set plans.
2. Until there is no abstract plan left in plans,
   - Choose an abstract plan P. Refine P by replacing an abstract action in P with all its instantiations, or its decomposition, creating a set of lower level plans $\{P_1, P_2, ..., P_n\}$.
   - Compute the expected utility of all newly created plans.
   - Remove P from plans and add $\{P_1, P_2, ..., P_n\}$.
   - Eliminate suboptimal plans in plans.
3. Return plans as the set of optimal plans.

If run to completion, DRIPS explores the entire space of possible plans. Since the planner only eliminates plans

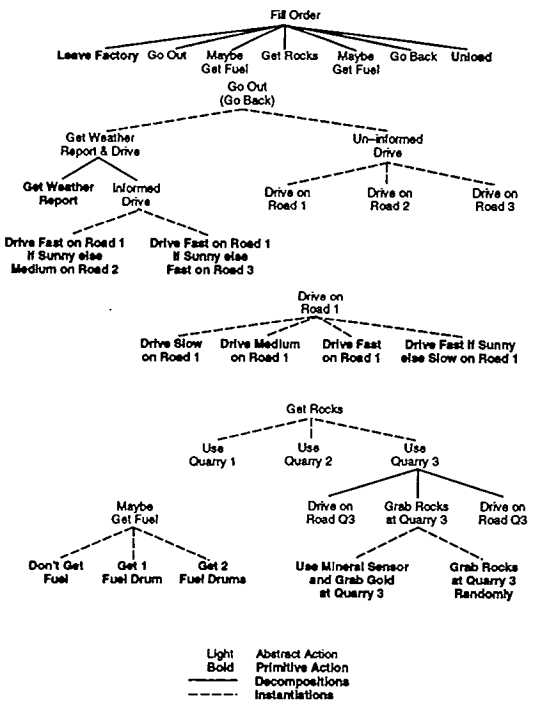

Figure 3: Abstraction/decomposition network.

it can prove are suboptimal and if a plan is suboptimal, it will eventually be eliminated by the planner, DRIPS is guaranteed to find the optimal plan or plans if it is run to completion.

As the algorithm proceeds, it maintains the set of abstract plans that cannot be eliminated because their expected utility intervals overlap. The set of instances of this set of abstract plans constitutes the current set of candidate plans. So the algorithm can be stopped at any time to yield the current set of candidate plans. The refinement process guarantees that the lowest EU of any plan in this set monotically increases as a function of computation time.

The DRIPS planner has been modified to support the interleaving of planning and execution by adding support for interrupting the algorithm, updating the world model, and changing the set of candidate plans. The main loop in DRIPS consists of selecting a plan to refine, refining an action within that plan, and eliminating any plans that are known to be sub-optimal. Our initial capacity to interrupt the planner is achieved by checking for a new command from the controller at the top of this loop. If no new command is found, the DRIPS proceeds with another refinement. At times, the controller may wish to have the planner wait at the top of the loop. This may be the case when new information about the world may be available momentarily. The planner then idles until a new command is received. Future versions of our abstraction-based system for planning and execution will add the ability to interrupt the DRIPS planner at points other than the top of the refinement loop.

After a sensing action is performed and the controller's model of the world has changed, the planner's world model must also be updated. The planner replaces its current model with one received from the controller. It then has to re-project each plan in the new world. Since this will likely change the expected utility of the plans, we can once again check to see if any may be eliminated.

Once the controller has committed the system to executing an action, it updates its set of candidate plans to reflect this commitment. When DRIPS receives this new set of plans from the controller it projects them in the current world and is prepared to resume planning.

Plan Execution and Simulation

The execution module of ASPIRE provides the interface with a physical device or a simulator. This allows the system to execute actions and obtain information about the state of the world. The executor is able to monitor the progress of an executed action, noting when it has completed. Every simulator has its own method of performing actions and obtaining information and these details are confined to the execution module.

Since we are interested in using the system in partially-observable domains, a simulator must be able to model the uncertainty of the world and the results of actions performed in the world. Our system is currently interfaced to the TRUCKWORLD simulator. TRUCKWORLD is a multi-agent test-bed designed to test theories of reactive execution. It simulates a robotic truck with the ability to drive around a world, pick up and transport objects, and sense world attributes. The world may contain exogenous events beyond the control of our system. Stochastic actions and noisy sensors can be modeled. The simulator provides a visualizer that allows the user to see the current state and position of the truck.

TRUCKWORLD uses very low level commands. As a result, in our domain, a primitive action from the plan being executed may actually be a sequence of up to ten TRUCKWORLD commands. The plan's primitive action can actually be viewed as a macro-action. The execution module translates the primitive actions to the appropriate simulator commands. For example, the action *drive fast on road 1 from location 1* is translated into the following TRUCKWORLD commands: *(arm-move arm-1 folded), (change-heading ne), (change-speed fast), (truck-move), (change-heading se), (truck-move)*. Developers of new domains or planning problems must provide a function that takes care of this translation.

When the controller decides to execute an action, it translates a primitive action into an executable action. Our execution module then receives this executable action from the control module. It retrieves the corre-

sponding sequence of TRUCKWORLD commands from the domain specific mapping and, prior to their execution, holds these in the *pending commands* queue. Since commands are generally sequential in nature, they are de-queued only when a currently executing command has terminated. When each command is in turn executed, it is de-queued and designated as the *transitional command.* The transitional command is sent to TRUCKWORLD, which confirms that the command has been initiated by responding with a unique token for tracking the command. The executor keeps track of the progress of this command by paring it with the token and placing it in the appropriate entry in list of *current commands.*

The current version of ASPIRE allows for some parallel execution of commands by having two separate *tracks* on which commands may be executed. The truck can be instructed to drive while at the same time sensing, as required by the action *drive while adjusting speed on road 1.* ASPIRE also allows commands to be executed conditionally or repeatedly. An example of a conditioonal command is *adjust speed*, which adjusts the speed of the truck depending on the current weather. The command *get a rock outside* is a loop command which repeats itself until five rocks have been grabbed.

TRUCKWORLD allows the user to access knowledge of the world through sensors which can be read. The scope, set of observable attributes, sensing duration, and noise level of the sensors are programmed when coding up the problem domain. Our example domain has a wide variety of sensors. The current condition of the road being traversed is reported by a condition sensor. A global weather sensor obtains information about the current weather of all sectors but is rather noisy while a local weather sensor determines the weather at only the truck's current location without erroneous results. Other "perfect" sensors include those that report the truck's location, speed, fuel level, and status. The truck also has a sensor which reports the positions of objects in its immediate neighborhood. This is useful when picking up fuel drums or rocks. A final sensor reports the probable mineral composition of individual rocks. This sensor has a fair amount of noise associated with it. The flexibility of sensor design allows the domain builder to model uncertainty in the world.

When sensor information is received by the executor from the simulator, it updates a record of current sensor readings. Upon request from the controller, it sends the list of readings. The controller is responsible for taking into account the accuracy of these readings when updating its world model.

Controller

The control module has a predetermined set of possible tasks which can be performed by our system. These tasks encompass the capabilities of each module within the system. The user is responsible for developing an algorithm which uses these available tasks to solve a problem. This specification forms the reactive planning and execution control strategy.

Tasks for the planner module include:

- *update worlds* - update planner's current world model.
- *update plans* - update planner's current set of candidate plans.
- *refine* - refine current set of candidate plans.
- *send plans* - send current set of candidate plans to the controller.
- *send state* - send planner's current state.
- *wait* - wait until issued a new command.

Tasks for the executor module include:

- *execute action* - execute the specified action in the simulator.
- *send-state* - send executor's current state

Tasks for information processing within the controller module include:

- *update plans* - update controller's plans with those from the planner.
- *select action* - select an action to execute.
- *commit to action* - edit the set of current candidate plans to reflect a commitment to executing the selected action.
- *mark time* - note a particular time in the future (e.g. 120 seconds from current time)
- *wait until* - idle until designated time is reached

The meta-level reactive planning and execution strategy is encoded in the control module. While the tasks the controller may wish to perform are fairly set, the question of how to best use these available task-options remains open. The basic decisions consist of what planning problem should be sent to the planner, how long to have the planner plan, what action should to execute, and how to recover from failures. Changing the strategy is facilitated by having these decisions localized to the control module.

Our initial reactive planning and execution algorithm was created by recognizing that problems can usually be divided into several distinct phases or stages. At different points during the problem solving process, different tasks may be appropriate. Each phase has a unique permutation of tasks suited for its particular role in the planning and execution process. When one phase has ended, a different phase begins. Developing a control strategy consists of organizing tasks into phases and then creating a mechanism for phase transition.

We identified five distinct phases for our control strategy. Three of the five are used in the beginning and end of the problem solving process. The other two are repeated throughout the process.

- *Initializing* - This phase performs initializing tasks in the three modules. Initial information about the planning problem and the initial state of the world is gathered, processed, and distributed.
- *Starting* - When the problem solving process begins, there exists a large number of candidate plans. It seems reasonable to spend a certain fixed amount of time planning at the start of the process to eliminate those plans which are particularly disastrous. In our particular problem, the initial action of leaving the factory is common to all plans so it can be executed immediately without deliberation. This phase performs that initial planning and action execution.
- *Updating and Selecting* - Part of the interleaving of reactive planning and execution requires that we update our current model of the world and select an action to execute. In this phase the controller processes any new information, gets the current set of candidate plans from the planner, selects an action to perform, and commits to that action.
- *Planning and Executing* - After information has been processed and a decision has been made, the action can be executed and additional planning may be performed. During this phase the planner is sent the revised set of plans and is instructed to continue refining while the executor executes the previously selected action. The controller keeps track of the amount of time spent in this phase and requires that it reach a certain minimum. This phase ends when the selected action has been completely executed and the planner has planned for at least the required minimal time.
- *Finishing* - This phase is entered when the plan has been fully executed. In it the controller shuts down the system.

Our control strategy has two different types of phases: *single-sequence phase* and *dual-sequence phase.* In a single-sequence phase there is only one sequence of tasks. Each task must finish before the next may begin. This behavior is required when one module requires information generated by a task in another module. For instance, in the *updating and selecting* phase, the controller must have the current set of candidate plans from the planner before it can select which action to execute. A single-sequence phase ends when its last task has ended.

A dual-sequence phase is one in which there can be two sequences of tasks operating in parallel. We see an example of this in the *planning and executing* phase, where the planner can be refining the current set of plans while the executor is executing the currently selected action. Each sequence of tasks is performed in the same manner as the sequence in a single sequence phase with the next task beginning when the previous one has ended. The sequences are independent of each other. While execution of one of the sequences will finish before the other, a dual-sequence phase ends when both of the sequences have ended.

When all the tasks associated with a phase have been completed, that phase ends and the controller then moves into another phase. While the sequencing of the tasks within each phase is fixed, the sequencing of phases may not be. Our strategy involves a combination of fixed and conditional phase transitions. Our strategy always starts with the *initializing* phase followed by the *starting* phase. After these have been completed, the heart of the interleaving strategy begins with the *updating* phase always followed by the *planning and executing* phase. When the *planning and executing* phase has ended, the controller checks to see if any further plans exist. If a plan or plans remain, the system once again enters the *updating* phase, otherwise it enters the final *finishing* phase. The complete reactive planning and execution strategy is described in the strategy diagram, Figure 4.

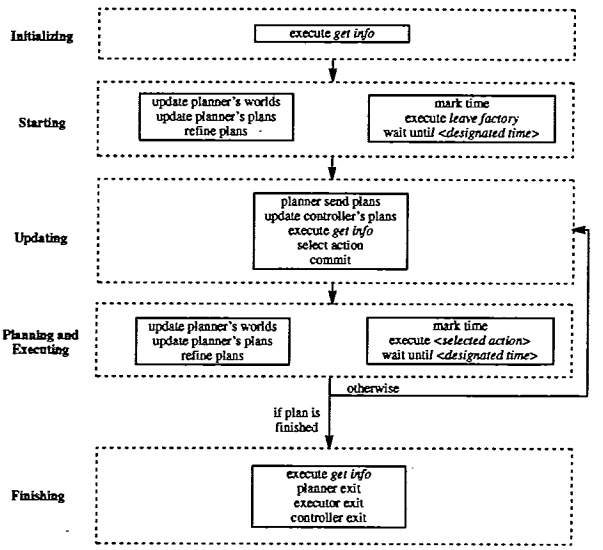

Figure 4: Planning and execution control strategy.

The control strategy indicates when tasks should be performed, but it must also determine the manner in which some tasks are performed. For example, the best method for selecting an action to execute remains open. In our strategy we examine the set of all *primitive initial actions* contained in the set of candidate plans. A primitive initial action is the first action in the sequence of primitive actions composing any concrete plan contained within an abstract plan. Our action descriptions within the hierarchy allow us to find the number of instances of each primitive initial action found in an abstract plan without explicitly examining each concrete plan. We look at the number of instances of all primitive initial actions and select the action with the highest rate of incidence.

This strategy seems reasonable for it allows us to commit to an action and throw out the minimum num-

ber of plans. The planner is guaranteed to only throw out sub-optimal plans but it does not indicate a relative ranking of the remaining candidate plans. By committing to a primitive initial action, we are throwing out any concrete plan which does not begin with that particular action. Since we have no way of knowing which concrete plan is optimal, short of explicit evaluation, and all remaining concrete plans are candidates, it seems reasonable to eliminate as few plans as possible. Each concrete plan is represented by having its instance of a particular primitive initial action recorded, selecting the primitive initial action with the highest rate of incidence retains the greatest number of, or removes the fewest, candidate plans, thus minimizing the risk of eliminating the optimal plan(s) from those under consideration.

Updating the Set of Candidate Plans and World Model

Once we have selected an action and have committed to its execution, we need to remove from the set of candidate plans any plans which are no longer valid. Since we can assume we will have executed that action, the only valid plans are those which had the selected action as their primitive initial action. Our procedure removes any plan that need editing and, if the plan is an abstract plan, replaces it with one or more refined plans. This continues until all the remaining plans are valid plans.

Procedure for Updating the Set of Candidate Plans

Mark all plans in the set of candidate plans as potential members of the set of valid plans.
Select a plan, P, from the set of candidate plans.
Examine the first action, A, in the plan's sequence.
If A is a inter-abstracted action,
  replace P with the set of plan sequences formed by each individual instance of A followed by the remaining actions from the original sequence P.
If A is a composed action,
  replace P with the plan sequence formed replacing the initial action A with the sequence it represents.
If A is a primitive action, not the selected one,
  remove the plan P.
If A is a primitive action, matching the selected one,
  replace P with the plan sequence formed by removing A from the original sequence, thus obtaining the plan which results after having executed A. Mark this plan as a certified member of the set of valid candidate plans.
Repeat this procedure until only certified plans remain in the set of candidate plans.

The real/simulated world is constantly changing. When information about these changes is received by the control module from the execution module, the control module must update its model of the world. The user is responsible for creating a mapping from attributes in the real/simulated world to attributes in the world model, i.e. a value of *rainy* for the weather of *sector-2* in TRUCKWORLD may correspond to a value of 1 for the ASPIRE attribute of *sector-2-weather*. The controller can then take new information and incorporate it into appropriate attribute values in the world model. In the current implementation of our abstraction-based system for planning and execution, the control module and planning module use the same world model. The controller can then simply pass its model of the world directly to DRIPS.

The attribute information of the real world comes to the controller in the form of a reading obtained as a result of a sensing action. Since these sensor reports are often affected by noise, we model the sensing action as one in which the current conditions result in a probability distribution over possible readings. In order to correctly update the controller's world model, the new information must be used in light of this noise and the previous belief of the attribute's value. We incorporate this information and obtain the new world model by projecting the results of the sensing action on the old world model, removing those worlds which are inconsistent with the actual sensor reading, and re-normalizing the remaining distribution.

Sample Run

To demonstate ASPIRE's operation we will trace through part of a planning problem in detail. The controller is presented with an initial belief about the state of the world and the abstract plan (*leave factory, maybe get fuel at depot 1, go out, get rocks, maybe get fuel at depot 2, go back, unload*), with the truck starting at the factory.

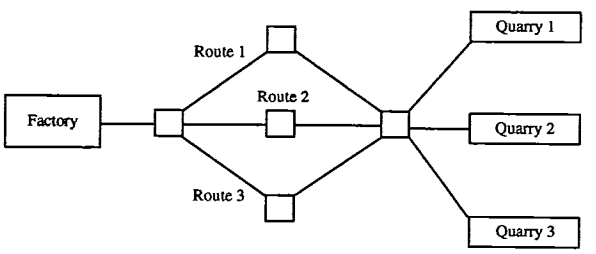

Figure 5: Map of the example world.

The system is first in the *initializing* phase. The controller gives the *execute get-info* command to the executor to get information on the current state of the truck. The system then enters the *starting* phase, a dual sequence phase. Since the first action, *leave factory*, is a primitive action, it is executed. At the same

time, the planner start refining the remaining actions in the sequence.

Once the truck has reached the first location where a route decision must be made. The system enters the *updating* phase where the controller gets the plans from the planner and selects an action to commit to. In this example the planner had refined the plans into the following:

(drive on road 1, don't get fuel, get rocks, maybe get fuel, go back, unload), (get forecast, maybe get fuel, informed drive, get rocks, maybe get fuel, go back, unload)

Plans containing the initial abstract actions of *drive on road 2* and *drive on road 3* have been removed. When the controller goes to select an primitive initial action it finds that these two abstract plans contain 6930 concrete plans. The distribution of primitive initial actions of concrete plans is the following:

2718 with *get forecast*
1053 with *drive slow on road 1*
1053 with *drive meduim on road 1*
1053 with *drive fast on road 1*
1053 with *drive fast if sunny else slow on road 1*

The controller selects *get forecast* as the action to execute. When the controller commits to this action the new set of abstract plans looks like this:

(maybe get fuel, informed drive, get rocks, maybe get fuel, go back, unload)

The *planning and executing* phase is entered. This time it is a short phase because *get forecast* action has a short duration, but we plan for a set minimum planning time of 2 minutes. The sensing action returned the following information:

```
(SENSOR-INFO SENSOR GLOBAL-WEATHER-SENSOR
 (((SECTOR-ID SECTOR-0) (WEATHER SUNNY))
  ((SECTOR-ID SECTOR-1) (WEATHER RAINY))
  ((SECTOR-ID SECTOR-2) (WEATHER SUNNY))
  ((SECTOR-ID SECTOR-3) (WEATHER SUNNY))
  ((SECTOR-ID SECTOR-4) (WEATHER SUNNY))))
```

The controller translates this into attribute information for the planner's world model:

```
WEATHER-1 : (((= WEATHER-1 1) 1))
WEATHER-2 : (((= WEATHER-2 0) 1))
WEATHER-3 : (((= WEATHER-3 0) 1))
WEATHER-4 : (((= WEATHER-4 0) 1))
```

The *updating* phase is entered again. Skipping ahead two cycles, we find the planner sending the following plans:

(drive fast on road 1 if sunny else medium on road 2, get rocks, maybe get fuel, go back, unload), (drive fast on road 1 if sunny else fast on road 3, get rocks, maybe get fuel, go back, unload)

The initial actions of each abstract plan is already a primitive action. The distribution is even and the controller selects *drive fast on road 1 if sunny else fast on road 3* and sends *(get rocks, maybe get fuel, go back, unload)* to the planner for refinement while entering the *planning and executing* phase. The selected action action is translated into the executable action *drive fast on road 1 from location 1* in light of the weather conditions and the truck's location. This action takes quite a while to execute and when the next *updating* phase is entered the planner has now refined the plans to *drive slow on road q1, grab rocks at quarry 1, drive slow on road q1, maybe get fuel, drive on road 3, unload.* Since the next three actions in the plan are primitive actions, during their execution the planner will be able to further refine the last action.

The system continues to cycle through the *updating* and *planning and executing* phases until the final action has been executed and the system exits.

References

Boutilier, C., and Dearden, R. 1994. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1016–1022.

Dean, T.; Kaelbling, L. P.; Kirman, J.; and Nicholson, A. 1993. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 574–579.

Doan, A. 1995. An abstraction-based approach to decision-theoretic planning for partially observable metric domains. Master's thesis, Dept. of EE & CS, University of Wisconsin-Milwaukee.

Doan, A. 1996. Modeling probabilistic actions for practical decision-theoretic planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96).*

Haddawy, P., and Suwandi, M. 1994. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings, Second International Conference on AI Planning Systems*, 266–271.

Haddawy, P.; Doan, A.; and Goodwin, R. 1995. Efficient decision-theoretic planning: Techniques and empirical analysis. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 229–236.

Hanks, S.; Pollack, M.; and Cohen, P. 1993. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magizine* 14(4):17–42.

Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1073–1078.